



BWBEV: A Bitwise Query Processing Algorithm for Approximate Prefix Search

Edleno S. de Moura   [Federal University of Amazonas | edleno@icomp.ufam.edu.br]

Berg Ferreira  [Federal University of Amazonas | berg@icomp.ufam.edu.br]

Altigran da Silva  [Federal University of Amazonas | alti@icomp.ufam.edu.br]

Ricardo Baeza-Yates  [Northeastern University | rbaeza@acm.org]

Received: 16 March 2024 • Accepted: 08 September 2024 • Published: 27 October 2024

Abstract We tackle the challenge of conducting an approximate prefix search within datasets of strings. We explore using a bit-parallelism technique to compute the edit distance between distinct strings and illustrate its adaptation for an approximate prefix search procedure referred to as *BWBEV*. This technique employs a unary representation of edit vectors alongside bitwise operations to efficiently update these vectors during the edit distance computation. We also show how to apply our new bit-parallelism technique strategy to online edit distance computation between strings without index structure. Our experiments with *BWBEV* applied to approximate prefix search for a query autocompletion task revealed a substantial acceleration of over 36% when contrasted against state-of-the-art methods.

Keywords: bit-parallelism, autocomplete, trie, error tolerance.

1 Introduction

Search systems are the core in many current applications such as e-commerce services, search engines [Alaofi *et al.*, 2022], and embedded in-vehicle interfaces [Zhong *et al.*, 2022]. However, even nowadays, these applications have some challenges in efficiently finding relevant results for their users. For example, around 10-15% of searches submitted to a search system have typing errors [Cucerzan and Brill, 2004], also when the users do not have enough knowledge about the application, they use the try-and-see approach [Ji *et al.*, 2009] and spend more time searching to relevant results.

An essential component adopted in the interaction between the user and the search system is the Query Autocompletion (QAC) system, which can guide users in choosing high-value queries to submit to the search system. Also, they help to reduce from 40% to 60% of typing effort on average [Ji *et al.*, 2009] and to correct errors in typing time, being an important component of usability, especially in mobile applications, where these devices have tiny keyboards and users can easily produce typographical errors.

The QAC systems suggest full queries based on a typed prefix and consist of the phases of matching and ranking. The matching phase receives the prefix and selects the strings similar to the prefix based on a threshold. The ranking phase sorts the matching results according to a score function and selects the top most relevant results for the user. For more details about QAC systems, see Cai and de Rijke [2016]; Phophalia [2011], which presents a detailed survey about query autocompletion in information retrieval.

An example of an error-tolerant QAC system is shown in Figure 1. In this example, the user receives suggestions such as “smartphone”, “smartphone samsung”, “smartphone xiaomi”, and “smartphone 5g”, all of which match the misspelled typed prefix query “smarph”. The user can continue

typing the whole query and the system needs to present efficiently suggestions for each keystroke.

smarph	send
smartphone	
smartphone samsung	
smartphone xiaomi	
smartphone 5g	

Figure 1. Example of an error-tolerant query autocompletion system.

We here focus on the matching phase approaching the issue of finding efficiently all items from a large set of string keys that match a given search prefix, while allowing a limited maximum number of matching errors. This study assumes that the string keys are previously stored in a large dataset and the system allows for approximate matching. In the following, we formally defined the problem that we are addressing for the matching phase:

- Σ being an alphabet composed of a finite number of symbols;
- p being a prefix query composed of symbols in Σ and $|p|$ denoting the length of p ;
- $\mathcal{S} = \{s_1, \dots, s_n\}$ being a set of strings to be searched and $s[1, j]$ denoting a prefix of s , where $1 \leq j \leq |s|$. For example, for $s = \text{“auto”}$ we have $s[1, 1] = \text{“a”}$, $s[1, 3] = \text{“aut”}$ and $s[0] = s[5] = \epsilon$ (an empty string out of the border);
- $p = s[1, |p|]$ being an exact prefix match between p and s ;
- $ed(p, s)$ being the *edit distances* given by the minimum number of insertions, removals, or substitutions of symbols required to transform s to p or vice-versa. For example, $ed(\text{“ant”}, \text{“auto”}) = 2$ as we can transform “ant”

to “auto” by an insertion of “o” in the end and a substitution from “n” to “u”;

- $ped(p, s) = \min_{j=|p|-\tau}^{|p|+\tau} ed(p, s[1, j])$ being the *prefix edit distances* between the two strings p and s and a maximum number of edit distances τ . For example, given a $\tau = 1$, the $ped(\text{“ant”}, \text{“auto”}) = \min(ed(\text{“ant”}, \text{“au”}), ed(\text{“ant”}, \text{“aut”}), ed(\text{“ant”}, \text{“auto”})) = 1$.

The *error-tolerant prefix search* consists of finding all strings $s_i \in S$ such that there is a prefix match between p and s_i with a maximum edit distance τ or more formally: $\mathcal{R} = \{s_i \mid s_i \in S \wedge ped(p, s_i[1, j]) \leq \tau\}$, where $|p| - \tau \leq j \leq |p| + \tau$;

The problem of approximate prefix search arises in various practical scenarios, including DNA fragment search in large DNA datasets, and approximate entity instance search in a large dataset (e.g., people names, locations, etc.). However, the specific application we are interested in is searching large datasets of query suggestions in a QAC system. The key challenge in the context of QAC systems is the need to suggest error-tolerant queries around or below 100ms [Miller, 1968] as the user types character by character to avoid noticeable delays during the user’s search session.

To efficiently support error-tolerant in QAC systems, existing methods [Chaudhuri and 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., 2020] adopt efficient data structures such as Trie [Fredkin, 1960] and Burst Tries [Ferreira et al., 2022] to index the strings in-memory. In search time, they compute the trie nodes whose edit distances to the prefix query are within the threshold, called *Active Nodes*, then the system proceeds with the *fetching* operation by traversing the trie to locate all leave nodes reachable from the active nodes, thus retrieving the corresponding results. The management of the active nodes set is an important factor in the efficiency of these methods. Another important factor is the way to calculate the edit distance between two strings, even in Learning-based approaches [Wang et al., 2018]. In this paper, we focus on improving the edit distance calculation.

1.1 Contributions

We present a new method to QAC systems called *Bitwise Boundary Edit Vector (BWBEV)* based on a bit-parallelism approach to calculate efficiently the Levenshtein [Levenshtein, 1966] edit distance between two strings. We incorporate BWBEV for approximate prefix search by improving the ideas presented in the method *Boundary Edit Vector Automata (BEVA)* [Zhou et al., 2016], one of the state-of-the-art methods for QAC systems. While BEVA uses an automaton referred to as *Edit Vector Automata (EVA)* to compute approximate prefix matching, we replace it with our bit-parallel approach and speed up the time processing over 36% when contrasted against state-of-the-art methods. This result allows us to use even larger datasets and more workloads in QAC systems.

We also present experiments with BWBEV when applied to the problem of computing the online edit distance between a pair of strings without using an index structure. While this second task was not our target application, the experiments

are useful to show the practical scenarios where our algorithm is competitive. For instance, it presents a competitive performance when the compared strings are expected to be different from each other in most of the cases, and when the number of errors allowed in the comparison is small.

The remainder of this article is organized as follows. Section 2 presents a review of the main related work. Section 3 explains related concepts and some definitions necessary to understand our proposed ideas. Section 4 presents details on the proposed method BWBEV. Section 5 presents experiments to compare the proposed method with baselines. Section 6 presents our conclusions and possible future research directions.

2 Related Work

2.1 Computing the Edit Distance

To enhance our comprehension of recent methods for approximate prefix search, let us begin by introducing how to adopt dynamic programming to compute the Levenshtein edit distance [Levenshtein, 1966] or just edit distance between strings p and s , with lengths n and m respectively. It populates a matrix denoted as M of dimensions $(n+1) \times (m+1)$. The following recurrence relation enables the computation of cell values in a single pass, either row-wise or column-wise:

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

where $\delta(a, b) = 0$ if $a = b$, and 1 otherwise. The values assigned to the boundaries are $M[0, j] = j$ and $M[i, 0] = i$. In Table 1 we show how to obtain the distance between the words “ant” and “auto”. We use the convention of placing the prefix query string horizontally and the data string vertically in the matrix. The edit distance between the two strings can be obtained by simply extracting the value from the cell position $M[n, m]$ within the matrix. The time complexity to calculate is $O(n \cdot m)$.

		0	1	2	3
		ϵ	a	n	t
0	ϵ	0	1	2	3
1	a	1	0	1	2
2	u	2	1	1	2
3	t	3	2	2	1
4	o	4	3	3	2

Table 1. When calculating the edit distance between “ant” and “auto” the dynamic programming matrix is utilized.

Ukkonen [1985] made a significant observation: the edit distance computation can be performed only on the matrix elements situated within the *k-diagonals*. Here, k ranges from $-\tau$ to τ , where τ represents the maximum edit distance allowed. In Table 2 we show the diagonals -1, 0 (in dark gray), and 1 for $\tau = 1$ and the words “ant” and “auto”. The time complexity to calculate is $O(\tau \cdot \min(n, m))$.

		0	1	2	3
		ϵ	a	n	t
0	ϵ	0	1	2	3
1	a	1	0	1	2
2	u	2	1	1	2
3	t	3	2	2	1
4	o	4	3	3	2

Table 2. k -diagonal definition for strings “ant” and “auto” and $\tau = 1$.

2.2 Approximate Prefix Search in Tries

The general concept behind recently proposed approximate prefix search methods involves calculating the edit distance operation between strings. Nevertheless, calculating this operation between every pair of strings individually would result in excessive computational costs, rendering real-time search results unfeasible. Therefore, these methods typically employ an index structure to simultaneously compute the edit distance between the entered prefix query and the entire string dataset. In this section, we explain in more detail the main approximate prefix search baseline methods and how they perform fast prefix search.

When examining the existing literature on approximate prefix search methods for retrieving results from large datasets, it becomes evident that several approaches [Li *et al.*, 2011; Deng *et al.*, 2016; Zhou *et al.*, 2016; Qin *et al.*, 2020] employ tries, initially proposed by Fredkin [1960], or some trie variations as an indexing data structure. Generally, most methods employ breadth-first search (BFS) to traverse the trie and a result list is generated for every character entered during the processing of the prefix query. The key idea in these methods involves the management of the active nodes set, where each node is associated with a trie node and corresponds to matches that satisfy a specified error limit τ . Various algorithms proposed in the literature adopt this approach, differing in how they handle the active nodes set.

The size of the active node set that is required to maintain during approximate prefix search can be exceedingly high, leading to a slower search process. ICPAN, presented by Li *et al.* [2011], introduces a trie-based technique that reduces the size of the active node set when compared to the methods previously proposed by considering only a subset of active nodes that do not involve the substitution or deletion of the last characters, ICPAN reduces both memory consumption and query response time.

In another effort to minimize the computational expenses of computing active nodes, Deng *et al.* [2016] proposed META, which also supports top- k query matches. META uses a compact tree index to maintain the set of active nodes, thus avoiding the redundant computations that occur in previous trie-based methods.

In their research, Zhou *et al.* [2016] introduced BEVA, which accelerates query processing by storing the *edit vector* values of each active node, enabling the storage of a minimal set of active nodes, known as *boundary active nodes*. The edit vectors are structures to support the edit distance calculation and are explored in detail in Section 3. In our study, we adopted BEVA as a baseline to explore the application of a bit-parallelism approach and we provide in Section 3.2 a

detailed explanation of how the BEVA method works.

Xiao *et al.* [2013] proposed IncNGTrie. It calculates the edit distance by detecting a common prefix between two strings and deleting a few characters in the prefix until the prefixes are the same. This approach significantly reduces the number of prefixes to be considered and speeds up the query response time.

It is worth noting that memory usage and index size are crucial factors to consider when designing autocomplete algorithms, especially for large datasets. Although the IncNGTrie algorithm is efficient in query processing time according to experiments reported by Xiao *et al.* [2013], it requires indexing several nodes for a single word, leading to a higher memory consumption compared to other algorithms in the literature. While the authors proposed a reduction in the number of nodes indexed by eliminating duplicate nodes, the algorithm still uses a large amount of memory, making the index size a severe restriction to the use of IncNGTrie.

To address this issue, Qin *et al.* [2020] proposed an improvement to IncNGTrie that reduces the index size. While their method still requires more memory than BEVA and META, their new proposal reduces the memory requirements of IncNGTrie, but increases the index building times. The trade-off between memory consumption and indexing time should be carefully considered when choosing an autocomplete algorithm for a specific application.

Ferreira *et al.* [2022] demonstrate how to adapt trie-based algorithms for approximate prefix search to use burst-tries [Heinz *et al.*, 2002]. This adaptation technique enables the methods to maintain almost the same performance as when using the full trie while significantly reducing the additional space required by the index. The burst tries experimented with by the authors demand only a small fraction of the space required by the full tries.

Hu *et al.* [2018] proposed a trie-based method that allows combining location-aware and approximate query autocomplete. Wang and Lin [2020] extended the ICPAN [Li *et al.*, 2011] method and propose a method called *AutoEL* to support approximate location-aware query autocomplete. The approximate 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.

2.3 Pattern Matching using Bit-Parallelism Approach

The bit-parallelism approach has two main applications: (1) It parallelizes the work of the non-deterministic automaton that solves the pattern-matching problem and (2) It parallelizes the work of the dynamic programming matrix. There is a variety of algorithms developed to perform approximate string or prefix search. Part of them is explored in detail in Navarro [2001].

The first of them is Shift-OR [Baeza-Yates and Gonnet, 1992]. The algorithm performs a string search by simulating the computation of a non-deterministic automaton by parallelizing its operations. Baeza-Yates [1999] represents the same automaton presented in Shift-OR, by using unary arithmetic to represent its states. We here adopt a similar idea for

representing each value for the k -diagonals of the dynamic programming matrix by using a unary representation. Several other authors have further explored the bit-parallelism approach by extending the initial idea [Navarro and Raffinot, 2001; Peltola and Tarhio, 2003; Durian *et al.*, 2009].

Wright [1994] introduced the first approach using bit-parallelism in dynamic programming matrices. The concept focuses on secondary diagonals from the upper right to the bottom left, where each new diagonal can be computed using the two previous ones. This algorithm stores differences using mod 4 and updates many diagonal cells in parallel through vectorized comparisons of pattern and text characters. Myers [1999] presented a similarly straightforward algorithm, requiring only $O(|\Sigma| + nm/w)$ time by computing a bit representation of the relocatable dynamic programming matrix, being $|\Sigma|$ the alphabet size, w the computer word and n and m two any strings. The algorithm's performance is consistent regardless of k , making it more efficient than previous methods for various choices of k and small m . The Myers's algorithm is one of our baselines to compute the edit distance.

Hyyrö [2003] proposed a novel approach inspired by Ukkonen's diagonal restriction method, where vertical delta vectors are tiled diagonally instead of horizontally by shifting the vertical vectors upwards before processing each column with complexity $O(|\Sigma| + \lceil \tau/w \rceil m)$. Furthermore, the algorithm explicitly maintains all values along the lower boundary of the filled area of the dynamic programming matrix. This involves setting values for diagonally consecutive cells and horizontally consecutive cells based on specific conditions. Hyyrö's algorithm is another of our baselines to compute the edit distance.

In this paper, we are interested in parallelizing the work of the dynamic programming matrix by using the compact representation of the k -diagonals proposed by Ukkonen [1985] and applying arithmetic operations over the bits.

3 Preliminaries

3.1 Edit Distance using Edit Vectors

We here better define the Edit Vector (EV) proposed by Zhou *et al.* [2016] adopted to compute the edit distance. Edit vectors serve as compact representations of the dynamic programming matrices utilized for edit distance calculations. Zhou *et al.* [2016] have shown the correctness of their algorithm for computing edit distance by using only edit vectors. They noted that a raw edit vector v_j , considering a threshold value of τ , corresponds to a vector of $2\tau + 1$ positions located at the j -th column of the dynamic programming matrix as shown in the Table 3. In this Table, each element $v_j[i]$ within the vector holds a value ranging from 0 to τ , indicating a reported match with $v_j[i]$ errors, or the value $\tau + 1$ represented by the symbol #, indicating a mismatch. The edit vector for column 0 consistently takes the form $[\underbrace{\tau, \tau - 1, \dots, 1, 0}_{2\tau+1}, \underbrace{1, 2, \dots, \tau}_{2\tau+1}]$, as the word in column 0 is empty. This characteristic is labeled as the *initial edit vector* and represented by V_0 . Correspondingly, the vector containing all values of $\tau + 1$, represented as

$[\underbrace{\tau + 1, \tau + 1, \dots, \tau + 1}_{2\tau+1}]$, is labeled as *final edit vector* and represented by V_\perp .

		p			
		0	1	2	3
s	0 ϵ	1	0	1	1
	1 p	0	1	1	1
	2 l	1	1	1	#
	3 a			#	#
	4 n				#
		V_0	V_1	V_2	V_\perp

Table 3. Edit vectors represented in yellow and green for the strings p and s and $\tau = 1$

The computation of the threshold edit distance involves calculating the j -th edit vector concerning a given threshold value τ , starting from $j = 0$, using the following equation:

$$\begin{aligned} v_{j+1}[i] &= \min(v_j[i] + \delta(p[j+1], s[j-\tau+i]), \\ &\quad v_j[i+1] + 1, \\ &\quad v_{j+1}[i-1] + 1), \forall 1 \leq i \leq 2\tau + 1. \end{aligned} \quad (2)$$

For instance, let us consider the strings $p = \text{"pet"}$ and $s = \text{"plan"}$ in Table 3 for $\tau = 1$. Then, the calculation of the new edit vector V_1 from V_0 follows:

$$v_1[1] = \min(1 + \delta(p, \epsilon), 0 + 1, \tau + 1) = 1 \quad (3)$$

$$v_1[2] = \min(0 + \delta(p, p), 1 + 1, 1 + 1) = 0 \quad (4)$$

$$v_1[3] = \min(1 + \delta(p, l), \tau + 1, 0 + 1) = 1 \quad (5)$$

Finally, the calculation of the edit distance between the string p and the data string s is determined as $v_{|s|}[\tau + 1 + (|p| - |s|)]$ when $|p| \in [|s| - \tau, |s| + \tau]$ or more than τ otherwise.

The edit vectors were defined in a context where the authors were interested in deriving a method for search on large string datasets, as part of the method BEVA described in the next section. While Zhou *et al.* [2016] have not explicitly considered the possibility of calculating the edit distance, the comparison between two strings can then be computed by only computing the values of the edit vectors, given a maximum error threshold τ . We adopt this algorithm in the experiments and name it as *EV* algorithm, which can be considered as a variant of Ukkonen's algorithm.

3.2 Query Processing in BEVA

In this section, we explain in more detail the query processing in BEVA [Zhou *et al.*, 2016]. BEVA is a method adopted for processing a prefix query search allowing errors on a large string dataset. BEVA adopts an automaton strategy to

compute the prefix search results. We describe BEVA in detail because we adopted it in the next section to use our bit parallelism approach instead of their original automaton approach.

The BEVA employs a technique wherein the edit vector values of each active node are stored in a structure called Edit Vector Automata (EVA), enabling them to maintain a minimal set of boundary active nodes necessary for conducting the edit distance computation. Hence, a boundary active node is always associated with an edit vector in the EVA structure. The calculation of the edit vector v_{j+1} was modeled by Zhou *et al.* [2016] as a function $f(v_j, b)$, where b is a bitmap of $2\tau + 1$ bits and $b = \neg\delta(p[j+1], s[j-\tau+i])$. For example, suppose the edit vectors of Table 3 for calculating the transition function $\neg\delta(p, \epsilon) = \mathbf{0}$, $\neg\delta(p, p) = \mathbf{1}$, $\neg\delta(p, l) = \mathbf{0}$, the bitmap $b = \mathbf{010}$. Then $f([1, 0, 1], 010) = [1, 0, 1]$.

The control of the bitmaps during query processing in BEVA [Zhou *et al.*, 2016] is carried out in a table \mathcal{H} that represents the characters that appeared or not in the prefix query. The size of \mathcal{H} can be at most the size of the alphabet. The update of \mathcal{H} is performed as the prefix query changes, i.e. the bitmaps have shifted 1 bit to the left. New characters are added to \mathcal{H} with bitmap 001, and in the iterations following it is only updated. A bitmap is removed from \mathcal{H} if it gets the value 000 after updating the bitmaps.

For example, suppose that the edit distance threshold is $\tau = 1$ and the last $2\tau + 1 = 3$ characters of the prefix query is “lov”. The dynamic table of bitmap \mathcal{H} maps “l” to 100, “o” to 010, “v” to 001, and all the other characters of the alphabet to 000. If a new character “e” is appended to the prefix query “lov” changing to “love”, thus with the last 3 characters of the prefix query being now “ove”. We shift leftwards all the bitmap values, making the mapping of “l” become 000, “o” 100, and “v” 010. Finally, we mark the bitmap entry of “o” in \mathcal{H} as 001, meaning that it matches with the last character of the prefix, and does not match with the two previous ones.

The authors of BEVA proposed a structure called Edit Vector Automata (EVA) that precomputes all the edit vectors that can be generated based on bitmaps, being a bitmap and a previous edit vector just the key to finding the correct new edit vector in this structure. The query processing in the BEVA method then is performed together with the EVA structure, and consequently, each node in the trie is associated with an edit vector incorporated in the automaton. In the following, we describe the main Algorithms 1 and 2 used in BEVA to query processing.

Algorithm 1 receives the c , $|p|$ and \mathcal{B} as parameters, which represents the current character from prefix query p , the length of p , and a boundary active nodes set from the previous character or empty when $|p| = 0$, respectively. Initially, in line 3, the global table of bitmaps \mathcal{H} is updated. When $|p| = 1$, the unique active node is the root node associated with the initial edit vector, as described in lines 4 and 5, which remains active until the prefix query p exceeds a threshold of τ characters ($|p| > \tau$) typed by the user. Subsequently, the method computes and stores the new set of boundary active nodes after each character is typed by making a scan in each child of \mathcal{B} in the trie, using the Algorithm 2.

In the Algorithm 2, the existing list of boundary active nodes becomes inactive upon the addition of a new charac-

Algorithm 1 Process prefix query p

```

1: procedure Maintain( $c$ ,  $|p|$ ,  $\mathcal{B}$ )
2:    $\mathcal{B}' \leftarrow \mathcal{B}$ 
3:   updateBitmap( $c$ )
4:   if  $|p| = 1$  then
5:      $\mathcal{B}' \leftarrow \langle r, V_0 \rangle$   $\triangleright r$  is the trie's root node.
6:   else if  $|p| > \tau$  then
7:      $\mathcal{B}' \leftarrow \emptyset$ 
8:     for each  $\langle n, S \rangle$  in  $\mathcal{B}$  do
9:        $\mathcal{B}' \leftarrow \mathcal{B}' \cup \text{findActiveNodes}(|p|, \langle n, S \rangle)$ 
10:  return  $\mathcal{B}'$ 

```

ter to the prefix query p . A scan is then performed in each of their children in the trie to compute their respective edit vector values by building a bitmap (line 6) and calling the transition function f (line 7) that just accesses the Edit Vector Automata (EVA) to retrieve the correspondent edit vector. Subsequently, the algorithm classifies the edit vector and the associated node as either terminal, inactive, or active node based on their edit vector values as follows:

- *terminal* - denotes the state of a node when it has no chance to activate other nodes.
- *inactive* - denotes the state of a node when it does not represent a match, although its edit vector value suggests the potential activation of one of its children.
- *active* - denotes the status of a node when it is included in the new list of boundary active nodes for the prefix query.

Algorithm 2 Find active nodes

```

1: procedure FindActiveNodes( $|p|$ ,  $\langle n, V \rangle$ )
2:    $\mathcal{B}' \leftarrow \emptyset$ 
3:    $level \leftarrow n.level + 1$ 
4:    $k \leftarrow |p| - n.level$ 
5:   for each  $child\ n'$  of  $n$  do
6:      $b_{n'} \leftarrow \text{buildBitmap}(|p|, level, n'.char)$ 
7:      $V' \leftarrow f(V, b_{n'})$ 
8:     if  $V' \neq V_{\perp}$  then  $\triangleright V_{\perp}$  is the final edit vector
9:       if  $V'[\tau + 1 + k] \leq \tau$  then
10:         $\mathcal{B}' \leftarrow \mathcal{B}' \cup \langle n', V' \rangle$ 
11:     else
12:        $\mathcal{B}' \leftarrow \mathcal{B}' \cup \text{findActiveNodes}(|p|, \langle n', V' \rangle)$ 
13:  return  $\mathcal{B}'$ 

```

Nodes classified as inactive have their children recursively scanned until either active or terminal nodes are found in all paths derived from them, as described in lines 8 to 12. The recursive process extends up to $2\tau + 1$ levels in the trie as matches can be identified for paths ranging from $|p| - \tau$ to $|p| + \tau$. After completing 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.

We here present a novel approach that uses the trie-based approximate match algorithm proposed in the BEVA method. But instead of using the edit vectors automaton (EVA) for fast computation of edit vector values on each active node,

represents the edit vectors with a bit unary representation and applies a bit-parallelism approach to calculate efficiently the edit distance between two strings during the query processing. Here we focus on improving the transition function f performed in line 7 of the Algorithm 2. Through experiments, we demonstrate that this combination results in a method that is considerably faster than BEVA and has the advantage of not requiring extra space to store the EVA.

4 BWBEV

In this section, we present a new method called **Bitwise Boundary Edit Vector (BWBEV)** to edit distance calculation. BWBEV replaced the EVA structure in BEVA with an algorithm for computing the edit vectors using bitwise operations. BWBEV performs the calculation of the Equation 2 using an efficient bit-parallelism approach. Then, from now we demonstrate how to calculate the Equation 2 efficiently using our proposed ideas. The complete source code can be found at GitHub repository¹.

The formal notation is given by w , being the length of the computer word (in bits). The sequence $b_1 \dots b_m$ is the bits of a mask of length m . We use the exponentiation to denote bit repetition (e.g. $0^2 1^2 = 0011$). We use the C-style syntax to denote the bitwise operations. The operations are $|$ to denote the bitwise-or, $\&$ to denote the bitwise-and, \ll to denote the bitwise-shift-left, which moves the bits to the left and enters zeros from the right, i.e. $b_m b_{m-1} \dots b_2 b_1 \ll r = b_{m-r} \dots b_2 b_1 0^r$ and \gg to denote the bitwise-shift-right, which moves the bits to the right and enters zeros from the left, i.e. $b_1 b_2 \dots b_{m-1} b_m \gg r = 0^r b_1 b_2 \dots b_{m-r}$.

4.1 Unary Representation

To accelerate the calculation of a new edit vector from a previous edit vector, the same operation we described in Section 3.1, we first propose and utilize a fixed unary representation to pack several values of each position of an edit vector into a single computer word w . In our representation a number k is written using n bits as a sequence of k consecutive zeros at left, followed by a sequence of $n - k$ bits with value one. Table 4 presents an example of representing numbers from 0 to 3 using 3-bit numbers. With this unary representation, we can convert the edit vector V_0 in Table 3, for instance, from $[1, 0, 1]$ to the number ‘011 111 011’ (the blank space is only for better visualization but it does not exist in the actual representation). From now on, the bit edit vectors are represented as just a number (an unsigned long in C++) and denoted by v .

4.2 Arithmetic Operations for Edit Vectors

To accelerate the update of edit vectors using bit-parallel operations, we demonstrate some arithmetic operations over the bits as the *addition* of 1 to all positions of an edit vector using our fixed-length unary representation, as well as how to compute the minimum operation in parallel.

Decimal	Unary
0	111
1	011
2	001
3	000

Table 4. Example of a unary fixed length code with 3 bits. Each number is represented by a sequence of bits set to zero followed by bits with value 1.

4.2.1 Add 1

Table 5 illustrates the addition process. To add 1 in parallel to all positions of an edit vector, we first perform a right shift of 1 bit on it, and then apply a $\&$ (AND) operation with the control mask to prevent 1’s from the end of a given position of the vector to be carried to the following position after the shift operation. The control mask value is a bit mask with value $[0[1]^\tau]^{(2\tau+1)}$, where r^n denotes the binary sequence r repeated n times. For instance, if $\tau = 2$, the control mask becomes $[0[1]^2]^{(5)}$, corresponding to ‘011 011 011 011 011’ in binary, with each 3-bit value representing a mask to match one of the positions of the edit vector with 5 positions.

In the example where $\tau = 2$, the bit edit vector v starts with ‘001 011 111 011 001’, representing five 3-bit fixed unary numbers, and thus the decimal values represented are $[2, 1, 0, 1, 2]$. After the shift and the $\&$ operation with the control mask, the final value of v becomes ‘000 001 011 001 000’, representing values $[3, 2, 1, 2, 3]$, as shown in Table 5.

	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$
Initial decimal values	2	1	0	1	2
v	001	011	111	011	001
$v \gg 1$	000	101	111	101	100
$[0[1]^\tau]^{(2\tau+1)}$	011	011	011	011	011
$(v \gg 1) \& [0[1]^\tau]^{(2\tau+1)}$	000	001	011	001	000
Final decimal values	3	2	1	2	3

Table 5. Adding 1 to all positions of an edit vector v in parallel. Example considering $\tau = 2$.

Notice that the proposed add operation yields a convenient result of $\tau + 1$ when we add 1 to $\tau + 1$, since this is the maximum value reached by each position of the edit vector when using the proposed unary representation.

4.2.2 Minimum Between Two Unary Numbers

Furthermore, to compute the *min* operation between two unary numbers u and v , we only need to perform a bitwise $|$ (OR) operation between v and u , as shown in Table 6.

	[1]	[2]	[3]	[4]	[5]
Initial decimal values u	2	1	0	1	2
Initial decimal values v	2	2	1	1	2
u	001	011	111	011	001
v	001	001	011	011	001
$u v$	001	011	111	011	001
Final decimal values	2	1	0	1	2

Table 6. Applying min operation between two unary numbers u and v .

¹<https://github.com/vdberg/BWBEV>

4.2.3 Align Positions

To align position $i + 1$ of an edit vector v with position i , we can shift $v \ll \tau + 1$ bits left, ie, $v \ll \tau + 1$. Similarly, to align the position $i - 1$ with position i , we shift $v \gg \tau + 1$ bits right, ie, $v \gg \tau + 1$. These operations can be performed in parallel for all positions of the edit vector, as shown in Table 7. Another advantage of our edit vector representation is that checking whether its current value represents a match or not is a low-cost operation. An edit vector represents a final edit vector with a mismatch when all positions have a value $\tau + 1$, which means this status can be detected when $v = 0$.

	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$
Initial decimal values	2	1	0	1	2
v	001	011	111	011	001
$v \ll \tau + 1$	011	111	011	001	000
Final decimal values	1	0	1	2	3
$v \gg \tau + 1$	000	001	011	111	011
Final decimal values	3	2	1	0	1

Table 7. Aligning the position $i + 1$ of v with position i using the bitwise operation $v \ll \tau + 1$ and aligning the position $i - 1$ of v with position i using the bitwise operation $v \gg \tau + 1$.

With our unary representation to pack each edit vector value and the arithmetic operations described, we can perform efficiently the computation of the Equation 2 using bit parallel operations. However, we observe that when two computed strings are completely different, ie. there is a complete mismatch between the two strings, we can simplify the Equation 2 to accelerate the edit distance calculation. This improvement is described in the next section.

4.3 Optimizing the Edit Vector Computation

We now show how to optimize the edit vector computation whenever the bitmap b is zero, which might be a quite common situation in practical applications. To show that this modification does not change the edit vector computation, thus assuring the correctness of our edit distance computation, We observe that whenever the bitmap b is zero, which means $\delta(p[j + 1], s[j - \tau + i]) = 1$, Equation 2 can be replaced by:

$$v_{j+1}[i] = \min(v_j[i] + 1, v_j[i + 1] + 1, v_{j+1}[i - 1] + 1) \quad (6)$$

But,

$$v_{j+1}[i - 1] + 1 = \min(v_j[i - 1] + 2, v_j[i] + 2, v_{j+1}[i - 2] + 2) \quad (7)$$

Taking the well-known property that $|v_j[x] - v_j[y]| \leq |x - y|$ for any given valid value positions x and y , we have that $v_j[i] + 2 > v_j[i] + 1$, and $v_j[i - 1] + 2 \geq v_j[i] + 1$, as a consequence of Equations 6 and 7, we have:

$$v_{j+1}[i] = \min(v_j[i] + 1, v_j[i + 1] + 1, v_{j+1}[i - 2] + 2) \quad (8)$$

repeating the same reasoning $\tau + 1$ times, we obtain:

$$v_{j+1}[i] = \min(v_j[i] + 1, v_j[i + 1] + 1, v_{j+1}[i - (\tau + 1)] + (\tau + 1)) \quad (9)$$

And as $\tau + 1$ is the maximum value achieved by an edit vector position, we can remove it from the Equation and have:

$$v_{j+1}[i] = \min(v_j[i] + 1, v_j[i + 1] + 1) \quad (10)$$

This result is significant because it enables us to reduce the computational cost of computing the v_{j+1} from v_j . We also observed that in query autocompletion tasks, our target application, the prefix queries are usually small and the vocabulary is large, making the scenario of b equals zero quite common. Therefore, if we can perform *add 1* and *min* operations in parallel for all cells of the edit vector, we can compute the new edit vector in parallel for such scenarios.

4.4 BWBEV Algorithm

We now present our algorithm for computing new edit vector values using bit parallelism. Algorithm 3 illustrates how to compute a new edit vector v_{j+1} , given the current edit vector v_j , the bitmap b indicating whether there is a match or not in each position of the prefix query and the maximum number of allowed errors τ . v_j and v_{j+1} represent the edit vector positions using the unary representation described in Section 4.1, and contain $2\tau + 1$ positions, each of them represented in a $\tau + 1$ number coded as a fixed unary number. The algorithm starts by assigning to $v_{j+1}[i]$ the *min* value between $v_j[i] + 1$ and $v_j[i + 1] + 1$ using a small set of bitwise operations. Notice that this operation is performed in parallel for all positions $\forall 1 \leq i \leq 2\tau + 1$ (lines 2 and 3). We should shift $v_{j+1} \tau + 1$ bits to the left, but since adding 1 requires a shift right of 1, we only shift left τ bits at line 2. If b is zero, the value of v_{j+1} is already computed, and can be returned. If not, the algorithm finishes the computation of each position $v_{j+1}[x]$ by computing the minimum between the already computed value and $v_j[x]$ for each position x where b indicates a match (lines 5 to 11). Finally, we update the value of each position $v_{j+1}[x]$ with the minimum between the already computed value and value of $v_{j+1}[x - 1] + 1$ (lines 12 to 15). To align $v_{j+1}[x - 1]$ with bits of $v_{j+1}[x]$, we need to shift right $\tau + 1$ bits and to add one to the elements we need an extra shift, thus a total $\tau + 2$ shift is required at line 14.

The BWBEV algorithm was specially designed for the context of a QAC system but is important to highlight that this algorithm can also calculate the online edit distance between any two strings in a most general context. In Algorithm 4, we describe the changes necessary to process edit

Algorithm 3 Computes v_{j+1} from v_j .

```

1: procedure computeNewEditVector( $v_j, b, \tau$ )
2:    $v_{j+1} \leftarrow (v_j \gg 1) \mid (v_j \ll \tau)$ 
3:    $v_{j+1} \leftarrow v_{j+1} \& [0^{1^\tau}]^{(2\tau+1)}$ 
4:   if  $b \neq 0$  then
5:      $mask \leftarrow 1^{\tau+1} 0^{2\tau \times (\tau+1)}$ 
6:     do
7:       if  $b \& 1[0]^{2\tau}$  then
8:          $v_{j+1} \leftarrow v_{j+1} \mid (v_j \& mask)$ 
9:          $mask \leftarrow mask \gg (\tau + 1)$ 
10:         $b \leftarrow b \ll 1$ 
11:      while  $b \neq 0$ 
12:      do
13:         $tmp \leftarrow v_{j+1}$ 
14:         $v_{j+1} \leftarrow v_{j+1} \mid ((v_{j+1} \gg (\tau + 2)) \& [0^{1^\tau}]^{(2\tau+1)})$ 
15:        while  $tmp \neq v_{j+1}$ 
16:      return  $v_{j+1}$ 

```

distance between any two strings and we refer to this algorithm as *Bitwise Edit Vector (BWEV)*.

When implementing both methods, we have empirically verified the correctness of our edit vector computation code by computing all possible transitions for $\tau = 1$ to $\tau = 4$ with our simplified and bitwise versions and comparing them to the original edit vector values. Notice that this simulation is easily implemented by using the EVA computed by BEVA to produce the reference value.

First, we need to pre-process the table of bitmaps \mathcal{H} , for each character in p , we mark the position that this character occurs in the bitmap, setting the $j + \tau$ th-bit to 1 starting on the left, as shown in lines 3 and 4. Second, we need to search by simply iterating over each character in the string s . And for each character in s , a value of τ and the Algorithm 3, calculate the new edit vector from the previous edit vector and a bitmap extracted from table \mathcal{H} correspondent to the current character in s . This search process follows until the last character in s or when it reaches a final edit vector, as shown in lines 7 to 13.

Algorithm 4 Computes edit distance between two strings p and s limited to a maximum number of errors τ .

```

1: procedure computeEditDistance( $p, s, \tau$ )
2:   /* preprocessing */
3:    $\mathcal{H} \leftarrow 0$   $\triangleright \mathcal{H}$  is the same bitmap table of BEVA.
4:   for  $j = 1, 2, \dots, |p|$  do
5:      $\mathcal{H}[p[j]] \leftarrow \mathcal{H}[p[j]] \mid 1 \ll (|p| - j + \tau)$ 
6:   /* searching */
7:    $v \leftarrow v_0$   $\triangleright v_0$  is the initial edit vector
8:   for  $i = 1, 2, \dots, |s|$  do
9:      $b \leftarrow \mathcal{H}[p[i]] \gg (|p| - i)$ 
10:     $b \leftarrow b \ll (w - (2\tau + 1))$ 
11:     $v \leftarrow \text{computeNewEditVector}(v, b, \tau)$ 
12:    if  $v = 0$  then
13:      Break
14:   return  $v[\tau + 1 + (|p| - |s|)]$   $\triangleright$  when
15:    $|p| \in [|s| - \tau, |s| + \tau]$  or more than  $\tau$  otherwise.

```

We also observed that if the pair of strings is large than the length of the computer word, ie. $|p| + \tau > w$ or $|s| + \tau > w$, we need to build the current bitmap of $2\tau + 1$ bits for each character in s , marking the occurrence of the j th-character by setting to 1 the bit in the bitmap starting on the left as follows: $b \leftarrow b \mid 1$ when $p[i] = s[j]$ or $b \leftarrow b \ll 1$ otherwise, $\forall i \leq j \leq \min((2\tau + 1 + i), |s|)$.

4.5 Computational Cost

The difference between our algorithm and EV [Zhou et al., 2016] is the way we compute the new values of the edit vectors given the previous values. Such bit parallel computation does not make sense when the proposed bit parallel edit vector does not fit in a computer word. In such situations, we should just switch to the sequential computation of edit vectors, using the EV method. For instance, when using a 64-bit computer word, the maximum value of τ should be 4, since our algorithm would require $(2\tau + 1)(\tau + 1)$ bits for the edit vector, which gives 45 bits. For τ equal to 5, the algorithm would require 66 bits, so the bit edit vector would not fit into a computer word. Notice that a virtual edit bit vector that aggregates more than one computer word would be possible, but performing bit operations in such a bit edit vector would become expensive and would not be worth it. The restriction for machine words in bit-wise operations is also present in previous works that adopt such strategy [Baeza-Yates and Gonnet, 1992; Silva de Moura et al., 2000; Navarro and Rafinot, 2001; Peltola and Tarhio, 2003; Durian et al., 2009].

Further, we need to start our table of bitmaps \mathcal{H} with zero in all positions, and this takes an extra cost $O(\Sigma)$, being Σ the size of the vocabulary. Given that, the time complexity of our bit parallel approach is Σ plus the time complexity for computing the edit vectors in the EV algorithm, so $O(\Sigma + \tau \cdot \min(m, n))$, which is also close to the cost of EV algorithm. Despite this not-so-good time complexity when compared to the baselines, the proposed algorithm is still fast for important practical scenarios. It takes $O(\tau)$ to update the edit vectors when $b \neq 0$, but when $b = 0$, the edit vector is updated at cost $O(1)$. In practical situations where the chance of finding symbols not present in the prefix query is high, such as a short prefix in a natural language text, our proposal speeds up the query processing for small values of τ . As we will show in the experiments, this property is particularly useful for our main target application, QAC.

The restrictions imposed by the τ limit also extend beyond QAC applications. Any domain that requires high error tolerance would face similar limitations. For example, in bioinformatics or error-prone data entry systems, where higher error bounds may be more common, the efficiency gains of our bit-parallel approach could be minimal, forcing a shift to less efficient sequential methods.

This discussion highlights the importance of aligning the τ limit with the application's error tolerance requirements. Although our algorithm excels in low-error scenarios typical of QAC, its applicability decreases as the error threshold requirement increases. Future research could explore optimizing bit-parallel computations for larger values of τ , potentially through innovative data structures or hardware advances that accommodate larger bit vectors in single or ag-

gregated computer words.

5 Experiments

First, we performed detailed experiments with BWBEV, our bitwise version of BEVA applied to QAC systems as our target application, and in the last section, we performed experiments with BWEV, our bitwise version of EV as a method for online edit distance calculation.

5.1 Setup

The way that algorithms search in a QAC system depends on the model adopted by the system to match a given user prefix query and the complete queries on a dataset. The match modes are chosen in a system design and can employ different methods and data structures [Krishnan *et al.*, 2017; Ferreira *et al.*, 2022]. An approach for matching two distinct strings in a QAC system performs a prefix match between a prefix query and the full queries from the dataset, allowing an approximate match, which is the most popular mode in the QAC system and we adopt it here for almost all the experiments according for our problem definition. Another alternative is to tokenize each string from the dataset and the prefix query already typed by the user into words. Subsequently, the system proceeds to compute an approximate prefix match between each word extracted from the user's entered prefix query and the words contained within the dataset. An additional step is needed to associate the lists of queries where the matched words appear with the prefix search. This match mode is also approached in our experiments.

Besides matching modes, another feature that can affect our experiments is that QAC systems use a standard practice to avoid presenting all matches directly to users. Instead, an intrinsic ranking mechanism is essential to carefully select and showcase the most pertinent results. The ranking can be established by considering diverse attributes, encompassing the occurrence frequencies of suggestions within indexed documents, the frequency of user clicks on suggestions, and other pertinent factors. Here we adopt the Most Popular Completion (MPC) [Bar-Yossef and Kraus, 2011] technique.

The performance of the proposed method was evaluated through experiments carried out on three distinct datasets. While two of these datasets were previously utilized in studies about approximate prefix search methods, it should be noted that they lack query logs and were not derived from an actual query autocompletion service, our main application here, then we describe them as *synthetic datasets*. To provide a comprehensive evaluation, we also incorporated a dataset extracted from an online query autocompletion service.

MEDLINE²: is the main bibliographic database from the US National Library of Medicine (NLM), which contains over 28 million references to articles in health science journals, with an emphasis on biomedicine topics. The title of each article was extracted. Each extracted title corresponds to an item.

DBLP³: This dataset encompasses approximately 4.3 million records of computer science publications. For the purpose of our experiments, we solely focused on the publication titles within the **DBLP** dataset. It is worth noting that **DBLP** has been extensively employed in the experiments conducted by various researchers [Li *et al.*, 2011; Xiao *et al.*, 2013; Qin *et al.*, 2020].

Detailed statistics about the synthetic datasets are presented in Table 8. In each case, any duplicated items within the datasets have been removed.

Dataset	Size (bytes)	Items	Avg Item Len
MEDLINE	2,555,416,200	27,941,081	91.4
DBLP	334,999,905	4,378,548	76.5

Table 8. Overview of dataset statistics utilized in the experiments.

For the experiments with QAC methods, we adopted a procedure introduced in previous work and selected 1,000 suggestions from the collection to serve as the basis for generating the queries. Random errors were added to all queries to create a set of queries for each tested edit distance threshold. Additionally, experiments were conducted using various prefix sizes by extracting the prefixes from these generated queries.

We also have performed experiments with QAC methods using a dataset extracted from a real-world query autocompletion search service. Specifically, we used the query autocompletion suggestion dataset from the Jusbrasil⁴, a Brazilian legal technology company offering a vertical search service. This dataset, which was previously introduced in Ferreira *et al.* [2022], contains 23,374,740 items and a log of 648,264 prefix queries. The dataset includes the prefixes typed by users before clicking to issue the query. Table 9 presents details about the JUSBRASIL dataset. The user queries contain the maximum prefix typed by users when sending queries to the system. For some queries, the user types a prefix and clicks on an option suggested by the site JUSBRASIL, while for others, the user types the entire query.

File	Size (bytes)	Items	Avg Item Len
query suggestions	829,373,817	30,628,391	27.0
prefix queries	15,441,446	751,313	20.5

Table 9. Overview of metrics regarding query suggestions and prefix queries within the JUSBRASIL dataset.

The experiment compares the performance of the following algorithms to QAC methods:

- ICPAN proposed by Li *et al.* [2011] is a trie-based method designed for approximate query autocompletion.
- BEVA [Zhou *et al.*, 2016] is a trie-based algorithm for approximate query autocompletion that reduces the size of the active nodes set by maintaining the boundary active nodes and using the Edit Vector Automaton (EVA) structure to compute the edit distance operations. The

²https://www.nlm.nih.gov/databases/download/pubmed_medline.html

³<https://dblp.uni-trier.de/faq/How+can+I+download+the+whole+dblp+dataset,+dataset+release+dblp-2019-04-01.xml>

⁴<http://www.jusbrasil.com.br>

default automaton used in BEVA for this experiment is EVA⁵.

- BEV adjusts BEVA to not use the EVA structure. In this method, the edit vector is built during the query processing instead of consulting the EVA structure to obtain the next edit vector.
- BWBEV is our proposed method that improves BEVA by computing the edit distance through a bit parallelism approach without the need to maintain the EVA structure.

The experiments were executed on a system featuring an Intel Xeon E5-4617 with a processor of 2.90 GHz, 64 GB of memory RAM, and running the Ubuntu 18.04.1 LTS operation system. The algorithms were coded in C++ and compiled using GCC version 7.4.0. We have tried to include the method IncNGTrie in the experiments, however, its memory requirements did not allow us to run it on our server machines for the datasets included in the experiments.⁶

5.2 Experiments Utilizing Synthetic Datasets

The results obtained from processing queries on the synthetic datasets DBLP and MEDLINE are presented in Tables 10 and 11, respectively. The tables display the outcomes achieved while varying the number of errors and prefix sizes. We report times for prefix sizes 9 and 17. The values reported for each prefix query size represent the cumulative time required to obtain the final results for both datasets. For example, when reporting the time for a prefix size of 17, we report the cumulative time to process prefixes from size 1 to 17. The times are reported with a 99% confidence interval.

Methods	Time (ms)					
	$\tau = 1$		$\tau = 2$		$\tau = 3$	
	9	17	9	17	9	17
BEVA	0.13 ± 0.002	0.14 ± 0.004	1.12 ± 0.018	1.15 ± 0.018	5.56 ± 0.084	5.65 ± 0.088
BEV	0.11 ± 0.002	0.12 ± 0.002	1.40 ± 0.022	1.43 ± 0.023	7.12 ± 0.100	7.22 ± 0.103
BWBEV	0.05 ± 0.001	0.06 ± 0.001	0.55 ± 0.011	0.57 ± 0.012	3.14 ± 0.058	3.20 ± 0.060
ICPAN	0.21 ± 0.004	0.24 ± 0.006	3.11 ± 0.071	3.19 ± 0.074	24.14 ± 0.568	24.53 ± 0.583

Table 10. DBLP - Processing times when using BEVA, BEV, ICPAN, and BWBEV to prefix queries size 9 and 17 and varying τ from 1 to 3.

We have observed that our method has the best query processing time compared to the BEVA, BEV, and ICPAN methods when processing the DBLP and MEDLINE datasets. The advantage is more significant when τ is large. For example, we can achieve up to a 7x speed up against ICPAN, up to a 2x speed up against BEV, and up to a 2x speed up against BEVA in DBLP. Additionally, the confidence intervals of BWBEV are smaller than those of the baselines. This finding is important because it demonstrates minimal variation in

⁵We verified the performance and accuracy of our implementation by comparing its output to the output from the original binary code made available by the authors. The findings indicate that our implementation exhibits superior speed compared to the provided binary code.

⁶The source codes and instructions for the experiments are publicly available on GitHub repository <https://github.com/vdbergg/BWBEV> to allow reproduction of the results.

Methods	Time (ms)					
	$\tau = 1$		$\tau = 2$		$\tau = 3$	
	9	17	9	17	9	17
BEVA	0.22 ± 0.005	0.24 ± 0.005	2.14 ± 0.040	2.20 ± 0.042	9.92 ± 0.167	10.12 ± 0.175
BEV	0.17 ± 0.004	0.19 ± 0.004	2.32 ± 0.043	2.38 ± 0.045	13.43 ± 0.215	13.65 ± 0.224
BWBEV	0.10 ± 0.002	0.11 ± 0.003	1.11 ± 0.026	1.14 ± 0.027	6.10 ± 0.118	6.21 ± 0.122
ICPAN	-	-	-	-	-	-

Table 11. MEDLINE - Processing times when using BEVA, BEV, ICPAN, and BWBEV to prefix queries size 9 and 17 and varying τ from 1 to 3.

the processing times of prefix queries, indicating a consistent and stable time expectation across different queries.

5.3 Comparison to QAC Baselines Methods

In Table 12, we show the comparison of processing times between our proposed method BWBEV, and the baseline methods. The confidence interval adopted is 99%. The query processing times for the JUSBRASIL dataset using our proposed method were almost twice as fast as the times for the BEVA, more than twice the times for BEV, and almost ten times faster than the time for ICPAN when allowing 3 errors. In such cases, BWBEV processed queries in an average time of 5.94 milliseconds, while BEVA, BEV, and ICPAN resulted in a time of 9.34, 12.91, and 57.12 milliseconds, respectively. This indicates that BWBEV was 36.41% faster than BEVA, which was the fastest method among the baseline methods.

Methods	Time (ms)		
	$\tau = 1$	$\tau = 2$	$\tau = 3$
BEVA	0.13 ± 0.001	1.53 ± 0.013	9.34 ± 0.026
BEV	0.12 ± 0.004	1.86 ± 0.048	12.91 ± 0.327
BWBEV	0.07 ± 0.002	0.84 ± 0.026	5.94 ± 0.177
ICPAN	0.31 ± 0.002	5.37 ± 0.067	57.12 ± 0.241

Table 12. JUSBRASIL - Processing times when using BEVA, BEV, ICPAN, and BWBEV and varying τ from 1 to 3.

5.4 Results with Larger Prefix Queries and Number of Errors

We also investigated the behavior of BWBEV when applied to a wider range of prefix sizes and edit distance thresholds. The experimental results, as illustrated in Figure 2, showcase the outcomes obtained by varying the prefix size from 3 to 30. ICPAN was removed from this experiment because of its high processing time, which would impair the visualization of the other results. As expected, the time performance of the methods remained in the same proportion as reported in the previous section for all tested prefix sizes. This finding is particularly important for larger prefixes. These findings also remain consistent when altering the edit distance threshold. In summary, our experiments demonstrate that BWBEV

outperforms BEVA and BEV in terms of processing time for all scenarios tested.

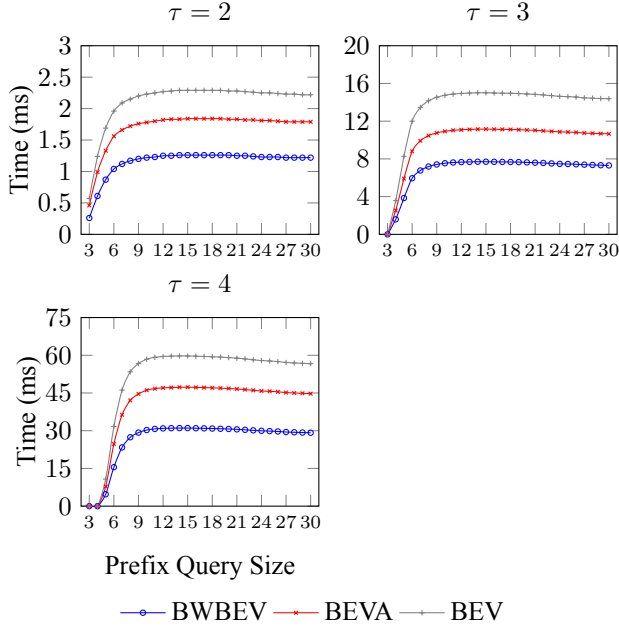


Figure 2. Time performance (in milliseconds) while adjusting the prefix query size and the permissible number of errors within the JUSBRASIL dataset.

5.5 Baseline Comparison on a Term-by-Term

We here consider another potential use case for the BWBEV method which involves performing matches term-by-term or just word-by-word, but with the added capability of allowing approximate matches. In this novel scenario, a query suggestion vocabulary is stored, consisting of all unique words, the matching is performed word by word, and post-processing is performed to select the best query suggestions. We only focus on evaluating the performance of matching, as we are utilizing data structures to carry out the prefix match operations.

The performance results of the compared methods when conducting word prefix matching in the JUSBRASIL dataset are presented in Table 13. The time achieved by BWBEV was 42% better than BEVA when analyzing $\tau = 3$. The time performance of BEV was worse when compared to the times achieved by BEVA, being 22% slower. ICPAN demonstrated significantly inferior time performance compared to both BEVA and BWBEV.

5.6 Scalability

We conducted experiments using Vegeta⁷, a versatile HTTP load-testing tool developed to test HTTP services with a constant request rate.

The target application that we address in our experiments does not require the system to provide all the matches as a result when searching for a prefix. Then only the top- k best-scored results are retrieved. To address the challenge

Methods	Time (ms)		
	$\tau = 1$	$\tau = 2$	$\tau = 3$
BEVA	0.09 ± 0.002	0.95 ± 0.016	5.06 ± 0.101
BEV	0.08 < 0.0001	1.18 ± 0.005	6.51 ± 0.037
BWBEV	0.04 < 0.0001	0.50 ± 0.003	2.92 ± 0.019
ICPAN	0.19 ± 0.001	3.34 ± 0.016	33.31 ± 0.175

Table 13. Processing time (ms) to mode word by word in BEVA, BWBEV, BEV and ICPAN when indexing the JUSBRASIL dataset.

of computing top- k results in the BWBEV and baselines algorithms for approximate prefix search, we have adopted an alternative Bar-Yossef and Kraus [2011] for fast retrieval of the best-scored matches. We adopted the well-known most popular completion (MPC) approach, which is based on the search popularity of queries matching the prefix typed by the user.

Figure 3 presents the results obtained by BEVA, BEV, BWBEV, and Elasticsearch completion methods on the JUSBRASIL dataset for $\tau = 1$, $\tau = 2$ and $\tau = 3$. The reported times encompass the complete server response durations, encompassing communication and other relevant times needed to generate the responses. We have configured Elasticsearch with the setup closest to the results of our system and included this option in the experiments to allow a comparison with a tool that is popularly adopted as a search engine. We used the standard completion sorting function, which utilized the BM25 sorting method, while our method used the sorting function based on the frequency of each suggestion in the log and the number of errors. Both sorting functions retrieved only the top-10 results.

Elasticsearch adopts a structure called the Finite State Transducer (FST), a finite state automaton optimized for prefix matches stored in memory. It also supports typo correction in completion queries using the n-gram-based typo correction technique. The N-gram technique is a text-shaping technique that breaks text into fixed-length strings of characters called n-grams. When indexing completion fields, Elasticsearch splits the text into fixed-length n-grams and stores these n-grams as completion tokens. This technique allows Elasticsearch to find suggestions that match a part of the query, even if the query has typos.

As shown in Figure 3, the servers using BEV and BEVA were the first to exceed the 100-millisecond threshold in $\tau = 1$, while BWBEV and elasticsearch maintained a limit of request per second (RPS) close to it. We remember that the 100 milliseconds threshold is commonly regarded as suitable for autocompletion services. BWBEV supported more than twice the workload of the baselines tested when analyzing $\tau = 2$ and $\tau = 3$. For instance, for $\tau = 3$, BWBEV achieved a processing rate of approximately 760 requests per second, delivering responses under 100 milliseconds, while BEVA and elasticsearch were only able to process less than 600 requests per second, which is approximately a 25% greater ability to process requests.

⁷<https://github.com/tzenart/vegeta>

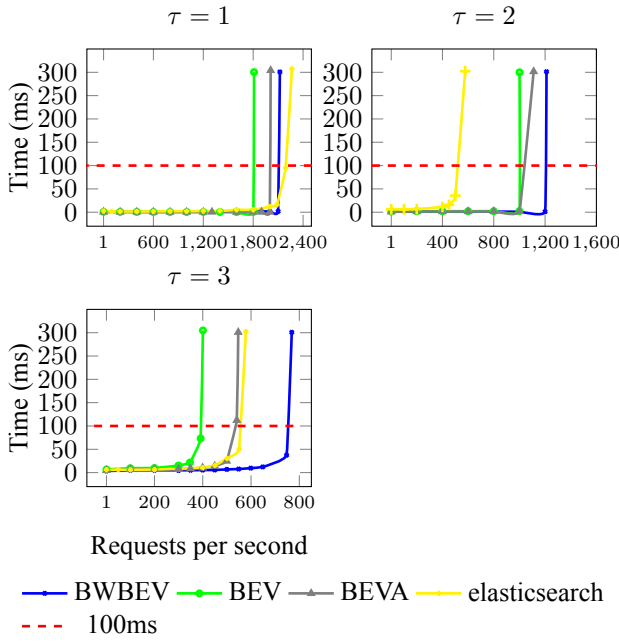


Figure 3. Processing time (in milliseconds) with increasing requests per second while varying τ (ranging from 1 to 3) within the JUSBRASIL dataset.

5.7 Performance as Dataset Size Increases

We conducted additional experiments by varying the size of the indexed base, ranging from 20% to 100% of JUSBRASIL. ICPAN was not included in this analysis due to its significant disparity in time performance, which would make it challenging to compare with the other methods. The behavior of the methods is shown in Figure 4.

Across all portions of the dataset tested, BWBEV consistently outperformed BEVA. As an example, with only 20% of the dataset indexed, BEVA demonstrated a performance 20% slower than BWBEV. This gap expanded gradually as more substantial portions of the dataset were indexed, reaching 25% when the entire dataset was indexed. This expanding difference indicates that BWBEV exhibits superior performance compared to BEVA when indexing larger query suggestion datasets.

5.8 Online Edit Distance Calculation

As a final experiment to evaluate the performance of BWEV - our online version of edit distance calculation, we compare its performance with other edit distance algorithms proposed in the literature. The experiment compares the performance of the following algorithms to edit distance calculation:

- BWEV is our proposed method that uses bitwise operations and edit vectors proposed by Zhou *et al.* [2016] for online edit distance calculation.
- MYERS is a fast method to edit distance calculation that employs bit parallel operations in the diagonal of the dynamic programming matrix.
- HYYROS is another fast method to edit distance calculation that also employs bit parallel operations to calculate the k -diagonals proposed by Ukkonen [1985].
- EV is the computation of edit vectors proposed by Zhou *et al.* [2016], adapted by us here to allow online edit

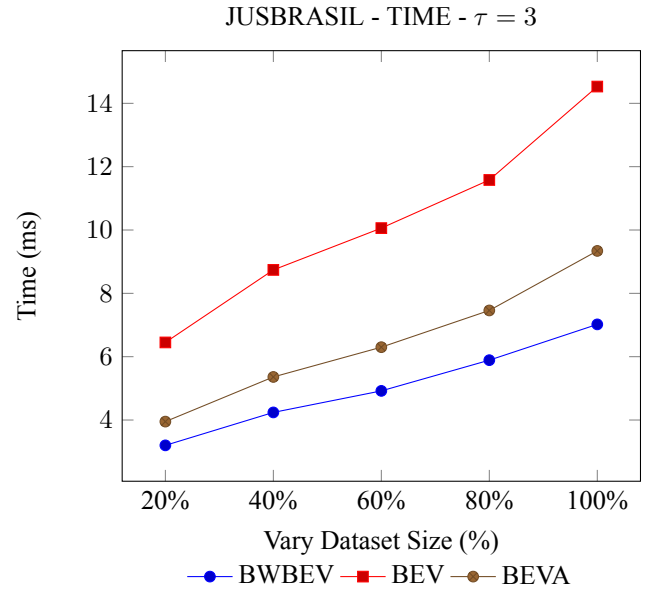


Figure 4. Time performance (ms) of BEVA, BEV, and BWBEV when indexing distinct amounts of JUSBRASIL.

distance calculation.

We randomly extract for each synthetic dataset a total of 16,255 pairs of strings that represent a sample of the dataset with a 99% confidence level and 5% margin of error. With these samples, we create two scenarios to test. The first one we name as *distinct strings set*, is a set randomly taking pairs of strings limited to 5, 50, and 150 characters. The second we name as *same strings adding errors set* is a set with pairs of strings also limited to 5, 50, and 150 characters, but each pair is derived from the same string, with one of the strings being the original form and the second being the string by randomly adding up to 4 errors.

We tested the two synthetic datasets DBLP and MEDLINE for τ varying from 1 to 4 and the pairs of strings with sizes 5, 50, and 150. When searching in the *distinct strings set*, as shown in the tables 14 and 15, Myer's method was faster when searching for prefixes with size 5 and for τ values 3 and 4. Myer's method was better in this scenario because it does not increase the time when the τ values increase, especially to short string sizes. However, for the strings with large sizes of 50 and 150, BWEV was faster for all τ values experimented. This happened because BWEV stops the computation when reaches a final edit vector and this is very common for τ values and large prefix strings that are not so similar to each other. Another factor is that BWEV processing strings have no similarities in their best case using fewer bit parallel operations to the edit vector computation as shown in lines 2 and 3 in the Algorithm 3. When the strings are not similar, BWEV just needs to process the Equation 10 instead of the full Equation 2.

In the *same strings adding errors set*, as shown in the tables 16 and 17, Myer's method was faster for all values of τ and strings sizes tested. Myer's method was better in this scenario because processing very similar strings represents the worst scenario to BWEV due to the need to complete the edit vector computation as shown in lines 5 to 15 in the Algorithm 3 to process the full Equation 2.

In front of these tests, we can conclude that BWEV is also

Methods	Time (ms)											
	m = n = 5				m = n = 50				m = n = 150			
	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$
BWEV	1.80	2.19	2.88	2.96	5.75	5.75	7.27	8.44	14.07	16.29	19.56	31.67
MYERS	2.41	2.43	2.38	2.39	16.75	16.72	16.77	16.84	36.22	36.29	36.26	36.67
HYYROS	2.65	2.61	2.75	2.79	9.24	9.35	9.25	9.14	41.43	41.49	41.56	41.63
EV	16.16	26.38	37.55	45.53	20.88	32.70	45.95	64.08	26.06	40.24	55.16	75.91

Table 14. Query processing of pair of strings no similarity in DBLP dataset.

Methods	Time (ms)											
	m = n = 5				m = n = 50				m = n = 150			
	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$
BWEV	1.77	2.21	2.88	2.98	5.04	5.70	7.19	8.54	11.67	13.98	17.22	28.61
MYERS	2.37	2.39	2.27	2.47	17.08	17.18	17.10	17.28	41.20	41.22	41.10	41.25
HYYROS	2.65	2.60	2.62	2.45	19.77	19.79	19.87	19.57	47.11	47.18	47.19	47.27
EV	16.50	26.77	36.67	45.55	20.39	32.06	45.68	63.09	26.55	40.71	57.21	79.06

Table 15. Query processing of pair of strings no similarity in MEDLINE dataset.

Methods	Time (ms)											
	m = n = 5				m = n = 50				m = n = 150			
	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$
BWEV	2.87	3.45	3.83	4.26	16.86	19.54	22.53	25.12	51.34	57.02	66.07	79.31
MYERS	2.26	2.36	2.21	2.29	14.39	14.29	14.31	14.19	35.60	35.65	35.69	35.50
HYYROS	2.59	2.49	2.68	2.65	16.53	16.50	16.51	16.43	41.12	41.10	41.19	41.12
EV	31.56	35.98	40.40	45.30	215.07	221.67	234.71	254.08	210.34	230.15	253.16	282.72

Table 16. Query processing of pair of strings with similarity between 0 and 4 errors in DBLP dataset.

Methods	Time (ms)											
	m = n = 5				m = n = 50				m = n = 150			
	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$
BWEV	2.90	3.54	3.77	4.12	17.32	19.77	22.58	25.49	55.23	61.32	65.64	79.93
MYERS	2.27	2.29	2.17	2.37	14.42	14.48	14.49	14.52	38.52	38.42	38.57	38.50
HYYROS	2.51	2.58	2.50	2.61	16.67	16.69	16.57	16.70	44.78	44.68	44.79	44.78
EV	31.50	36.02	40.34	44.82	217.34	226.22	241.79	257.50	235.61	254.26	284.24	310.45

Table 17. Query processing of pair of strings with similarity between 0 and 4 errors in MEDLINE dataset.

a good option to be used as a method of online edit distance calculation, but it is good just for scenarios where the number of errors is small and when there are many mismatches in the compared strings, commons scenarios in QAC methods, for example.

6 Conclusion

We have proposed a Bitwise Boundary Edit Vector (BWBEV) method based on a bit-parallelism approach to calculate efficiently the edit distance between two strings. We incorporate BWBEV for a QAC method by improving the ideas presented in the BEVA method, one of the state-of-the-art methods for QAC systems. The experiments presented in this study have demonstrated that the BWBEV method is a new competitive method and as key conclusions, we can present:

- BWBEV outperforms state-of-the-art approximate prefix search methods in all tested scenarios.
- BWBEV is specifically designed for scenarios where the dataset can be indexed for fast prefix search.
- BWBEV can enable the development of faster and more

scalable search systems.

- BWBEV is a competitive method for approximate prefix search on large datasets.

For future work, we plan to study the application of pruning algorithms to BWBEV, combining other alternative ranking methods, including learning-to-rank methods, to accelerate query processing. Also, we plan to study how we can apply BWBEV to practical scenarios of big companies by enabling it to be used in a distributed system. Finally, a possible future direction would be to explore possible adaptation of QAC methods to take advantage of parallel Single Instruction Multiple Data (SIMD) machines, such as GPUs.

Declarations

Funding

This research is partially supported by FAPEAM under the POS-GRAD 2022 Program and the NeuralBond Project (UNIVERSAL 2023 Proc. 01.02.016301.04300/2023-04); by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) financial code 001; and by CNPq under Project IAIA

(406417/2022-9) and individual grants from CNPq to Altigran da Silva (307248/2019-4) and Edleno Moura (310573/2023-8).

Authors' Contributions

Edleno S. de Moura contributed to *Conceptualization, Methodology, Data curation, Formal analysis, Investigation, Software, Writing – original draft, Supervision, and Project administration*. Berg Ferreira collaborated on *Conceptualization, Methodology, Data curation, Formal analysis, Investigation, Software, and Writing – original draft*. Altigran da Silva contributed to *Supervision and Writing – review & editing*, while Ricardo Baeza-Yates participated in *Supervision and Writing – review & editing*. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests or conflicts of interest that could have influenced the results or conclusions of this study.

References

- Alaofi, M., Gallagher, L., McKay, D., Saling, L. L., Sanderson, M., Scholer, F., Spina, D., and White, R. W. (2022). Where do queries come from? In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '22, page 2850–2862, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3477495.3531711.
- Baeza-Yates, R. (1999). Faster approximate string matching. *Algorithmica*, 23:127–158. DOI: 10.1007/PL00009253.
- Baeza-Yates, R. and Gonnet, G. H. (1992). A new approach to text searching. *Commun. ACM*, 35(10):74–82. DOI: 10.1145/135239.135243.
- Bar-Yossef, Z. and Kraus, N. (2011). Context-sensitive query auto-completion. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 107–116, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1963405.1963424.
- Cai, F. and de Rijke, M. (2016). A survey of query auto completion in information retrieval. *Foundations and Trends® in Information Retrieval*, 10(4):273–363. DOI: 10.1561/15000000055.
- Chaudhuri, S. and Kaushik, R. (2009). Extending auto-completion to tolerate errors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 707–718, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1559845.1559919.
- Cucerzan, S. and Brill, E. (2004). Spelling correction as an iterative process that exploits the collective knowledge of web users. In *Conference on Empirical Methods in Natural Language Processing*, volume 4, pages 293–300. <https://aclanthology.org/W04-3238>.
- Deng, D., Li, G., Wen, H., Jagadish, H. V., and Feng, J. (2016). Meta: An efficient matching-based method for error-tolerant autocompletion. *Proc. VLDB Endow.*, 9(10):828–839. DOI: 10.14778/2977797.2977808.
- Durian, B., Holub, J., Peltola, H., and Tarhio, J. (2009). Tuning bndm with q-grams. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, page 29–37, USA. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611972894.3.
- Ferreira, B., de Moura, E. S., and Silva, A. d. (2022). Applying burst-tries for error-tolerant prefix search. *Inf. Retr.*, 25(4):481–518. DOI: 10.1007/s10791-022-09416-9.
- Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499. DOI: 10.1145/367390.367400.
- Heinz, S., Zobel, J., and Williams, H. E. (2002). Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223. DOI: 10.1145/506309.506312.
- Hu, S., Xiao, C., and Ishikawa, Y. (2018). An efficient algorithm for location-aware query autocompletion. *IEICE TRANSACTIONS on Information and Systems*, 101(1):181–192. DOI: 10.1587/transinf.2017EDP7152.
- Hyyrö, H. (2003). A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nord. J. Comput.*, 10(1):29–39. Available at: <https://www.semanticscholar.org/paper/A-Bit-Vector-Algorithm-for-Computing-Levenshtein-Hyyr%C3%B6/813e26d8920d17c2afac6bf5a15c537b067a128a>.
- Ji, S., Li, G., Li, C., and Feng, J. (2009). Efficient interactive fuzzy keyword search. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 371–380, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1526709.1526760.
- Krishnan, U., Moffat, A., and Zobel, J. (2017). A taxonomy of query auto completion modes. In *Proceedings of the 22nd Australasian Document Computing Symposium*, ADCS 2017, pages 2271–2280, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3166072.3166081.
- Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707. Available at: <https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf>.
- Li, G., Ji, S., Li, C., and Feng, J. (2011). Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20:617–640. DOI: 10.1007/s00778-011-0218-x.
- Miller, R. B. (1968). Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1476589.1476628.
- Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415. DOI: 10.1145/316542.316550.
- Navarro, G. (2001). A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88. DOI: 10.1145/375360.375365.
- Navarro, G. and Raffinot, M. (2001). Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. Exp. Algorithmics*, 5. DOI: 10.1145/351827.384246.
- Peltola, H. and Tarhio, J. (2003). Alternative algo-

- rithms for bit-parallel string matching. In Nascimento, M. A., de Moura, E. S., and Oliveira, A. L., editors, *String Processing and Information Retrieval*, pages 80–93, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-39984-1_7.
- Phophalia, A. (2011). A survey on learning to rank (letor) approaches in information retrieval. In *2011 Nirma University International Conference on Engineering*, pages 1–6. DOI: 10.1109/NUiConE.2011.6153228.
- Qin, J., Xiao, C., Hu, S., Zhang, J., Wang, W., Ishikawa, Y., Tsuda, K., and Sadakane, K. (2020). Efficient query autocompletion with edit distance-based error tolerance. *The VLDB Journal*, 29:1–25. DOI: 10.1007/s00778-019-00595-4.
- Silva de Moura, E., Navarro, G., Ziviani, N., and Baeza-Yates, R. (2000). Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139. DOI: 10.1145/348751.348754.
- Ukkonen, E. (1985). Algorithms for approximate string matching. *Information and Control*, 64(1):100–118. International Conference on Foundations of Computation Theory. DOI: [https://doi.org/10.1016/S0019-9958\(85\)80046-2](https://doi.org/10.1016/S0019-9958(85)80046-2).
- Wang, J. and Lin, C. (2020). Fast error-tolerant location-aware query autocompletion. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1998–2001. IEEE. DOI: 10.1109/ICDE48307.2020.00223.
- Wang, P.-W., Kolter, J. Z., Mohan, V., and Dhillon, I. S. (2018). Realtime query completion via deep language models. *CEUR Workshop Proceedings*. Available at: https://www.huan-zhang.com/pdf/sigir_ecom18.pdf.
- Wright, A. H. (1994). Approximate string matching using within-word parallelism. *Softw. Pract. Exper.*, 24(4):337–362. DOI: 10.1002/spe.4380240402.
- Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., and Sadakane, K. (2013). Efficient error-tolerant query auto-completion. *Proc. VLDB Endow.*, 6(6):373–384. DOI: 10.14778/2536336.2536339.
- Zhong, Q., Zhi, J., and Guo, G. (2022). Dynamic is optimal: Effect of three alternative auto-complete on the usability of in-vehicle dialing displays and driver distraction. *Traffic injury prevention*, 23(1):51–56. DOI: 10.1080/15389588.2021.2010052.
- Zhou, X., Qin, J., Xiao, C., Wang, W., Lin, X., and Ishikawa, Y. (2016). Beva: An efficient query processing algorithm for error-tolerant autocompletion. *ACM Trans. Database Syst.*, 41(1). DOI: 10.1145/2877201.