Universidade Federal do Amazonas – UFAM

Instituto de Computação – ICOMP
Programa de Pós-Graduação em Informática – PPGI

Van Den Berg da Gama Ferreira

Implementing Efficient Error-Tolerant Query Autocompletion Systems

Manaus-AM, Brazil

2024, v-1.0

Van Den Berg da Gama Ferreira

Implementing Efficient Error-Tolerant Query Autocompletion Systems

Thesis presented to the Institute of Computing of the Federal University of Amazonas as a requirement for obtaining the PhD in Computer Science.

Universidade Federal do Amazonas – UFAM

Instituto de Computação – ICOMP

Programa de Pós-Graduação em Informática – PPGI

Supervisor: Prof. Dr. Edleno Silva de Moura

Manaus-AM, Brazil

2024, v-1.0

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Ferreira, Van Den Berg da Gama
F383i Implementing Efficient Error-Tolerant Query Autocompletion Systems / Van Den Berg da Gama Ferreira . 2024
107 f.: il. color; 31 cm.

Orientador: Edleno Silva de Moura
Tese (Doutorado em Informática) - Universidade Federal do Amazonas.

1. Error-tolerant. 2. Autocompletion. 3. Trie building. 4. Burst trie. 5. Bit parallelism. I. Moura, Edleno Silva de. II. Universidade Federal do Amazonas III. Título

Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"IMPLEMENTING EFFICIENT ERROR-TOLERANT QUERY AUTOCOMPLETION SYSTEMS"

VAN DEN BERG DA GAMA FERREIRA

Tese de Doutorado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Dr. Edleno Silva de Moura - **Presidente**

Prof. Dr. Altigran Soares da Silva - **Membro Interno**

Profa. Dra. Rosiane Rodrigues de Freitas - **Membro Interno**

Dr. Thierson Couto Rosa - **Membro Externo**

Dr. Leandro Balby Marinho - **Membro Externo**

Manaus, 18 de novembro de 2024.

Documento assinado eletronicamente por **Thierson Couto Rosa**, **Usuário Externo**, em 04/12/2024, às 17:00, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

This thesis is dedicated to my parents, who, despite not having had the opportunity to pursue higher education, always understood its transformative power. Their unwavering belief in the value of learning and their constant support were the foundation that allowed me to walk this path. For their endless encouragement and sacrifices, I am forever grateful.

Acknowledgements

First of all, I want to thank GOD for giving me health, strength, and determination throughout this journey.

MY FAMILY, especially my wife Samile and unconditional companion and my daughter Anallu, for being the fuel that propelled me to continue every day with strength and motivation.

TO MY FATHER and MY MOTHER, who, even though they were a few kilometers away, remained tireless in their expressions of love and affection.

TO MY SUPERVISOR, for the great learning and dedication to help with his brilliant mind, and also for providing me with great challenges allowing me to evolve as a person and researcher. My thanks.

TO THE UFAM GRADUATE PROGRAM, represented by Prof. Dr. Eduardo Luzeiro Feitosa and all the teachers who took part in this journey, teaching great questions of study, research, and extension. My gratitude.

TO ICOMP, for providing all the support I needed to carry out my research.

TO CAPES, this study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work was partially supported by Amazonas State Research Support Foundation - FAPEAM - through the POSGRAD project.

Finally, to all those who contributed in one way or another so that this journey could be completed.

"Nothing is impossible.
If it can be dreamed,
then it can be done.
(Theodore Roosevelt)

Abstract

In this thesis, we focus on developing effective and efficient algorithms and data structures for implementing error-tolerant query autocompletion (ETQAC) systems. An ETQAC system suggests fully ranked queries based on a typed prefix and consists of two main phases: matching and ranking. The matching phase involves selecting query suggestions that match a given prefix, while the ranking phase involves sorting the matched results according to a score function that attempts to select the most relevant suggestions.

We discuss the use of a bit-parallel approach to compute the edit distance between two strings and demonstrate how it can be adapted for approximate prefix search methods. We propose a trie-based method, called *BWBEV*, that uses a unary representation of edit vectors and bitwise operations to update them when computing edit distances. We also show how to apply our new bit-parallelism technique strategy to online edit distance computation between strings without index structure. Our experimental results with BWBEV indicate that it can significantly improve processing speed by more than 36% compared to state-of-the-art methods. In addition, we also study how to optimize the computation of top results when performing the ranking by combining the match and ranking phases to prune results while computing the matches, consequently accelerating the query processing. ETQAC systems usually need to present just a few top-ranked suggestions to their users and we can take advantage of this limit in the number of answers to reduce the computational costs when implementing an ETQAC system.

Regarding methods for computing matching results, several previous studies in the literature have utilized tries and their variations as in-memory data structures to implement the matching phase of ETQAC systems. However, these methods may require a significant amount of memory to process queries. We explore the use of burst tries, a compact version of tries, as the underlying data structure to implement state-of-the-art trie-based error-tolerant prefix search methods. Burst tries are an alternative compact trie implementation that builds lightweight containers in the leaf nodes of the index based on a criterion or parameter to reduce storage costs while maintaining close performance to tries. We examine the trade-off between memory usage and time performance while varying the parameters used to build the burst trie index. For instance, when indexing the JusBrasil dataset, one of the datasets utilized in our experiments, the use of burst tries reduces the memory required by a full trie to 26% and increases time performance to 16%.

Keywords: error-tolerant, autocompletion, trie, burst trie, trie building, bit parallelism, top-k.

Resumo

Nesta tese, desenvolvemos algoritmos e estruturas de dados eficazes e eficientes para sistemas de autocompletar consultas tolerantes a erros (ETQAC). Esses sistemas sugerem consultas classificadas com base em um prefixo digitado, passando por duas fases principais: correspondência e classificação. A fase de correspondência seleciona sugestões que combinam com o prefixo, enquanto a fase de classificação organiza os resultados de acordo com uma função de pontuação que busca as sugestões mais relevantes.

Discutimos o uso de uma abordagem de paralelismo de bits para calcular a distância de edição entre strings, adaptando-a para métodos de busca aproximada por prefixo. Propomos um método baseado em tries chamado *BWBEV*, que utiliza uma representação unária de vetores de edição e operações de bits para atualizá-los ao calcular distâncias de edição. Demonstramos também como aplicar essa técnica para computar distâncias de edição online sem uma estrutura de índice. Nossos experimentos mostram que o BWBEV melhora a velocidade de processamento em mais de 36% em comparação com métodos de ponta.

Além disso, investigamos a otimização do cálculo dos resultados principais, combinando as fases de correspondência e classificação para eliminar resultados irrelevantes durante a correspondência, acelerando assim o processamento. Como ETQACs precisam apresentar apenas algumas das melhores sugestões, essa limitação é explorada para reduzir custos computacionais.

Em relação à fase de correspondência, estudos anteriores utilizaram tries e variações como estruturas em memória. No entanto, esses métodos podem exigir muita memória. Exploramos o uso de burst tries, uma versão compacta de tries, como estrutura subjacente para métodos de busca de prefixo tolerante a erros. Burst tries constroem contêineres leves nos nós folha do índice, reduzindo custos de armazenamento sem comprometer o desempenho. Ao indexar o conjunto de dados JusBrasil, o uso de burst tries reduziu o consumo de memória para 26% de uma trie completa e aumentou o desempenho de tempo em 16%.

Palavras-chaves: tolerância a erros, autocompletion, trie, burst trie, construção de tries, paralelismo de bits, top-k.

Contents

List of Figures

List of Tables

List of Algorithms

List of abbreviations and acronyms

QAC	Query Autocompletion
ETQAC	Error-Tolerant Query Autocompletion
EVA	Edit Vector Automata
BEVA	Boundary Edit Vector Automata
IncNGTrie	Incremental algorithm based on Neighborhood Generation and a Trie index
ICAN	Incremental Computing Active Nodes
ICPAN	Incremental Computing Pivotal Active Nodes
META	Matching-based framework for Error-Tolerant Autocompletion
EAT	Extending Autocompletion To Tolerate Errors

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. To cope with this, 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.

QAC systems suggest full queries based on a typed prefix, which consists of two phases: Matching and Ranking. Matching refers to selecting query suggestions according to the exact or approximate match between a given prefix and the full queries in the suggestions dataset. The ranking phase sorts the matching results according to a score function that attempts to select the top most relevant suggestions for the user. Figure 1 shows how a QAC system works in matching and ranking phases. The user enters a prefix and receives a list of queries that match the typed prefix, ranked according to scores that estimate their relevance.

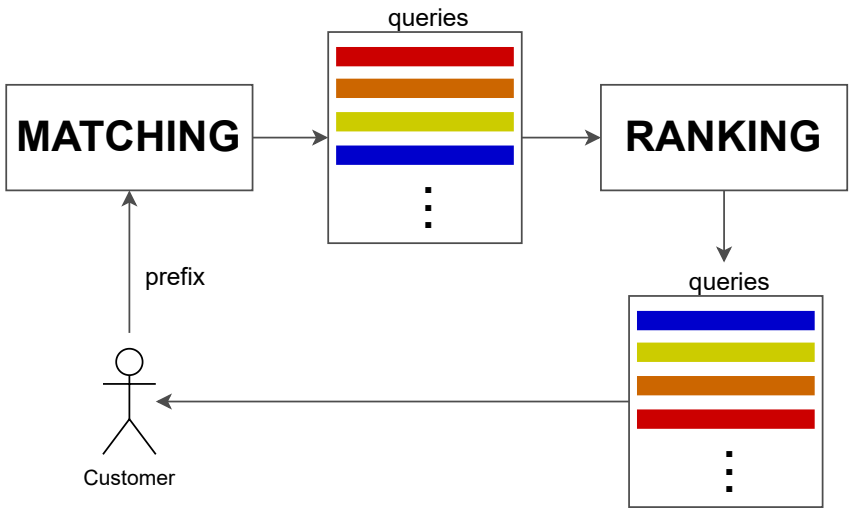
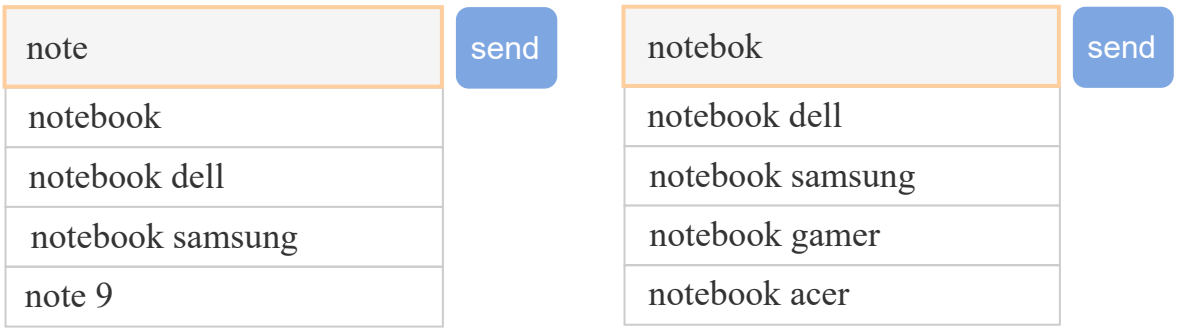


Figure 1 – Example of how a query autocompletion system works.

Figure 2(a) shows an example where the user has typed the prefix query “note”,



(a) Exact prefix search. (b) Error-tolerant prefix search.

Figure 2 – Examples of query autocompletion using (a) exact prefix search and (b) error-tolerant prefix search.

and the system suggests possible queries that match it. In this example, the first top suggestion to the prefix query "note" is the string "notebook". Most likely because it has a higher search frequency in search logs when compared to other suggestions in this ranking. These characteristics or features of suggestions can include search frequency, the position from a match in a sentence, or any other information that allows one suggestion to be highlighted from another. These features can vary in different applications. For example, some QAC systems may use context information from a user to rank suggestions, i.e., the suggestion "note 9" at position 4 could be suggested at the top of the ranking for another user who has previously searched for smartphones.

The design of a high-quality and efficient QAC system is a complex task. First, QAC systems need to be fast because suggestions should be selected and presented as the user types a query. Further, computational costs related to QAC systems may be an important aspect when applied to commercial applications. The quality of results is another challenge since systems need to present effective query autocompletion suggestions to their users. Several decisions may affect the quality of results, including decisions about the source for query suggestions, the choice of features to be adopted when computing the scores of a suggestion given a prefix query already typed by the user, the alternative ways of computing the match between the suggestions and the prefixes and so on. We consider here a scenario where a QAC system needs to be efficient and deliver a good trade-off between the quality of results, query processing time, and memory usage.

When searching in a system that allows query autocompletion, users can submit prefix queries containing typos that might result in unsatisfactory or even in empty query suggestion results in an exact match system. Because of this, recent works have proposed error-tolerant prefix search algorithms for such applications (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., 2019). The error-tolerant approach is a specialized case of approximate match and may help the users to spell difficult queries or to fix typos when the user

is incorrectly spelling the words when writing a query. An example of a search system that allows error-tolerant query autocompletion is shown in Figure 2(b), where the user receives suggestions “notebook dell”, “notebook samsung”, “notebook gamer”, “notebook acer” and “notebook lenovo”, all of them being answers that match with the erroneously typed prefix query “notebok”.

1.1 Matching phase

An important step in the matching phase is the task of choosing the match mode that will be used to perform the match between a given user prefix query and the query suggestions in a dataset. There are many ways to perform the match, and each one may require specific algorithms and data structures. For instance, the match can be performed word by word, comparing words of the query suggestions to the words already typed by the user, or be performed comparing the whole strings of the prefixes and query suggestions. The choice of a specific match mode is a system design and may vary according to the specific properties of each application.

In this thesis, we focus on solving the approximate prefix-matching problem. Let's first define the prefix matching problem: Let Σ be an alphabet. A string s is a sequence of symbols from Σ. We use $|s|$ to denote the length of s, $s[i]$ to denote the i-th symbol of s, starting from 1, and $s[i..j]$ to denote a sub-string of s starting at position i and finishing at position j. Let s and p be two distinct strings composed of the symbols in Σ. We say that p is a prefix of s if $p{=}s[1..|p|]$. When p is a prefix of s, we say that there is an exact prefix match between p and s.

When searching and allowing errors, we need to define a metric to measure the *distance* between two compared strings p and s. 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 required to transform p into s or vice-versa. The number of errors to accept a match becomes a parameter represented by the symbol τ and when the edit distance between p and s is equal to τ, we say that p matches to s with τ errors. If p matches with τ errors to any prefix of a string s, we say there is an *error-tolerant prefix match* with τ errors between p and s.

Given the above concepts, we can now explain the more general prefix-matching problem addressed here as part of the development of an ETQAC system. Let p be a prefix and let $S{=}\{s_1, \ldots, s_n\}$ be a set of strings to be searched. The *error-tolerant prefix search problem* addressed here consists of finding all strings $s_i \in S$ such that there is an *error-tolerant prefix match* between p and s_i with a maximum number of errors τ. In this specific application, we assume that the string keys are previously stored in a large dataset. For instance, one of the datasets adopted in our experiments is the dataset of

suggestions provided by JusBrasil, a Brazilian law tech company whose dataset contains more than 23 million query suggestions.

1.2 Ranking phase

Query autocompletion (QAC) systems do not show all matching results to a user, usually presenting just a small list of the top best-ranked results according to a given score function. The sorting of results and choice of the top list is performed at the ranking phase. The ranking phase takes as input a set $R=\{r_1, \ldots, r_m\}$ of m matching results returned by the matching phase, being each result r_i associated to a tuple of feature values $F_{r_i} = (x_1, .., x_n)$ such that $x_i \in \mathbb{R}$ and n is the number of feature values available. It computes a score $S\colon \mathbb{R}^n \rightarrow \mathbb{R}$ that maps each tuple of feature values to a numerical score. Elements of R are then sorted in a non-increasing order score. The ranking usually requires only the top best k results, the ones with higher scores.

As examples of features and ranking functions found in literature, we can cite Chaudhuri and Kaushik (2009); Qin et al. (2019) that present an approach to select top-k results that uses a static score in the trie node associated with each string from a dataset and combines it with edit distance or similarity between two strings. Another way is to rank candidates according to the string popularity or frequency (Cai et al., 2016; Jiang et al., 2014a; Shokouhi and Radinsky, 2012a), sorting the suggestions in decreasing order of frequency. The frequency feature is also adopted by Jiang et al. (2014a); Shokouhi (2013a). Another alternative to produce score functions is to use Machine Learning techniques to produce ranking functions or just LTR.

1.3 Thesis goals

General

Optimize operations performed in the matching and ranking phases of error-tolerant query autocompletion (ETQAC) systems by studying efficient ways to reduce their query processing time and memory requirements.

Specifics

1. Evaluating the performance of a bit-parallelism approach for error-tolerant prefix search.

2. Adapting the Burst Trie data structure for error-tolerant query autocompletion to reduce storage costs in the matching phase.

3. Exploring different techniques for combining the match and ranking phases to prune results during the matching phase.

4. Investigating alternative methods for efficiently building the index.

1.4 Contributions

Our first contribution in the matching phase is focused on reducing query processing time, we present a new method for approximate prefix search called *BWBEV*, which improves ideas presented in the BEVA (Zhou et al., 2016) method, one of the state-of-art methods from literature. While BEVA uses an automaton referred to as EVA to compute approximate prefix matching, we replace it with a bit-parallel approach. *BWBEV* reduces the time performance up to 36% when compared to the state-of-the-art ETQAC methods. In addition to the speedup resulting from using bit-parallel operations, we discuss in Section 4.3 changes to the edit distance calculation that further speeds up matching when using the bit-parallel approach adopted by *BWBEV* which enable us to create a new method to online edit distance calculation called *BWEV*. *BWBV* reduces time performance of no similarities pair of strings up to 70% when compared to state-of-the-art methods.

BWBEV and the most successful QAC systems in literature adopt indexes based on tries (Fredkin, 1960). 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. Thus, in our second contribution in the matching phase, 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 up to 73% when compared to using full tries, while achieving a similar query processing time performance. Furthermore, the approach can be easily adapted 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 the combination of MCD and MCK as a third heuristic and present an experiment 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 an error-tolerant search, proposing a naive, but effective way of organizing trie nodes in memory when creating the index. The algorithms usually work by inserting one key at a time into the tree, a strategy we call *DFS* index building. Here we experiment with another index-building strategy, the *BFS*, where the nodes are inserted level by level, instead of key by key. It requires the trie keys to be known in advance since it requires the insertion of nodes one level at a time. Also, nodes need to be alphabetically sorted before the insertion. We argue and experimentally show that this strategy, when applicable, can considerably reduce query processing times up to 53%. This gain in performance is achieved because the BFS index-building strategy favors breadth-first search (BFS) in the trie nodes. BFS is adopted by many of the previously proposed trie-based error-tolerant prefix search algorithms (Chaudhuri and Kaushik, 2009; Ji et al., 2009; Li et al., 2011; Deng et al., 2016; Zhou et al., 2016). This performance improvement is achieved without requiring any change in the query processing algorithm. In our experiments using BFS index building and $\tau = 3$ in the JusBrasil dataset, the query processing times were more than twice faster than using the DFS index building strategy.

We also propose and study pruning alternatives to accelerate the approximate prefix search when the application assigns scores to each matching result and requires only the top best-scored matches. The main idea is to turn the matching phase aware of the scores adopted in the ranking phase and use this information to prune matching results while computing the matching, similar to pruning methods used in search systems such as Wand, BMW, and WAVES (Broder et al., 2003; Ding and Suel, 2011; Daoud et al., 2017). We propose two distinct pruning approaches and perform experiments to compare their performance. Our results indicate that pruning methods can significantly reduce query processing times. Thus, the inclusion of studies about how to compute top-k results is important when presenting methods for approximate prefix search on large datasets.

1.5 Overview

Contributions and results can be summarized as follows:

- We proposed and performed an evaluation of the BWBEV method as an alternative to update the set of active nodes of the error-tolerant prefix search methods.
- We present an initial study about alternative pruning methods to produce faster results in ETQAC systems by developing ranking-aware matching algorithms.
- We investigate the impact of building the trie using a BFS index-building strategy as an alternative to the more intuitive DFS index-building strategy for trie node

allocation. While requiring the keys to be sorted, we show that BFS index building, when applicable, may largely reduce query processing times.

- We discuss and evaluate the application of burst tries in error-tolerant prefix search tasks.

The remainder of this thesis is organized as follows. Chapter 2 reviews related work for a QAC system focused on the matching and ranking phases. This chapter also states the problem we tackle, explains related concepts and presents some definitions necessary to understand our proposed ideas and methods. It focuses on trie-based error-tolerant algorithms in the literature, including a brief description of the state-of-the-art methods. Chapter 3 presents our environment of experimentation describing the datasets, baseline methods, baseline structures, and evaluation metrics adopted in the experiments. Chapter 4 presents details about the proposed method BWBEV. We present experiments about the alternative solution to calculate the string similarities. Chapter 5 presents a discussion about practical implementation issues, especially a discussion about BFS and DFS trie-building strategies. We present the experiments applying the optimizations proposed. Chapter 6 presents our discussion about how to use burst tries as indexes to perform error-tolerant prefix searches. We present experiments to alternative burst tries implementation studied here and a comparison of their performance with representative baseline data structures, verifying the impact of using our ideas in a practical scenario adopting a real dataset extracted from an online search service. Chapter 7 presents our conclusions and possible future research directions.

2 Background and Related Work

In this chapter, we explore the necessary concepts to understand our contributions and related work about string matching problem applied to query autocompletion (QAC) systems defined previously in Section 1.1. Exploring the matching and ranking phases of ETQAC methods.

2.1 Query autocompletion matching phase

2.1.1 Match modes

Query autocompletion (QAC) systems can perform different matches between a given user prefix query and the complete queries on a dataset according to its application context. There are a variety of alternative ways to define the semantics of a "match" when performing the task of QAC. These match modes are chosen in a system design and can use different methods and data structures. Krishnan et al. (2017b) identified and documented five query autocompletion match modes as follows:

Mode 1 - This is the most basic mode that performs an exact match between two distinct strings.

Mode 2 - This is probably the most common approach for matching two distinct strings in a QAC system, this mode performs a prefix match between a prefix query and the full queries from the dataset.

Mode 3 - This is mode 2 applied to strings with multi-term or split by words. In this mode, there is at least one word in the prefix query that matches the string in the dataset. The inverted list of each word in the dataset that matches the prefix query must be merged.

Mode 4 - This mode performs a standard sub-string match over each word of one given prefix query multi-term and a string in the dataset. It is multi-word matching similar to Mode 3, but instead of prefix matching, a sub-string search is carried out over the words of the prefix query. The sub-string matching occurs when a given prefix query $p{=}s[i...i+|p|]$, being $i \geq 1$ and s the string in the dataset.

Mode 5 - This is mode 4 adding a specified edit distance or Hamming distance of the prefix query.

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. (2017b)

describe that mode 3 allows finding a match between the query “gam rone” and the suggestion “game of thrones”. At 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 evidence shows 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 2 and 3 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 3, 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 3 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 a query autocompletion system that adopts mode 3.

We plan to study these match modes in more detail as future research, aiming to make comparisons of the quality of results and performance between the different match modes. In this work, we adopted match modes 2 and 3 adding the feature of allowing error. Notice, that match mode 1 is contained in match mode 2.

2.1.2 Efficient index-based structures

2.1.2.1 Tries

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$”, “autonomy$”, “auto_off$”, “book$”, “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 Figure 3. 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 3, 6, 9, 13, 21, and 28 in Figure 3.

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

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

Table 1 – Sample dataset.

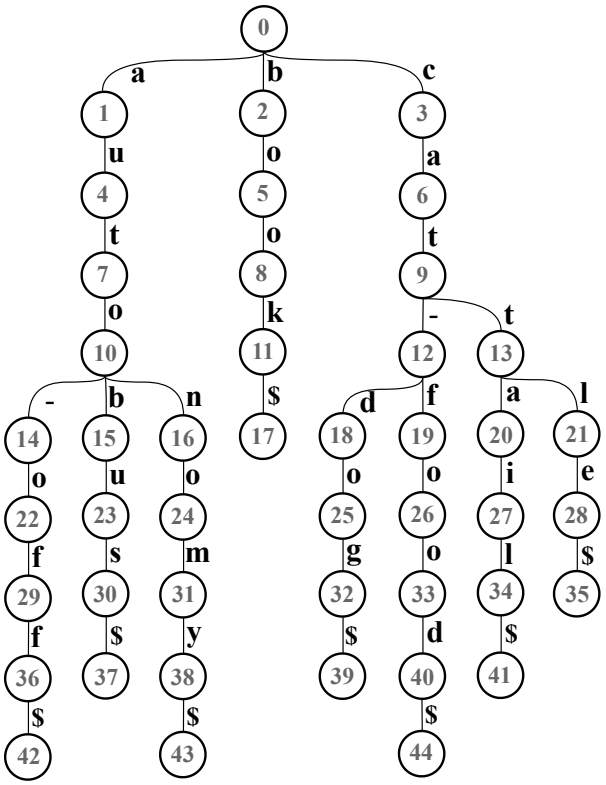


Figure 3 – A trie containing the 8 strings of our sample dataset.

The operation of insertion of a new string s in a trie, starts with a search operation to find the maximum path that already matches the inserted string in the trie. This search makes the root as the current node, which is pointed by *curr*, and the current position *pos* in the inserted pattern as 1, the first character of the string. It then repeats the following procedure: searches for a child of *curr* that contains the key value equal $s[pos]$. When finding this child, making *curr* point to it and increasing *pos* by 1. When not found, a new child node of *curr* is created with the value of $s[pos]$ as its key, making *curr* pointing to this new child and increasing *pos* by 1. We repeat the process until reaching the end of the string being inserted. Each string inserted in the trie should end with a string terminator symbol so that the last node inserted marks the end of a word. For strings that are already present in the tries, no new nodes are created by the insertion process.

Analogous to insertion, the search for a pattern string into a trie follows the proce-

dure described above, except that the search returns a fail when not finding a child node equal to $s[pos]$. Thus the search stops and returns a fail, instead of creating new nodes. In case of success until the terminator symbol of the string, the search indicates that the key was found. If considering a linear search on the children nodes, in both search and insertion the cost is $O(|\Sigma| \cdot |s|)$, where $|\Sigma|$ is the size of the alphabet and $|s|$ is the size of the searching string key. Notice that the cost does not depend on the size of the number of strings inserted in the trie, which makes the trie a very attractive data structure for indexing strings.

2.1.2.2 Burst tries

Trie is a fast data structure and represents a good alternative for building 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 consist of three distinct components, a set of records, a set of containers, and an access trie.

Records: A record contains a string. Each string is unique.

Containers: A container is a small set of records, maintained as a simple data structure such as a list or a binary search tree (BST). Each container also stores a header, for saving the statistics used by heuristics for bursting. The use of a BST as a container enables the retrieval of records in sort order. An in-order traversal of the burst trie starts at the root. Although the records within the container store only suffixes of the original strings, they can be reconstructed by keeping track of the traversal path. However, even if an unordered structure such as a list were used to represent containers, the containers themselves would be in sorted order and their small size, by design, means they can be sorted quickly.

Access trie: An access trie is a trie whose leaves are containers.

Searching involves using the initial characters of a query string to identify a particular container, and then using the remainder of the query string to find a record in the container. Heinz et al. (2002) experimented with alternative data structures to store information on each container and reported that using a binary search tree was a competitive alternative.

Bursting is the process of replacing a container at depth k by a trie node and a set of new containers at depth $k + 1$, which between them contains all the records in the original container. The burst process is based on some heuristics that determine when a container must start the process of bursting. Heinz et al. (2002) studied three heuristics described below.

ratio: The first heuristic, which requires two counters for each container. The counters keep track of two values: the number of times a container has been searched and the number of searches that have ended successfully at the root node of the container, referred to as *direct hit*. A drawback of Ratio is the additional memory required to maintain two counters per container, and the number of tests required at each access.

limit: The second heuristic, which fixed several records, aims to eliminate large containers and limit total search costs. Here we apply this heuristic in our studies and named it as maximum container keys (MCK).

trend: The third heuristic, which also uses one counter per container. Whenever a container is created it is allocated a set amount of capital C. The current capital is modified on each access. On a direct hit, the capital is incremented by a bonus B. If a record is accessed that is already in the container but is not a direct hit, the capital is decremented by a penalty of M. When the capital is exhausted, the container bursts.

Figure 4 shows a trie representation with the strings “life”, “live”, and “love” converted to a burst trie representation. The criteria used is the limit heuristic, where the number maximum of keys is set to 2 in the container.

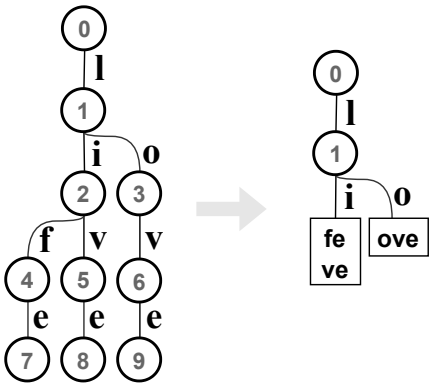


Figure 4 – Burst trie representation generated from a trie. Trie in left and burst trie in right.

The authors of the burst trie did not explore the idea of performing an approximate search. In this thesis, we adapt the burst trie for query autocompletion and present experiments comparing error-tolerant prefix search implementations with burst tries.

2.1.2.3 Compact prefix trees

McCreight (1976) introduced the compact versions of a trie that we named here as *compact prefix trees* (CPT), and that are also known as *prefix trees* 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 that 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. Figure 5 shows a trie representation with the strings "life", "live", and "love" converted to a CPT representation. In this thesis, we present experiments comparing implementations of error-tolerant prefix search with compact prefix trees.

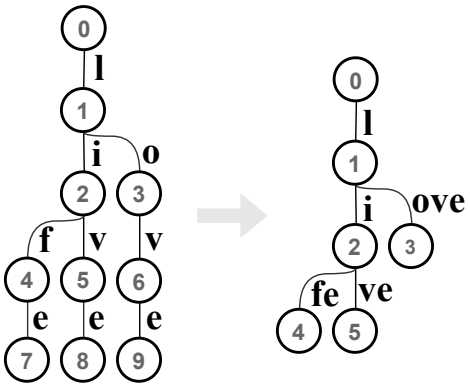


Figure 5 – CPT representation. Trie is on the left and CPT is on the right.

2.1.2.4 Suffix trees and suffix arrays

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 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 for 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 *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). In this thesis, we present experiments comparing implementations of error-tolerant prefix search with suffix arrays.

2.1.2.5 Other tries variations

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.

Askitis and Sinha (2007) introduce the *HAT-trie*, a cache-conscious burst trie implementation that uses the hash as containers in a burst trie. Askitis and Zobel (2011) explore two alternatives to the standard representation of strings when building burst tries: including the string in its node, and, for linked lists, replacing each list of nodes with a contiguous array of characters. They present experiments showing that the changes resulted not only in a reduction in memory usage but also significantly improved search time.

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 to reduce storage costs, at the price of increasing the computational cost for search in the data structure when compared to a conventional trie.

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 them 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 access to trie nodes.

The *Marisa trie* proposed by Yata (2011)[2], is a static trie that consists of recursively compressed Patricia tries stored in the level-order unary degree sequence (LOUDS) representation. It recursively encodes edge labels in a Patricia trie using another Patricia trie. Yata's implementation of the structure is public and supports prefix searches, but

[2] https://code.google.com/archive/p/marisa-trie/

does not support error-tolerant prefix search.

Besides the compact representation, other efficient implementations of tries are discussed in several contexts of applications in the literature, including name lookup in 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 also proposed compact and efficient trie variations, but none of them addressed 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 as 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 an error-tolerant prefix search.

2.1.3 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 Leveinstein 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:

$$\begin{aligned} M[i,j] = min(&M[i-1,j-1] + \delta(p[j], s[i]),\\ &M[i-1,j] + 1,\\ &M[i,j-1] + 1), \end{aligned} \tag{2.1}$$

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 2 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 2 – When calculating the edit distance between "ant" and "auto" the dynamic programming matrix is utilized.

Ukkonen and Wood (1993) 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 3 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 3 – k-diagonal definition for strings "ant" and "auto" and $\tau = 1$.

2.1.3.1 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 4. 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 $[\underbrace{\tau, \tau - 1, \ldots, 1}, 0, \underbrace{1, 2, \ldots, \tau}]$, as the word in column 0 is empty. This characteristic is labeled as the *initial edit vector* and repre-

sented by V_0. Correspondingly, the vector containing all values of $\tau + 1$, represented as $[\underbrace{\tau + 1, \tau + 1, \ldots, \tau + 1}_{2\tau+1}]$, is labeled as *final edit vector* and represented by $V_\perp$.

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

Table 4 – 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:

$$\begin{aligned} v_{j+1}[i] = min(&v_j[i] + \delta(p[j+1], s[j-\tau+i]), \\ &v_j[i+1] + 1, \\ &v_{j+1}[i-1] + 1), \forall 1 \leq i \leq 2\tau + 1. \end{aligned} \tag{2.2}$$

For instance, let us consider the strings $p =$ "pet" and $s =$ "plan" in Table 4 for $\tau = 1$. Then, the calculation of the new edit vector V_1 from V_0 follows:

$$v_1[1] = min(1 + \delta(p, \epsilon), 0 + 1, \tau + 1) = 1 \tag{2.3}$$

$$v_1[2] = min(0 + \delta(p, p), 1 + 1, 1 + 1) = 0 \tag{2.4}$$

$$v_1[3] = min(1 + \delta(p, l), \tau + 1, 0 + 1) = 1 \tag{2.5}$$

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 Section 2.1.5. 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.

2.1.4 Error tolerant prefix search in tries

The central idea of performing prefix search in tries is to maintain a list of active trie nodes to a string p, being p the prefix query already typed by the user. An *active node* represents a match at each step of the prefix search, given a maximum edit distance threshold (maximum number of errors) represented by the symbol τ. An active node is formally characterized by the edit distance between p and s' that is within τ, ie, $ed(p, s') \leq \tau$, where s' represents a prefix of a possible suggestion s of the set of sentences $\mathcal{S}$. The active node set is formally defined as: $\mathcal{A} = \{s'_i \mid s'_i = s_i[1..|p|] \wedge ed(p, s'_i) \leq \tau\}$.

When $ed(p, s') = 0$ we have an exact prefix search. When $ed(p, s') > 0$ we have an error-tolerant prefix search. Notice that the exact prefix search is contained in the error-tolerant prefix search method. The error-tolerant prefix search task must also be able to efficiently process the subsequent prefix query p', where p' is p with a new character appended. In this case, the list of active nodes of p can be used to calculate the new list of active nodes of p'.

In Figure 6, we show the naive steps to computing the active nodes set as the prefix query changes. Consider the prefix query p = "love", $\tau = 1$ and the dataset of strings[3] "life", "live" and "love" indexed in the trie. Initially, we have $p = \epsilon$, where ϵ represent the empty string, we consider the nodes 0 and 1 as active nodes because $ed(\epsilon, \epsilon) = 0$ (dashed circle) and $ed(l, \epsilon) = 1$ (bold circle). When p changes to "l", we have to compute the new active nodes from the active nodes in the previous prefix query. So we need to analyze the nodes 0 and 1 again and also their children. The stopping criterion is when we find a prefix that is outside the edit distance threshold. So, we get the nodes 0, 1, 2, and 3 as active nodes because $ed(\epsilon, l) = 1$ (bold circle), $ed(l, l) = 0$ (dashed circle), $ed(li, l) = 1$ and $ed(lo, l) = 1$ (both with bold circles), the other nodes have an edit distance greater than our edit distance threshold 1 and therefore are not part of the active nodes set for p. When p changes to "lo" the root node is no longer an active node for this prefix query because $ed(\epsilon, lo) = 2$. The rest of the active nodes from the previous prefix query remain as active nodes for p and their edit distances are updated. In addition, node 6 becomes an active node. This algorithm is repeated for the remainder of p, until reaching p = "love", in which the active nodes set is 6, 8, and 9, and the result is represented by the strings "live" and "love". Detailed character-by-character steps are shown in Table 5. Notice that the prefix "lov" is not a result because this prefix does not belong to the dataset of strings and thus the active node 6 associated with this prefix is not a leaf node marked with a terminator symbol.

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

[3] In some sections in this chapter we adopt temporarily a different and small dataset of the presented in Table 1 to facilitate the explanation of specifics concepts.

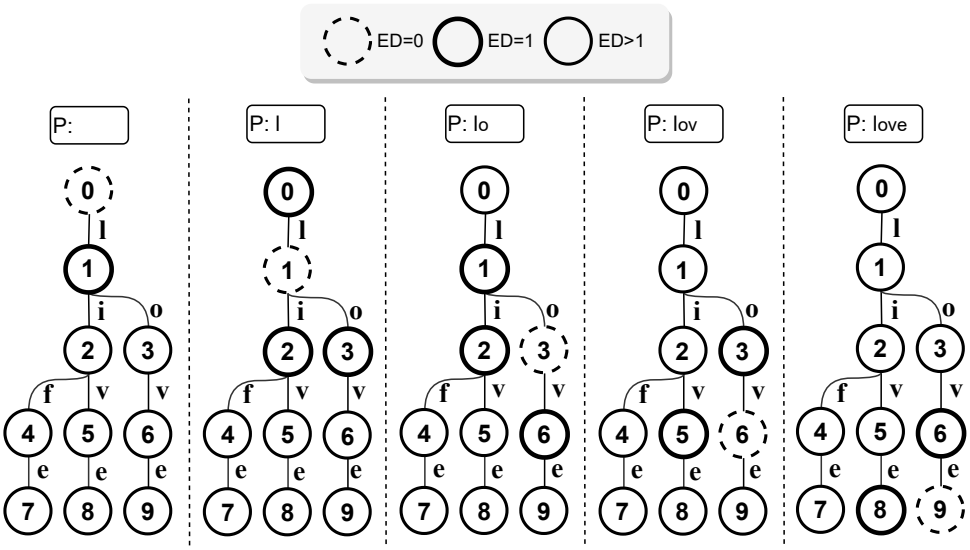


Figure 6 – Computing the active nodes for p = "love" and $\tau = 1$. The strings "live" and "love" are similar to p.

p	Active nodes set
ϵ	{0, 1}
l	{0, 1, 2, 3}
o	{1, 2, 3, 4}
v	{3, 5, 6}
e	{6, 8, 9}

Table 5 – Computing the active nodes set character by character for p = "love", $\tau = 1$ and dataset strings "life", "live" and "love".

operation in the search process when there are large numbers of matches or active nodes to the prefix query.

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.

When looking to the literature about error-tolerant prefix search methods applied to query autocompletion, several approaches (Chaudhuri and 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 in the strategy to maintain the set of active nodes. In the next section, we explain in more detail the main error-tolerant prefix search baseline methods.

2.1.5 Baseline methods

Query autocompletion has been frequently studied in the 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 3, 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.

The main baselines methods about error-tolerant prefix search methods applied to query autocompletion (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., 2019) are described in more detail below.

EAT and ICAN

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 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. An example of these methods is shown in detail in Section 2.1.4 because the approach adopted by them is the most basic.

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.

ICPAN

A drawback of the ICAN algorithm is that maintaining the set of active nodes can be very costly for large amounts of data, especially at the beginning of processing, when the word is short and needs to activate many nodes. This affects processing time and requires more memory consumption.

The subsequent research on 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, called *Pivotal active nodes*. This method has two advantages: (1) Reduces the space to store active nodes, and (2) Improves search performance since they do not need to scan all the active nodes for incremental computation.

The definition of the pivotal active nodes is given in Li et al. (2011) as follows. Given a query keyword p, a trie node n is a pivotal active node of p with respect to an edit-distance threshold τ, if and only if (1) n is active node of p and (2) there exists a transformation from p to n with $ed(n, p)$ edit operations, and the operation on the last character of n is neither deletion $ed(n, p) \neq ed(n', p) + 1$ nor substitution $ed(n, p) \neq ed(n', p') + 1$, where n' and p' are respectively the prefixes of n and p which do not contain the last character.

The intuition of this approach can be illustrated as follows. The example in Li et al. (2011) consider a query keyword "nl" with an edit distance threshold $\tau = 2$, and active node set $\mathcal{A} = \{\langle n_{12}, 1\rangle; \langle n_0, 0\rangle; \langle n_{13}, 2\rangle; \langle n_{19}, 2\rangle\}$ shown in Figure 7, where the tuple $\langle n_{13}, 2\rangle$, for example, is the number of the node in the trie and the current edit distance, respectively. Although n_{13}("li") and n_{19}("lu") are active nodes, we do not need to keep them, since we can use the active node n_{12}("l") to compute the similar words of "li" and "lu" using "l". In other words, we only need to keep the active node "l" to compute the same set of similar words for the query keyword. Seen example character by character in Figure 7.

META

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.

Previous presented prefix search methods (Ji et al., 2009; Li et al., 2011) focus on the threshold-based error-tolerant autocompletion problem, which, given a threshold τ, finds all the strings that have a prefix whose edit distance to the query is within the

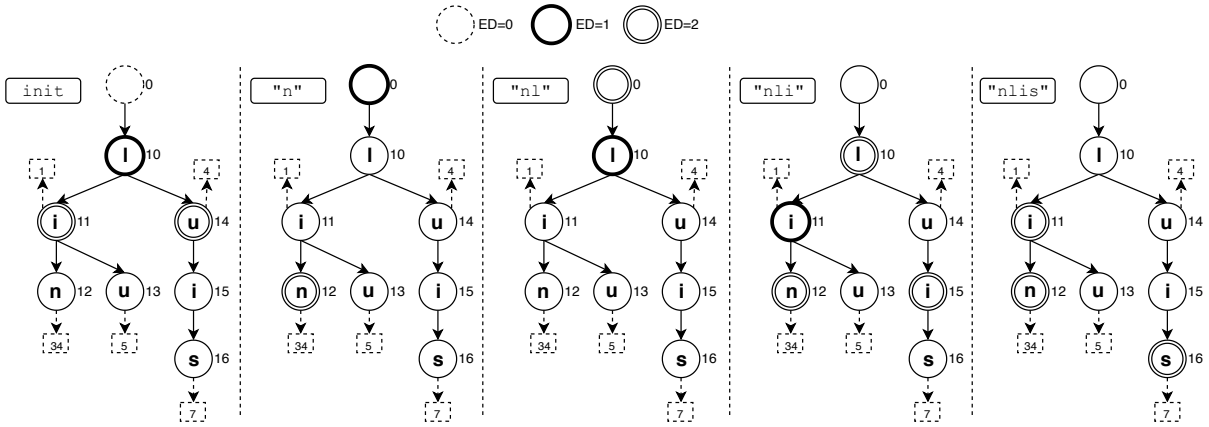


Figure 7 – Fuzzy search of prefix queries of "nlis" (threshold $\tau = 2$) ICPAN.

threshold τ.

Deng et al. (2016) showed that these methods have three limitations. First, they cannot meet the high-performance requirement for large datasets. For example, they take more than 1 second per query on a dataset with 4 million strings. Second, they involve redundant computations to compute the active nodes. Third, it is rather hard to set an appropriate threshold, because a large threshold returns many results while a small threshold leads to few or even no results. For example, the query "parefurnailia" and its top match for human observer "paraphernalia" has an edit distance of 5 which is too large for short words and common errors.

An alternative is to return top-k strings that are most similar to the query. However, existing methods cannot directly and efficiently support top-k error-tolerant autocompletion queries. This is because the active nodes set is dependent on the threshold, and once the threshold changes they need to calculate the active nodes from scratch.

Deng et al. (2016) proposed a matching-based framework for error-tolerant autocompletion, called META, which computes the answers based on matching characters between queries and data. META can efficiently support threshold-based and top-k queries. To avoid redundant computations, Deng et al. (2016) designed a compact trie structure, which maintains the ancestor-descendant relationship between the active nodes and can guarantee that each trie node is accessed at most once by the active nodes.

IncNGTrie

Previous approaches index data in a trie, and continuously maintain all the prefixes of data string whose edit distance from the prefix query is within the threshold. The major inherent problem is that the number of such prefixes is huge for the first few characters of the query and is exponential in the alphabet size. This results in slow query response even if the entire query approximately matches only a few prefixes. Xiao et al. (2013) proposed a novel neighborhood generation-based algorithm called IncNGTrie, which can achieve up to two orders of magnitude speedup over existing methods for the error-tolerant query

autocompletion problem.

Unlike previous algorithms, IncNGTrie 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. For example, consider a dataset of sentences S={“test”, “text”} and $\tau = 1$. Figure 8 shows the trie constructed using the IncNGTrie algorithm. Each path in the trie represents a deletion-marked variant of a data string.

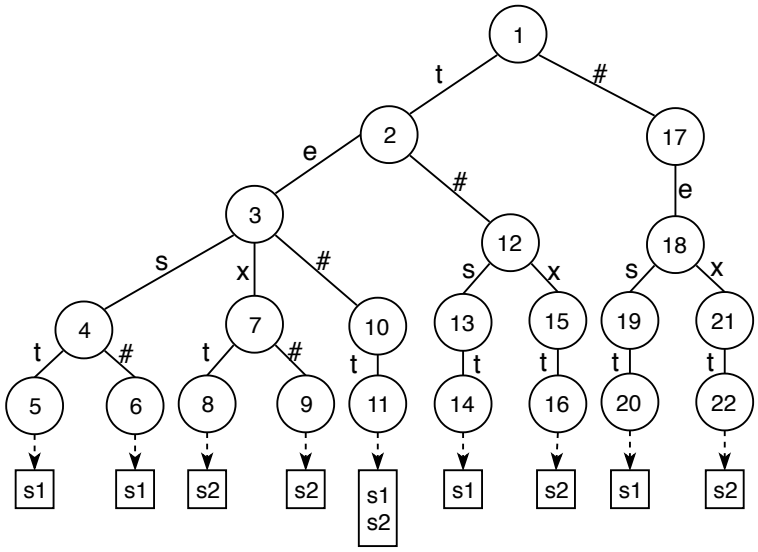


Figure 8 – Example of IncNGTrie to strings s_1=“test” and s_2=“text”

Although the algorithm IncNGTrie shows in its experiments to be efficient in the processing time of queries, the algorithm needs to index several nodes for a single word, so the amount of memory used is higher than other algorithms in the literature. Although the authors propose a reduction in the number of nodes indexed by eliminating duplicate nodes, the algorithm still uses a large amount of memory. Thus, the index size represents a severe restriction to the use of IncNGTrie. Qin et al. (2019) improved the method to reduce the index size produced by IncNGTrie. They also studied the usage of their method to solve the problem of duplicate removal. While their method still requires much more memory than BEVA and META, their new proposal reduces the memory requirements of IncNGTrie, at the price of increasing the time for indexing the databases.

BEVA

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

Chaudhuri and Kaushik (2009) and Ji et al. (2009) showed that all prefixes that satisfy the edit distance constraint are kept as active nodes. Li et al. (2011) maintain a subset of these active nodes, achieving better efficiency both in terms of space and time complexities. It is natural to ask what is the smallest set of prefixes that are within the edit distance threshold that an algorithm must maintain for the error-tolerant autocompletion problem. Zhou et al. (2016) propose the **boundary active nodes set**, which is the smallest set that retrieves all responses efficiently and correctly. The *boundary active node* satisfies the edit distance constraint with the current prefix query, and none of its prefixes (or ancestors in the trie) satisfies the edit distance constraint. The boundary active prefix set is defined formally as: $\mathcal{B} = \{a \mid a \in \mathcal{A} \wedge (\nexists a' \in \mathcal{A}\}$, where a' is the parent node of a.

In Figure 9 we show how the active nodes and boundary active nodes are obtained for the prefix query $p =$ "love" and $\tau = 1$, in a trie data structure with the strings "life", "live" and "love" indexed. Initially, we have $p = \epsilon$ and the nodes 0 and 1 are active nodes and the node 0 is a boundary active node. When changes to $p =$ "l", the nodes 0, 1, 2, and 3 are the nodes that have the edit distance for the prefix query within the edit distance threshold, that is, $ed(\epsilon, l) = 1$, $ed(l, l) = 0$, $ed(l, li) = 1$ and $ed(l, lo) = 1$, therefore are considered active nodes. The root node 0 continues to be the only node that is within the set of boundary active nodes because it is part of the active nodes and does not have any active node that has a prefix smaller than it. When we have $p =$ "lo", the set of active nodes and boundary active nodes are obtained from the active nodes of the previous query. Thus, the new active nodes are nodes 1, 2, 3, and 6. And the set of boundary active nodes is the active node that has the lowest prefix among the other active nodes, that is, there is no ancestor to this node that is an active node and therefore we only have node 1. And so on, until reaching the end of the prefix query.

An important note is that a boundary active node cannot be a boundary active node of the following query and therefore the algorithm in BEVA analyzes only the children of the set of boundary active nodes from the previous query, unlike the maintenance of active nodes (used by Chaudhuri and Kaushik (2009); Ji et al. (2009)) that keeps all prefixes that are within the distance threshold. The root problem that causes much overhead in these solutions is due to their definition of active nodes, which inherently allows ancestor-descendant relationships among active nodes. But, the essential reason for keeping such redundancy in these methods is to ensure that edit distance information can be easily and correctly passed on to the descendant node.

For example, in Figure 10 we have the computation of the active nodes (in green) and the computation of the boundary active nodes (in blue) for the prefix queries "lo" and "lov", $\tau = 1$ and the prefixes indexed in the trie are "life", "live" and " love". As we explained earlier, the active nodes for the prefix query "lo" are nodes 1, 2, 3, and 6. Analyzing the boundary active nodes in this same step we have only the boundary

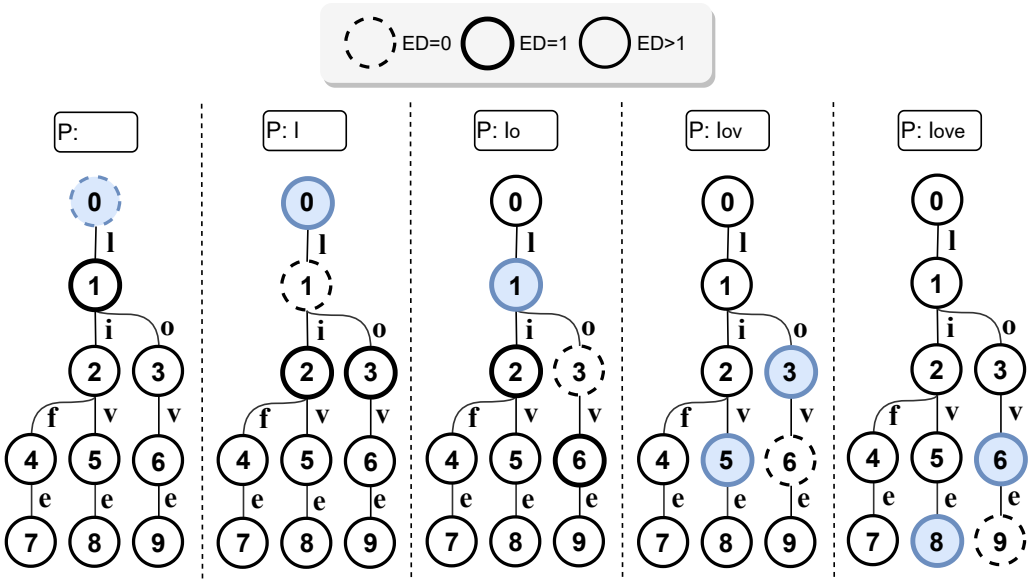


Figure 9 – Representation of boundary active nodes (in blue) obtained character by character for the prefix query "love" and $\tau = 1$.

active node 1, without having to save the other nodes as boundary active nodes because it may cause duplicate results when taking the complete suggestions and the necessity of applying a deduplication in the results. In the computation of the active node, if we do not keep node 2 for the next prefix query "lov" we will not be able to obtain the active node 5, despite this must be part of the result, because when analyzing the child nodes of the node 1 we obtain $ed(lov, li) = 2$, which is outside our edit distance threshold and then the processing ends without reach all the answers correctly. For this reason, the previous methods need to keep all active nodes.

To ensure that all the query results can be computed correctly the key idea in BEVA is to keep for each node all its edit distance values between its $(-\tau)$-and τ-diagonals. Zhou et al. (2016) formalize this idea as edit vectors and show that it can be encoded as a state in a data structure named *edit vector automaton*. This also allows us to maintain only the boundary active nodes set.

The algorithm starts with the root node as being a boundary 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 is typed. The current list of boundary active nodes becomes inactive whenever a new character is added to the prefix query p. A scan of 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.

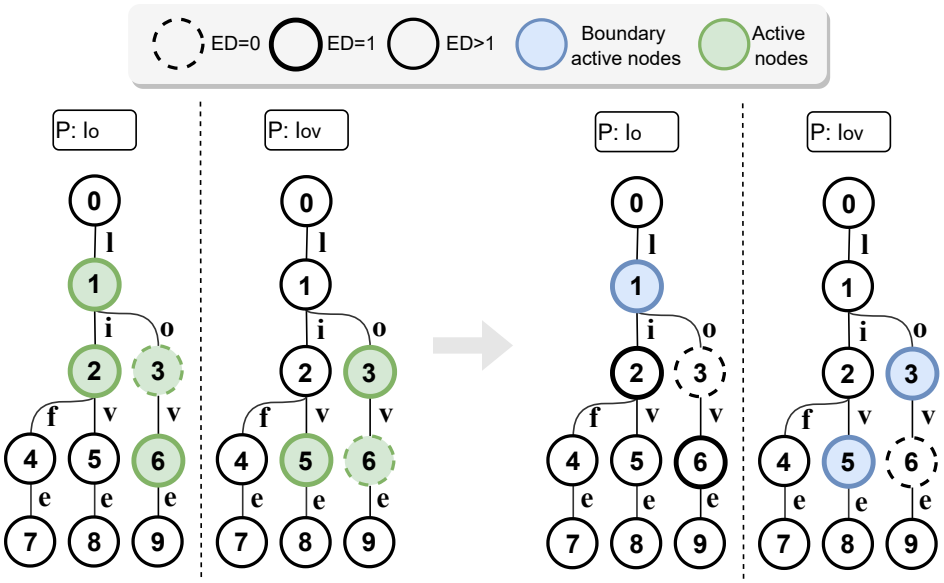


Figure 10 – Difference between obtaining active nodes and boundary active nodes when changing the prefix query from "lo" to "lov" and $\tau = 1$. On the left side, we have the active nodes in green, and on the right side the boundary active nodes in the same step.

- *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 the 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 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 boundary active nodes, and checks for descendant nodes that match the prefix when adding this new symbol, creating a new list of boundary active nodes with them. The list of current boundary active nodes is then replaced by the new list found. After processing all the symbols from the prefix searched, the final list of boundary active nodes is then used to fetch the strings from the dataset

that match the query.

To better illustrate how BEVA traverses a trie, consider a search in our sample dataset using the trie presented in Figure 3. Consider a search task allowing 1 error and the prefix query "cut". The set of boundary active nodes achieved when applying BEVA is presented in Table 6. When starting the search, the root node becomes active for the first symbol 'c', since we allow 1 error in our example. When processing the letter 'u', the algorithm takes the list of current boundary active nodes, only node 0 of Figure 3, 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 the 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 6.

Prefix query	Boundary active nodes set
ϵ	{0}
c	{0}
cu	{3, 4}
cut	{7, 9}

Table 6 – Query processing in BEVA method to prefix query "cut" in our sample dataset.

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 researchers in literature have proposed a 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.

2.1.5.1 Summary of Baseline Methods

To better understand the differences among the baseline methods for query auto-completion, Table 7 summarizes their key characteristics, including their approach, data structure, memory usage, and computational complexity. This comparison highlights the trade-offs between accuracy, efficiency, and memory consumption.

Table 7 – Summary of Baseline Methods for Query Autocompletion.

Methods	Key Feature	Advantages	Drawbacks
EAT ICAN	Incremental active nodes in trie	Efficient for small data, precomputes classes for faster start-up	High memory usage for large datasets; costly for short prefixes
ICPAN	Pivotal active nodes	Reduces memory usage and improves query response time by pruning unnecessary active nodes	Still sensitive to trie size; requires computation for pivotal nodes
META	Compact tree for top-k and threshold queries	Avoids redundant computation; supports top-k queries; scalable to larger datasets	Requires additional structure for compact indexing
IncNGTrie	Deletion-marked variants	Speedup in query processing (up to 2x); efficient for error-tolerant searches	High memory usage; exponential trie growth; reduced efficiency in indexing large datasets
BEVA	Optimized active node evaluation	Efficient memory usage; faster query processing compared to other trie-based methods	Requires detailed implementation; specific optimizations for active node evaluations

2.2 Query autocompletion ranking phase

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 2, number of errors in match mode 3, 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 discuss ranking with pruning strategies here as future work for this thesis.

Besides efforts to improve the efficiency of query autocompletion methods, there has also been 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.

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 thesis.

The most basic strategy for ranking query completions is to use the query's popularity in the search log history. But, time may affect this information, hence time-related aspects have been studied for QAC, such as the popularity of recent queries trends, the periodic phenomena, or the predicted future query popularity based on time series analysis (Whiting and Jose, 2014; Cai et al., 2014; Shokouhi and Radinsky, 2012b) are frequently considered in time-related QAC approach. Another approach is to use the personal information of a user to infer their specific interest and search intent. This approach consists of previous queries in their current session (Bar-Yossef and Kraus, 2011b), search query behavior such as adding terms (Jiang et al., 2014b) as well as their profile context as gender and age (Shokouhi, 2013b). In the next section, we detail the main approaches to rank candidate queries.

2.2.1 Approaches to rank candidate queries

In this section we define some candidate query ranking approaches for QAC. We assume that we have a query log containing past queries Q, documents click C, a collection of documents D with possible suggestions and, for personalized approaches, all queries previously searched by the current user as the *user context*. So, we formalize each approach as a scoring function $scoring(q)$, where $q \in Q_p$, according to Di Santo et al. (2015).

Most popular ranker (MP) : It is a naive and *baseline* approach based on the queries past popularity. Bar-Yossef and Kraus (2011b) named as the *most popular* ranker model:

$$MPC(p) = \frac{f(q)}{\sum_{q_i \in Q} f(q_i)} \tag{2.6}$$

where, $f(q)$ denotes the frequency of query q in search log Q and the denominator of this division is a normalizer by the total sum of query frequencies. As shown in Di Santo et al. (2015), in the literature, there are other variants to this approach. Shokouhi and Radinsky (2012b) replaced the query's actual frequency with a predicted frequency, while Strizhevskaya et al. (2012) modeled the query frequency using a time series.

Term occurrence ranker (TO) : ranks candidates queries based on the term popularity of those candidates. Term popularity is calculated as the mean of the frequency of the term inside the query log and the $TF - IDF$ of the term within the corpus of documents being searched, according to Di Santo et al. (2015). The score for a candidate query is the mean of the scores for its terms.

$$TO(p) = \frac{\sum_{t \in q} \frac{(tfidf_{ql}(q) \cdot tfidf_c(q))}{2}}{|q|} \tag{2.7}$$

String similarity ranker (SS) : ranks candidates queries based on query similarity. The similarity is calculated between a query and all previous queries issued by the user who submitted that query. The Leveishtein edit distance (Levenshtein, 1966) and other similarity measures can be used.

$$SS(p) = \frac{\sum_{q_i \in Q} LED(q, q_i)}{|Q|} \tag{2.8}$$

Clicked documents ranker (CR) : ranks candidates queries based on documents clicked. This approach models the user's interests as the content of documents previously clicked (D_c), where this set is represented by the terms present inside the document titles. The candidate query is represented in the same way, but considering all of the documents previously clicked for that candidate query in term query log (D_q). The *cosine similarity* measure is used to score the candidate query representation by its similarity to the representation of the user's interests (Di Santo et al., 2015).

$$\begin{aligned} CR(p) = cosine(q_d, c_d) \; q_d &= \{t | t \in d_{title}, d \in D_q\} \\ c_d &= \{t | t \in d_{title}, d \in D_c\} \end{aligned} \tag{2.9}$$

2.3 Pattern matching using bit-parallelism approach

Approximate string matching refers in general to the task of searching for substrings of a text that are within a predefined edit distance threshold from a given pattern.

This is a classic problem in computer science, with applications for example, in spelling correction, bioinformatics, and signal processing. There are a variety of algorithms that solve the pattern matching problem and a good part of them are explored in detail in Navarro (2001). In this section, we only address the bit-parallelism approach to solve the pattern-matching problem because it has important concepts to understand our contributions in Chapter 4.

The bit-parallelism approach has two main applications: (1) parallelize the work of the non-deterministic automaton that solves the pattern-matching problem and (2) parallelize the work of the dynamic programming matrix. The application of this technique in string matching was first presented in Baeza-Yates and Gonnet (1992). It consists in taking advantage of the intrinsic parallelism in bit operations like AND/OR inside a computer word. Since 1992, bit parallelism is directly used in string matching for matching efficiency improvement.

The formal notation is given by w, being the length of the computer word (in bits). The sequence $b_1...b_m$ 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, $\hat{}$ to denote the bitwise-xor, $\sim$ to denote the complement of all the bits, $<<$ to denote the bitwise-shift-left, which moves the bits to the left and enters zeros from the right, i.e, $b_m b_{m-1}...b_2 b_1 << r = b_{m-r}...b_2 b_1 0^r$ and $>>$ to denote the bitwise-shift-right, which moves the bits to the right and enters zeros from the left, i.e, $b_1 b_2...b_{m-1} b_m >> r = 0^r b_1 b_2...b_{m-r}$.

2.3.1 Shift-OR

We now explain the first bit-parallel algorithm named Shift-OR (Baeza-Yates and Gonnet, 1992), since it is the basis of much of which follows. The algorithm searches a pattern in a text and simulates the computation of a non-deterministic automaton by parallelizing its operations to find a match to the pattern. The automaton is presented in Figure 11 to the pattern "ufam" and no errors.

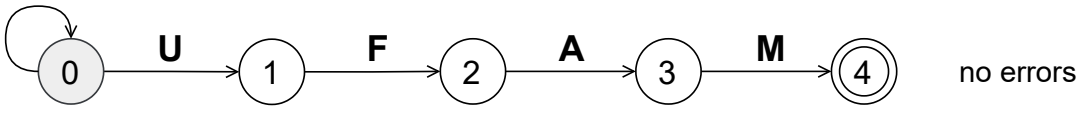


Figure 11 – Non-deterministic automaton that searches the pattern "ufam" exactly.

Given a pattern p of length m and a text t of length n, we representation of the automaton have $m + 1$ states. The Shift-OR algorithm first builds a table B, which for each character $c \in \Sigma$ a bit mask is set as $B[c] = b_1...b_m$. The mask in $B[c]$ has the ith bit equal to 0 if and only if the character in $p[i] = c$. Next, the state of the search is kept in a register named here as $R0$. We initialize $R0$ to no errors with bit values $1^{|p|}$.

For each new character processed from the pattern, the register $R0$ should be updated to simulate the active states of the automaton in Figure 11. The register is updated using the following formula:

$$R0' = (R0 >> 1) \mid B[t[i]] \tag{2.10}$$

After the pattern is processed, the correspondence between p and t can be verified when $R0$ has the last bit equal to 0, i.e., when the last state is active, which represents that the pattern ended with a match. Activation of a state or matching only takes place when the previous state of the automaton is already activated, otherwise, the state cannot be activated. The match verification can be performed using the naive bitwise operation $r_i = (R0 \ \& \ 0x1)$, where $r_i = 1$ meaning a match at position i or $r_i = 0$ a non-match. The Shift-OR algorithm achieves $O(mn/w)$ worst-case time. For patterns longer than the computer word, i.e. $m > w$, the algorithm uses $[m/w]$ computer words for the simulation. The algorithm is $O(n)$ on average.

2.3.2 Shift-OR-Extended

Wu and Manber (1992) extended the Shift-OR algorithm to handle wild cards and allow errors. This algorithm also simulates the states from an automaton and here we named it as Shift-OR-Extended. The automaton is presented in Figure 12 to the pattern "ufam" with a maximum number of errors equal to $k = 1$. The symbols Σ and λ represent a predefined alphabet and the empty string, respectively.

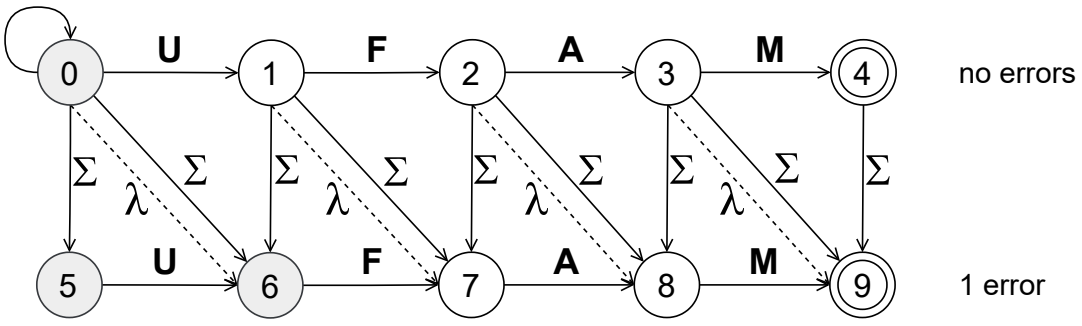


Figure 12 – Non-deterministic automaton that searches the pattern "ufam" exactly and with 1 error.

Given a pattern p of length m and a text t of length n, we automaton representation have $(m+1)(k+1)$ states. The Shift-OR-Extended algorithm first builds a table B, which for each character $c \in \Sigma$ a bit mask is set as $B[c] = b_1...b_m$. The mask in $B[c]$ has the ith bit equal to 0 if and only if the character in $p[i] = c$. Next, the state of the search is kept in registers named here as $R0$ and $R1$, being the number of registers equal to the number of errors allowed plus one more, in this case, we allow 1 error and have two registers. We initialize $R0$ to no errors and $R1$ to 1 error, with bit values $1^{|p|}$ and $01^{|p|-1}$,

respectively. This initial values to B, $R0$ and $R1$ are presented in Figure 13. Also, for each new character that Shift-OR-Extended processes from the pattern, two auxiliary variables $R0_{prev}$ and $R1_{prev}$ are used to store the previous values of $R0$ and $R1$.

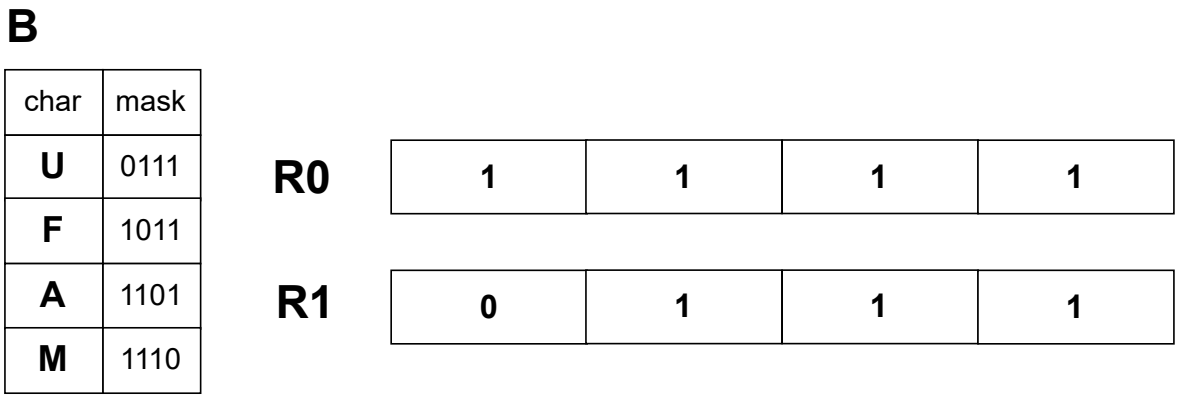
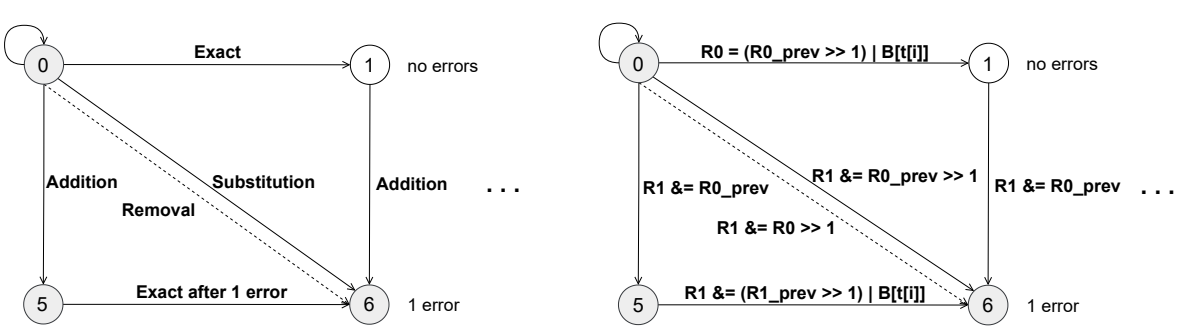


Figure 13 – The table B with the bit mask values to the pattern "ufam" and the registers $R0 = 1111$ and $R1 = 0111$ that store the initial state of the search before processing the pattern.

The transitions of the automaton in Figure 12 represent the edit distance operations on characters, which include insertion, removal, and substitution. We show the edit distance operations on each transition in Figure 14(a). The horizontal transition on the first line simulates an exact match. The horizontal transition on the second line also simulates an exact match, but when 1 error has already occurred previously. The vertical transition on the first and second lines simulates adding a character. The diagonal transition simulates the substitution of one character for another. Finally, the dashed diagonal transition simulates the removal of a character. Whenever the second line of the automaton is reached, it means that at least 1 error has occurred.

For each new character processed from the pattern, the registers $R0$ and $R1$ should be updated to simulate the active states of the automaton in Figure 12. The registers are updated using the following formulas:



(a) Edit distance operations. (b) Bitwise operations.

Figure 14 – (a) Edit distance operations represented in the automaton. (b) Bitwise operations are represented in the automaton.

$$R0 = (R0_{prev} >> 1) \mid B[t[i]] \tag{2.11}$$

$$R1 = ((R1_{prev} >> 1) \mid B[t[i]]) \;\&\; (R0_{prev}) \;\&\; (R0_{prev} >> 1) \;\&\; (R0 >> 1) \tag{2.12}$$

We present in Figure 14(b) each part of the above-described formulas and the relationship of each edit distance operation with the parts of these formulas, which allow the string matching of a pattern with no errors or 1 error to the text. If we need to allow more errors, for example, allow two errors, just add a new register $R3$ and calculate the value for it with the same formula as $R1$, replacing the places where the information is $R1$ with $R2$. An auxiliary variable $R3_{prev}$ also is required.

After the pattern is processed, the correspondence between p and t can be verified when $R0$ or $R1$ has the last bit equal to 0, i.e., when the last state is active, which represents that the pattern ended with a match. The match verification can be performed using the naive bitwise operation $r_i = (R_i \;\&\; 0x1)$, where $r_i = 1$ meaning a match at position i or $r_i = 0$ a non-match. The Shift-OR-Extended algorithm achieves $O(k\lceil m/w \rceil n)$ worst-case time and average case the algorithm is $O(kn)$ on average, where k is the number of errors.

2.3.3 Parallelizing the dynamic programming matrix

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. Hyyro's algorithm is another of our baselines to compute the edit distance.

In this thesis, we are also interested in parallelizing the work of the dynamic programming matrix by using the compact representation of the k-diagonals proposed by Ukkonen and Wood (1993) and applying arithmetic operations over the bits.

3 Evaluation Environment

This chapter presents the environment to evaluate the performance of our contributions to error-tolerant query autocompletion.

3.1 Server settings

The algorithms were implemented in C++ version 11, compiled using GCC 7.4.0 and optimization level -O3. Our server of evaluation has the following specifications:

- Intel Xeon E5-4617 processor (2.90 GHz);
- 64 GB of RAM;
- The machine cache sizes are: L1d of 32 KB, L1i of 32 KB, L2 of 256 KB, L3 of 15,360 KB;
- Operation system Ubuntu 18.04.1 LTS;

3.2 Datasets

We present experiments to evaluate the performance of the studied data structures and algorithms using four distinct datasets. Most of the experiments are reported using a query autocompletion suggestion dataset extracted from JusBrasil. We also report results using three synthetic datasets adopted in previous research articles, DBLP[1], MEDLINE[2] and UMBC[3]. JusBrasil[4] is a Brazilian law-tech company that provides a vertical search service for its users. This dataset, which was previously introduced in Ferreira et al. (2022), 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 9 presents details of the JusBrasil dataset.

Table 8 presents two examples of prefix queries and suggestions that belong to the 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

[1] https://dblp.uni-trier.de/faq/How+can+I+download+the+whole+dblp+dataset, dataset release dblp-2019-04-01.xml
[2] https://www.nlm.nih.gov/databases/download/pubmed_medline.html
[3] https://ebiquity.umbc.edu/resource/html/id/351/UMBC-webbase-corpus
[4] http://www.jusbrasil.com.br

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 matches the prefix with 1 error. In the second example of a 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.

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 8 – Examples of prefix queries and suggestions of JusBrasil and DBLP datasets.

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 then gets the suggestion from the JusBrasil query autocompletion system, and finally 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 9 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.

File	Size (bytes)	Items	Distinct Words	Avg Item Len
Query suggestions	633,078,803	23,374,740	493,173	27.0
Prefix queries	12,189,441	648,264	100,649	18.8

Table 9 – Statistics about query suggestions and prefix queries typed by users in the JusBrasil dataset.

Table 10 shows the importance of performing an error-tolerant prefix search when performing query autocompletion in the JusBrasil dataset. To evaluate such importance, Table 10 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 10, 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. These results conclude that the introduction of errors has a large impact on the capacity of the system to suggest correct queries. Of course, even with an 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.

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

Table 10 – Hit rate (%) for relevant queries in the JusBrasil dataset as we vary the number of errors allowed in the search.

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 researchers 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 presenting experiments with only the JusBrasil dataset, the only dataset containing real query logs that we had available, we have also included three datasets adopted in previous work.

The datasets chosen are DBLP, MEDLINE, and UMBC, and they were previously adopted in articles that studied error-tolerant prefix search methods. We classified them as synthetic since these synthetic datasets do not contain query logs and were not extracted from a real case query autocompletion service. Table 8 presents examples of two prefix queries and suggestions that belong to DBLP. They are useful for showing 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", 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). MEDLINE[5]: 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. UMBC[6]: 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 11 presents detailed statistics about these synthetic datasets. In all cases, we have removed duplicated items from the datasets.

Dataset	Size (bytes)	Items	Distinct Words	Avg Item Len (bytes)
MEDLINE	2,555,416,200	27,941,081		91.4
DBLP	334,999,905	4,378,548	208,806	76.5
UMBC	17,427,838,773	38,449,902	1,197,965	453.2

Table 11 – General statistics about the synthetic datasets adopted in the experiments.

While these three 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 simulates any particular distribution of errors in a query log. However, while the methodology for creating the queries does not guarantee the reproduction of user behaviors when interacting with a real-case autocompletion dataset, the inclusion of these three datasets is important because they have been used for experiments in previous articles.

[5] https://www.nlm.nih.gov/databases/download/pubmed_medline.html
[6] https://ebiquity.umbc.edu/resource/html/id/351/UMBC-webbase-corpus

3.2.1 Static and dynamic scenarios

We have experimented with the 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 Chapter 5. We also consider that the dataset is lexicographically sorted, so we can use ranges by position of the items. The sort applied to the datasets was done in an offline script using the native method "sort" from the Python programming language. 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 Chapter 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 Chapter 5. In the dynamic scenario, we adopted linked lists to represent the elements in the burst containers.

3.3 Baselines methods

The following methods were selected or implemented to be baselines used in comparisons with our proposed ideas:

- ICPAN (Li et al., 2011) is one trie-based algorithm for error-tolerant query autocompletion which improves ICAN (Ji et al., 2009) by reducing the size of active nodes set named pivotal active nodes. The code was provided by authors from Li et al. (2011).

- BEVA (Zhou et al., 2016) is one trie-based algorithm for error-tolerant query autocompletion which reduces the size of active nodes set by keeping only the boundary active nodes and using the Edit Vector Automaton (EVA) structure to compute the edit distance between the query and the prefix query. The default automaton used in BEVA for this work is EVA. BEVA is among the best algorithms proposed to be used with tries. We have completely implemented the BEVA method[7].

- 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.

[7] We have validated the performance and correctness of our BEVA source code by comparing it to the original binary code provided by the authors. Results indicate our implementation is even faster than the binary code provided by the authors.

- 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.

- 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.

3.4 Baselines structures

The following structures were implemented to be baselines used in comparisons with our proposed ideas:

- Trie or full trie: The standard trie data structure.

- CPT: The CPT data structure 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).

- Suffix array: 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 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 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.

3.5 Evaluation metrics

In this section, we present some metrics to evaluate our contributions effectively and efficiently.

3.5.1 Efficiency

The main efficiency metrics adopted are related to processing speed and costs such as time performance, memory requirements, cache hit rate, and throughput rate when processing queries in query autocompletion methods.

3.5.1.1 Time performance

The time performance of query autocompletion methods is composed of processing and fetching. The processing is the time spent to retrieve all the active nodes (defined in Section 2.1.4) which are the results of a query. The fetching is the time spent on the algorithm retrieving all strings from active nodes. In most experiments of this thesis, we present the time to process queries without separating the fetching times, and when we need to separate, we make it clear.

The time performance to a query is computed as the sum of the time spent for each character of the prefix query, for instance, if the prefix query has 17 characters, the time performance to this prefix query is the sum of each one of the 17 characters because the algorithms process a prefix query character by character until to reach the prefix query size limit. The time performance reported is an average of times per query tested in milliseconds. The time was collected with the chrono[8] time library from C++ programming language.

3.5.1.2 Memory requirements

The memory consumption is captured through the 'ps' command available in Linux distributions, using the parameters '-p' which allows to inform the number of the specific process that is running the algorithm, and '-o size' to get the memory used by this

[8] https://www.cplusplus.com/reference/chrono/

process. The complete command is called '/bin/ps -p <PID> -o size', where <PID> is the number of the process that is running the algorithm. The memory is captured in bytes and converted to megabytes. This command is called multiple times during the query processing of one query and a memory consumption average is calculated for these calls. The final memory consumption reported is the average of the memory consumption of all queries tested.

The memory consumption experiments were performed separately from the time performance experiments, as the memory usage collection causes an overhead in the execution of the algorithm. Due to this overhead, the memory consumption experiments were limited to just 100 queries, as the average of 1,000 executions was similar.

3.5.1.3 Cache hit rate

The computer's memory hierarchy is organized from the high level to the low level. High-level memory has lower capacity, higher speed, and higher cost. On the other hand, low-level memory has higher capacity, lower speed, and lower cost. Our experiments were performed considering the indexes fully stored in the RAM of the computer. The memory hierarchy when using RAM includes: Registers, the highest level, are fast memories inside the processor. A certain number of cache memory levels are indicated as L1 and L2, and so on. Finally, the internal or main memory (RAM).

The cache memory is a quick access device used to store frequently accessed data, which serves as an intermediary between the computer processor and the storage device as RAM. The main advantage of using a cache memory is to avoid accessing the slower storage device. When the processor needs to access data, if the data is already in the cache memory, we call this operation *cache hit*. When data is not in the cache memory and needs to be fetched to the next level, we call this operation *cache miss*.

To evaluate the cache hits and cache misses during the query processing of query autocompletion algorithms we selected the CacheGrind program[9]. It simulates a machine with independent first-level instruction and data caches and a unified second-level cache. Some modern machines have three or four levels of cache. For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and last-level caches. The reason for this choice is that the last-level cache has the most influence on runtime, as it masks access to the main memory.

The performance of a cache is measured using the metrics hit hate. The hit rate is the fraction or percentage of users that a relevant suggestion appears in the list of results when considering all users from the system. This metric is calculated based on the following formula:

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

$$hit_{rate} = \frac{|U_{hit}|}{|U|} \tag{3.1}$$

, where $|U_{hit}|$ is the number of users with relevant result and U is all users.

When using this metric is important to observe the length of the list of results retrieved. If the length is larger, we will have a higher hit rate, because there is a higher chance that the relevant result be included in the list of results.

3.5.1.4 Throughput rate

A load test is an experiment to measure the maximum point at which a system is able to respond to requests within an acceptable time. When a system is no longer able to respond to requests within an acceptable time, we say that the system has saturated and has reached its maximum working limit. The acceptable time to the query autocompletion services is 100 milliseconds according to Miller (1968).

To evaluate the point of saturation of the query autocompletion methods experimented we have implemented a server with an endpoint as '/autocompletion?q=' to receive the queries. This server was implemented using the Crow[10] library. Crow is a C++ microframework for the web inspired by Python Flask[11]. To make multiple calls to the server, we adopt the Vegeta[12] program, which is a versatile HTTP load-testing tool built out of a need to drill HTTP services with a constant request rate, to compare the throughput of the methods experimented.

The number of queries per second attended by a system is named as throughput rate, and the saturation point is the maximum throughput rate supported by the system.

When performing the throughput rate experiments, the number of strings to be fetched by each ETQAC system was limited to 1,000 to ensure the server did not suffer delays during the multiple calls of requests. This number is commonly fetched in practical scenarios due to the functions of pruning that are applied during the fetching step of ETQAC systems.

[10] https://github.com/ipkn/crow
[11] https://flask.palletsprojects.com/en/2.1.x/
[12] https://github.com/tsenart/vegeta

4 BWBEV

In this chapter, we present a new method called **B**it**w**ise **B**oundary **E**dit **V**ector (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 2.2 using an efficient bit-parallelism approach. Then, from now we demonstrate how to calculate the Equation 2.2 efficiently using our proposed ideas. The complete source code can be found at GitHub repository[1].

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 2.1.3.1, 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 12 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 4, 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.

Decimal	Unary
0	111
1	011
2	001
3	000

Table 12 – 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 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.

[1] https://github.com/vdbergg/bwbev

4.2.1 Add 1

Table 13 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 13.

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

Table 13 – 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 Mininum 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 14.

	[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
$u \mid v$	001	011	111	011	001
Final decimal values	**2**	**1**	**0**	**1**	**2**

Table 14 – Applying min operation between two unary numbers u and v.

4.2.3 Align Positions

To align position $i+1$ of an edit vector v with position i, we can shift v $\tau+1$ bits left, ie, $v << \tau+1$. Similarly, to align the position $i-1$ with position i, we shift v $\tau+1$ bits right, ie, $v >> \tau+1$. These operations can be performed in parallel for all positions of the edit vector, as shown in Table 15. 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
$v << \tau+1$	011	111	011	001	000
Final decimal values	**1**	**0**	**1**	**2**	**3**
$v >> \tau+1$	000	001	011	111	011
Final decimal values	**3**	**2**	**1**	**0**	**1**

Table 15 – Aligning the position $i+1$ of v with position i using the bitwise operation $v << \tau+1$ and aligning the position $i-1$ of v with position i using the bitwise operation $v >> \tau+1$.

With our unary representation to pack each edit vector value and the arithmetic operations described, we can efficiently compute the Equation 2.2 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 2.2 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 quite common 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 2.2 can be replaced by:

$$\begin{aligned} v_{j+1}[i] = min(&v_j[i]+1, \\ &v_j[i+1]+1, \\ &v_{j+1}[i-1]+1) \end{aligned} \tag{4.1}$$

But,

$$v_{j+1}[i-1]+1 = min(v_j[i-1]+2, \\ v_j[i]+2, \\ v_{j+1}[i-2]+2) \tag{4.2}$$

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 4.1 and 4.2, we have:

$$v_{j+1}[i] = min(v_j[i]+1, \\ v_j[i+1]+1, \\ v_{j+1}[i-2]+2) \tag{4.3}$$

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

$$v_{j+1}[i] = min(v_j[i]+1, \\ v_j[i+1]+1, \\ v_{j+1}[i-(\tau+1)]+(\tau+1)) \tag{4.4}$$

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

$$v_{j+1}[i] = min(v_j[i]+1, \\ v_j[i+1]+1) \tag{4.5}$$

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 1 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.

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

```
1:  procedure COMPUTENEWEDITVECTOR(v_j, b, τ)
2:      v_{j+1} ← (v_j >> 1) | (v_j << τ)
3:      v_{j+1} ← v_{j+1} & [0[1^τ]]^(2τ+1)
4:      if b ≠ 0 then
5:          mask ← 1^{τ+1} 0^{2τ×(τ+1)}
6:          do
7:              if b & 1[0]^{2τ} then
8:                  v_{j+1} ← v_{j+1} | (v_j & mask)
9:              mask ← mask >> (τ + 1)
10:             b ← b << 1
11:         while b ≠ 0
12:         do
13:             tmp ← v_{j+1}
14:             v_{j+1} ← v_{j+1} | ((v_{j+1} >> (τ + 2)) & [0[1^τ]]^(2τ+1))
15:         while tmp ≠ v_{j+1}
16:     return v_{j+1}
```

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 2, we describe the changes necessary to process edit distance between any two strings and we refer to this algorithm as *Bitwise Edit Vector (BWEV)*.

When implementing both methods, we empirically verified the correctness of our edit vector computation code by computing all possible transitions from $\tau = 1$ 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 4 and 5. Second, we must search by simply iterating over each character in the string s. And for each character in s, a value of τ and the Algorithm 1, 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 8 to 14.

Algorithm 2 Computes edit distance between two strings p and s limited to a maximum number of errors τ.

```
1:  procedure computeEditDistance(p, s, τ)
2:      /* preprocessing */
3:      H ← 0                                   ▷ H is the same bitmap table of BEVA.
4:      for j = 1, 2, ...|p| do
5:          H[p[j]] ← H[p[j]] | 1 << (|p| − j + τ)
6:
7:      /* searching */
8:      v ← v_0                                 ▷ v_0 is the initial edit vector
9:      for i = 1, 2, ...|s| do
10:         b ← H[s[i]] >> (|p| − i)
11:         b ← b << (w − (2τ + 1))
12:         v ← computeNewEditVector(v, b, τ)
13:         if v = 0 then
14:             Break
15:     return v[τ + 1 + (|p| − |s|)] ▷ when |p| ∈ [|s| − τ, |s| + τ] or more than τ 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 jth-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 $b \leftarrow b << 1$ 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 Raffinot, 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 aggregated computer words.

4.6 Pruning

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. Rather, each match has a score associated with it, and only the top-k best-scored results are retrieved. The actual challenge of the top-k algorithm is not only to return the correct top-k answers but also to do it as quickly as possible.

To address the challenge of computing top-k results in the BWBEV algorithm for approximate prefix search, we have studied several alternatives for fast retrieval of the best-scored matches. In this new scenario, the score function takes the edit distance as one of the features to compose the final score of each result. We assume that the remaining

features are available before finishing the query processing, as this is likely the case in most query autocompletion systems that allow approximate matches. However, the number of errors is a feature that can only be computed after finishing the prefix search, which poses a challenge for developing pruning methods.

When pruning, we have made modifications to the trie by incorporating information about the maximum possible score among all the children of each node in the trie. As we cannot know in advance the number of errors for the node, we compute the maximum score considering every possible error level allowed in the system. As this number is typically small, only a few numbers are required for each node. Therefore, we can assume that we can obtain the maximum score of a node.

Two pruning strategies have been employed in this study:

Pruning 1: Consists in keeping only the top-k best-scored results in a top-k heap. In this strategy, after calculating all active nodes from a specific number of errors allowed, we compute the maximum score that each active node may achieve and then get only the top-k active nodes. Thus, we set the pruning threshold θ as the minimum max score among the selected active nodes. We then start to traverse the active nodes to fetch the final results. Keeping the best top-k results in a minimum heap, and inserting only the results with a score higher than the threshold θ until the heap becomes full. Once the heap becomes full, the threshold θ is set as the smallest score in the heap, and the smaller element is substituted whenever a new result has a score higher than it.

Pruning 2: The second pruning strategy is to perform the search incrementally. We start the prefix search with $\tau = 0$ (exact match) to get the top-k results with zero errors using the *Pruning 1* strategy. Next, the query processing is performed to $\tau = 1$ and the heap obtained to $\tau = 0$ is utilized to avoid the processing of nodes with a maximum score below the score threshold in the heap. No results with zero errors are inserted in this second round, and only active nodes containing 1 error are taken into account in the fetching. It is important to note that with this strategy, nodes can be pruned not only at the fetching phase but also at the matching phase, as all results taken into account in a round would necessarily have the same number of errors. The procedure is incrementally repeated to insert results with higher error levels until we reach the maximum number of errors allowed, with results for steps up to $\tau = i$ being used to prune the processing for $\tau = i + 1$.

4.7 Experiments

This section presents the experiments we carried out to evaluate the performance of the proposed BWBEV method.

4.7.1 Experiments Utilizing Synthetic Datasets

The results obtained from processing queries on the synthetic datasets DBLP and MEDLINE are presented in Tables 16 and 17, 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 16 – DBLP - Processing times when using BEVA, BEV, ICPAN, and BWBEV to prefix queries size 9 and 17 and varying τ from 1 to 3.

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

4.7.2 Comparison to QAC Baselines Methods

In Table 18, 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 18 – JUSBRASIL - Processing times when using BEVA, BEV, ICPAN, and BWBEV and varying τ from 1 to 3.

4.7.3 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 26, 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.

4.7.4 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

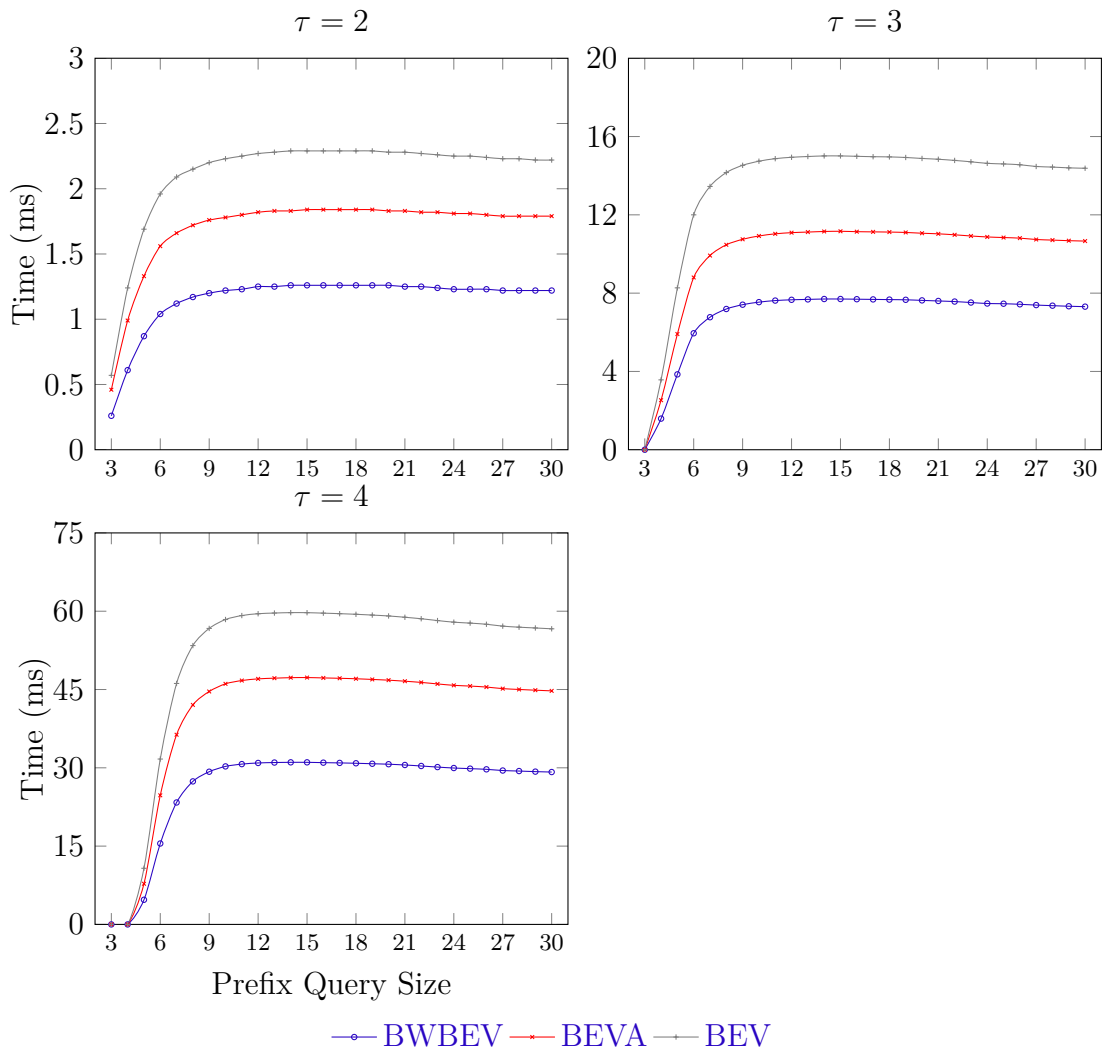


Figure 15 – Time performance (in milliseconds) while adjusting the prefix query size and the permissible number of errors within the JUSBRASIL dataset.

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 32. 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.

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

4.7.5 Scalability

We conducted experiments using Vegeta[2], a versatile HTTP load-testing tool developed to test HTTP services with a constant request rate.

The target application we address in our experiments does not require the system to provide all the matches when searching for a prefix. Then only the top-k best-scored results are retrieved. To address the challenge of computing top-k results in the BWBEV and baselines algorithms for approximate prefix search, we have adopted an alternative (Bar-Yossef and Kraus, 2011a) to quickly retrieve the best-scored matches. We adopted the well-known most popular completion (MPC) approach, based on the search popularity of queries matching the prefix typed by the user.

Figure 27 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

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

has typos.

As shown in Figure 27, 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.

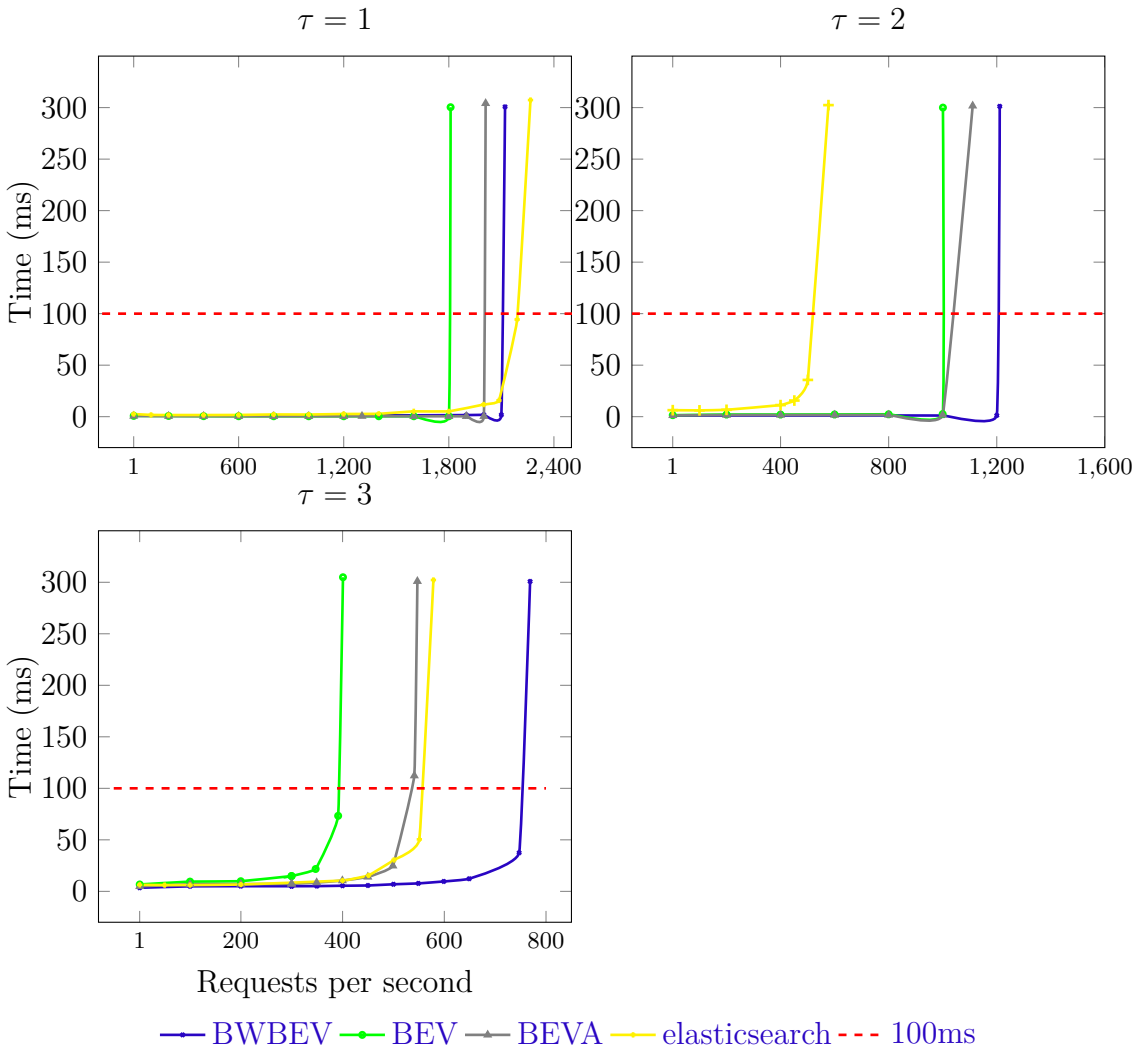


Figure 16 – Processing time (in milliseconds) with increasing requests per second while varying τ (ranging from 1 to 3) within the JUSBRASIL dataset.

4.7.6 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 28.

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.

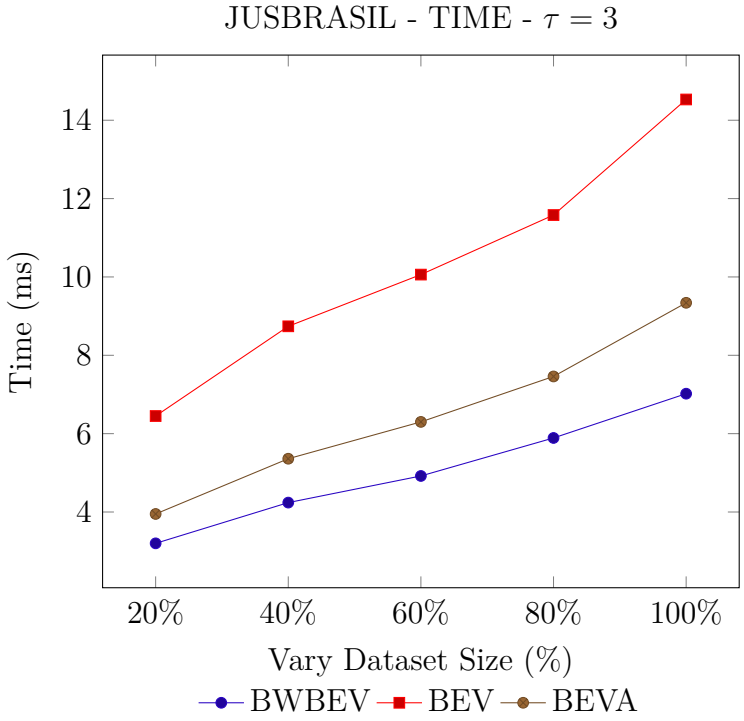


Figure 17 – Time performance (ms) of BEVA, BEV, and BWBEV when indexing distinct amounts of JUSBRASIL.

.

4.7.7 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 and Wood (1993).

- EV is the computation of edit vectors proposed by Zhou et al. (2016), adapted by us here to allow online edit 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 20 and 21, 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 1. When the strings are not similar, BWEV just needs to process the Equation 4.5 instead of the full Equation 2.2.

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 20 – Query processing of pair of strings no similarity in DBLP dataset.

In the *same strings adding errors set*, as shown in the tables 22 and 23, 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

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 21 – Query processing of pair of strings no similarity in MEDLINE dataset.

BWEV due to the need to complete the edit vector computation as shown in lines 5 to 15 in the Algorithm 1 to process the full Equation 2.2.

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 22 – 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 23 – Query processing of pair of strings with similarity between 0 and 4 errors in MEDLINE dataset.

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

4.7.8 Top-*k* query processing

Finally, we present the experiments conducted to compute top-k results using the proposed pruning methods, which were performed solely on the JUSBRASIL collection. The score adopted for ranking combines the features available in the collection, as shown in Equation 4.6:

$$score(p, q) = (freq_{log}(q) + nr(q) + 1)) \times \left(\frac{100}{log_2(max(|p|, 2))} \right)^{\tau - ed} \tag{4.6}$$

Here, $freq_{log}(q)$ is the number of times the suggestion q appears in JUSBRASIL log, $nr(q)$ is the number of results provided by the search system for the given query

suggestion q, $ed(p,q)$ is the number of errors between the prefix p and suggestion q, τ represents the maximum number of allowed errors, and $|p|$ is the size of the prefix typed by the user. It should be noted that other features could be included to produce more sophisticated ranking functions, but such extensions fall outside the scope of the present work.

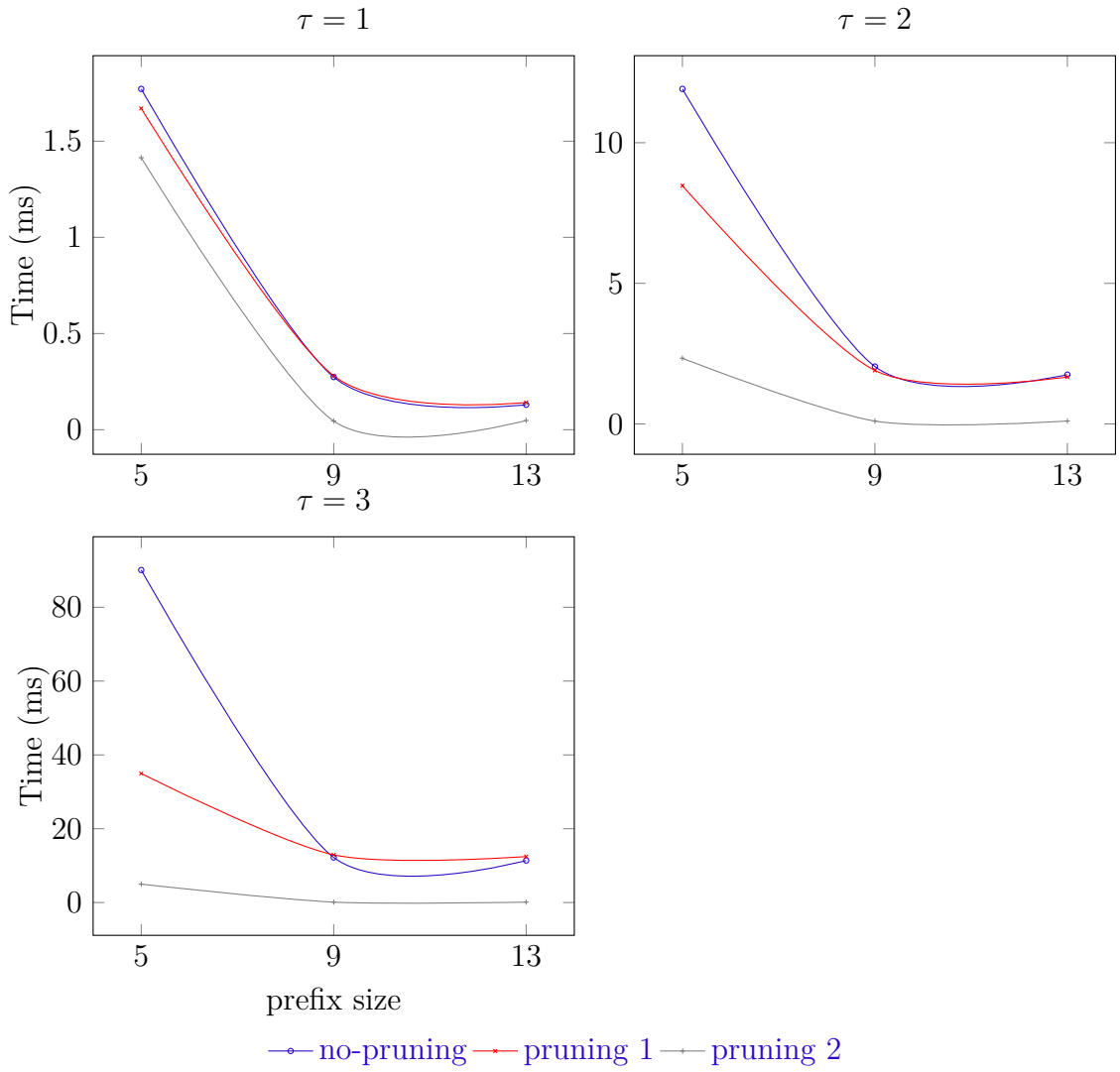


Figure 18 – Processing and fetching times in ms when varying the prefix size and the number of errors allowed for top-10 results in JusBrasil dataset.

In Figure 18 we present the time performance (in ms) of processing and fetching operations for prefixes with sizes 5, 9, and 13, with τ values varying from 1 to 3, using the pruning methods no-pruning, pruning 1 and pruning 2. It is observed that the method of pruning 1 was faster for all τ significant differences occurring when processing smaller prefixes and for $\tau > 1$. This happens because short prefixes or large values of τ often have a large list of active nodes with many suggestions to be retrieved, and since these active nodes are at the top of the trie, they have more child nodes in case of short prefixes. As a result, the no-pruning method tends to analyze many nodes and consequently many

suggestions. When the prefix is longer or $\tau < 2$, the list of active nodes is usually small, which does not add much overhead to the fetching operation and any improvement in fetching is rarely noticeable. In table 24, we provide more detailed information about this behavior for $\tau = 3$. For instance, in the prefix of size 5, the fetching time in pruning 2 drops to 4.87ms compared to 83.54ms in the no-pruning method, resulting in a reduction of about 94.17%. When the prefix size increases to 9, the fetching time in pruning 2 drops to 0.0007ms compared to 1.22ms for the no-pruning method. However, both are very low and acceptable times for a fetching operation in a QAC system.

Methods	Time (ms) with Pruning Methods for $\tau = 3$			
	$\|q\| = 5$		$\|q\| = 9$	
	Processing	Fetching	Processing	Fetching
no-pruning	6.54	83.54	10.98	1.22
pruning 1	6.83	28.13	12.05	0.87
pruning 2	0.11	4.87	0.11	0.0007

Table 24 – Average time performance (ms) when using BWBEV for $\tau = 3$, with prefix size values 5 and 9 for top-10 results in methods no-pruning, pruning 1, and pruning 2.

5 Efficiency Issues

In this chapter, we discuss possible alternatives to efficiently implement the error-tolerant search methods using tries and their variations. 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, meaning that no insertion or removal is allowed before a complete index rebuilding, and sorted, or at least partially sorted. Second, we discuss alternative trie-building solutions that are available when the dataset is static and sorted.

5.1 Reducing fetching costs

An alternative way to reduce the fetching cost is to assume that the string dataset is previously sorted. In such a case, the strings represented by each node may be stored in consecutive positions in the dataset, and thus we can keep the range of elements of the dataset associated with each node as information to avoid the subtree traversal when fetching the results. This idea has been previously adopted by other authors when dealing with tries in scenarios where the dataset is static. See for instance Pibiri and Venturini (2017), and see also the work of Gog et al. (2020), which study the use of this idea when implementing a query autocompletion system. Figure 19 shows the burst trie[1] containing range information for our example dataset.

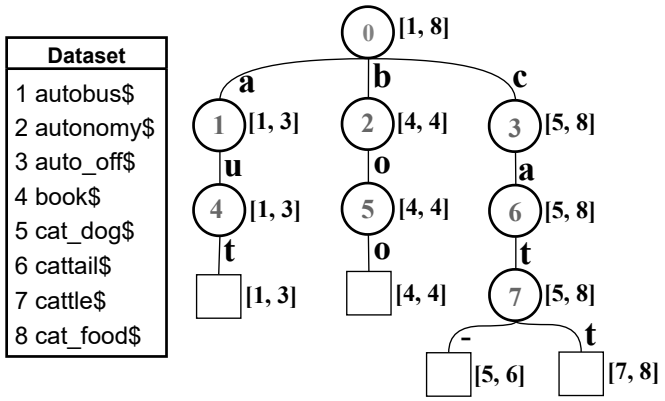


Figure 19 – Burst trie with MCD set to 3 and MCK also set to 3 with the range information to each node and container in the burst trie to our sample dataset.

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

[1] We show the idea of the range with burst trie for convenience, but it could be any other trie representation.

For instance, the index-building process takes just about one minute for the datasets we adopted in our experiments.

Besides 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, since instead of storing the strings in the container, we may represent them by only storing the range of the keys in a container, and use such range information to have fast access to container elements directly in the dataset. For instance, when reaching the leftmost container of Figure 19, we find a range $1 - 3$, meaning that the first three strings from the dataset are stored in the container. Updates in the autocompletion systems in this case might be done using a smaller index structure to log updates, allowing fast updates in the system.

5.2 BFS index building

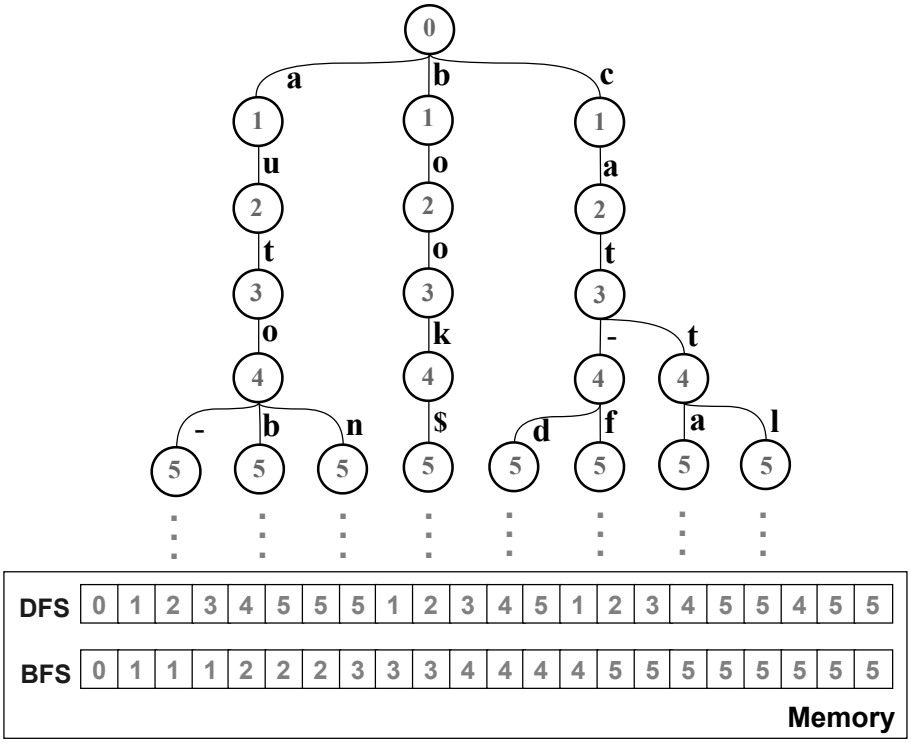


Figure 20 – Memory organization of nodes of distinct levels in a trie when building the index using the DFS approach and when using the BFS approach. Nodes are labeled according to their levels in the trie.

We may also create the trie nodes in a cache-friendly disposition when considering the dataset as static and previously 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 since they

need to find a new set of answers for each key typed by the user. Authors of BEVA have 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 to the search box, but this is not the most common case. Furthermore, their experiments have shown that even in these specific cases DFS BEVA was only slightly faster. Thus, we assume that BFS is adopted as the default strategy for searching if considering a user typing one character at a time.

On the other hand, given a certain dataset, the order by which keys are inserted in the trie determines the physical position of their nodes in memory, which in turn may impact the search performance. This impact occurs due to cache effects in the memory hierarchy. When building a trie, the more natural way of inserting keys is by creating all the nodes needed for representing a key as soon as this key is inserted. We call this approach DFS index building since nodes are inserted in an order that resembles the DFS.

The DFS approach has an important side effect: nodes in the same depth are inserted in a non-contiguous form, which may slow down the BFS query processing due to the cache effects in the memory hierarchy. This phenomenon is illustrated in Figure 20, where we compare how the trie nodes are created when inserting the keys using the DFS approach and using the BFS approach. In this figure, we can see that the more natural way of inserting keys in a trie, which resembles a DFS, tends to spread the nodes of equal depths in the trie along the memory used by the data structure. Although this behavior is obvious, it is usually not considered as 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 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 in the same depth may increase with the number of nodes inserted in the trie. As a result, nodes in the same depth may span different levels in the cache system and the BFS used for query processing is likely to yield a high rate of cache misses. Thus, 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 build index trees for error-tolerant prefix search which inserts all keys in parallel. We call this approach *BFS index building*, whose goal is to reduce the relative distance of nodes in the same depth in terms of memory allocation. In BFS we start by inserting the first character of all keys from the dataset, then insert the second character, and so on. As a result, the nodes at each depth become contiguous in the memory, thus creating a more cache-friendly data structure. By the end of the process, the position of the trie nodes in memory becomes sorted by their depths, as illustrated in the lowermost vector in Figure 20. We show

in the experiments section that this simple procedure has a great impact on the time performance of the query autocompletion task.

5.3 Experiments

In this section, we present the evaluation of the proposed optimizations in trie index building.

5.3.1 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 running BEVA method and using only the JusBrasil dataset since conclusions were similar when comparing other trie variations, datasets, and methods.

Table 25 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 most of our experiments through the chapters of this thesis, we 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 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 the 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.

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

Table 25 – Average prefix query processing and fetching times (ms) per query in the JusBrasil dataset when using BEVA, varying the full trie index version.

The fetching time values presented in Table 25 show 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 26, 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 items does not change when switching from one method to another.

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

Table 26 – Average fetching times (ms) per item retrieved in the JusBrasil dataset when using BEVA, varying the full trie index version.

To better understand the reasons for the difference in performance achieved by the BFS and DFS index building, we investigated the hypothesis of better using the cache memory system. The results confirmed our hypothesis. To illustrate this issue, in Table 27 we present the number of cache misses when processing queries in the JusBrasil dataset for $\tau = 3$. Data were obtained with the CacheGrind program[2]. When comparing the results using tries, we can see a reduction in the average number of cache misses at D1 from $3,657,308,594$ to $2,757,363,310$ and at DL from $3,211,465,633$ to $2,294,981,511$, 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.

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

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

Table 27 – Average cache miss per prefix query in the JusBrasil dataset for τ=3.

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 comparisons of results since conclusions were similar to the 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 other trie variations studied.

6 Applying Burst Tries for Error-Tolerant Prefix Search

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.

6.1 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, is 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 21 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.

Regarding the query processing costs, notice that MCD does not limit the maxi-

mum 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 the 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 to 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 an MCD tends to become close to the number processed when using the full trie.

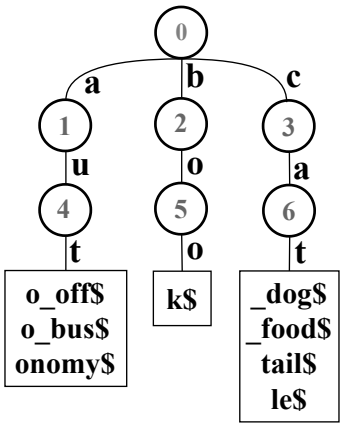


Figure 21 – Example of a burst trie with the minimum container depth (MCD) set to 3.

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 22 shows an example of a burst trie for the sample dataset when using the 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 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.

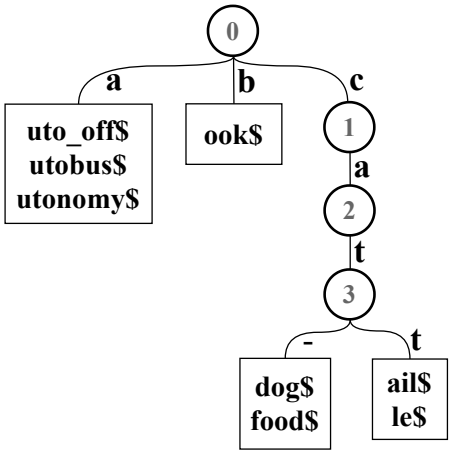


Figure 22 – Example of a burst trie limiting containers to 3 elements.

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 Figure 23.

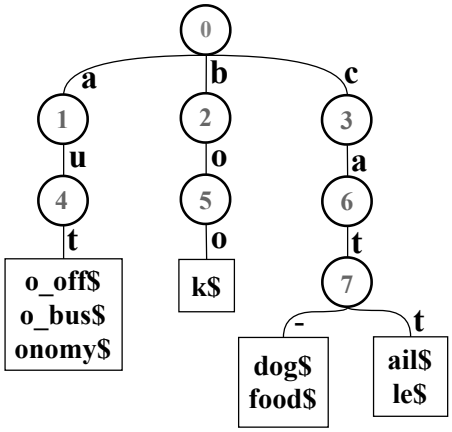


Figure 23 – Example of a burst trie with MCD set to 3 and MCK also set to 3.

6.2 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 a 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 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. 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 24 shows an example to illustrate the burst trie visualization presented in Figure 22. 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 a special leaf node that connects the container to the access trie, and the root node of a container represents the entire container in the burst trie.

When looking at Figure 24, 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 the root of a container, which is also an allocated node of the burst trie. When pointing to virtual nodes, 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 process a prefix “aut” using BEVA with an 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 the 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 the 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.

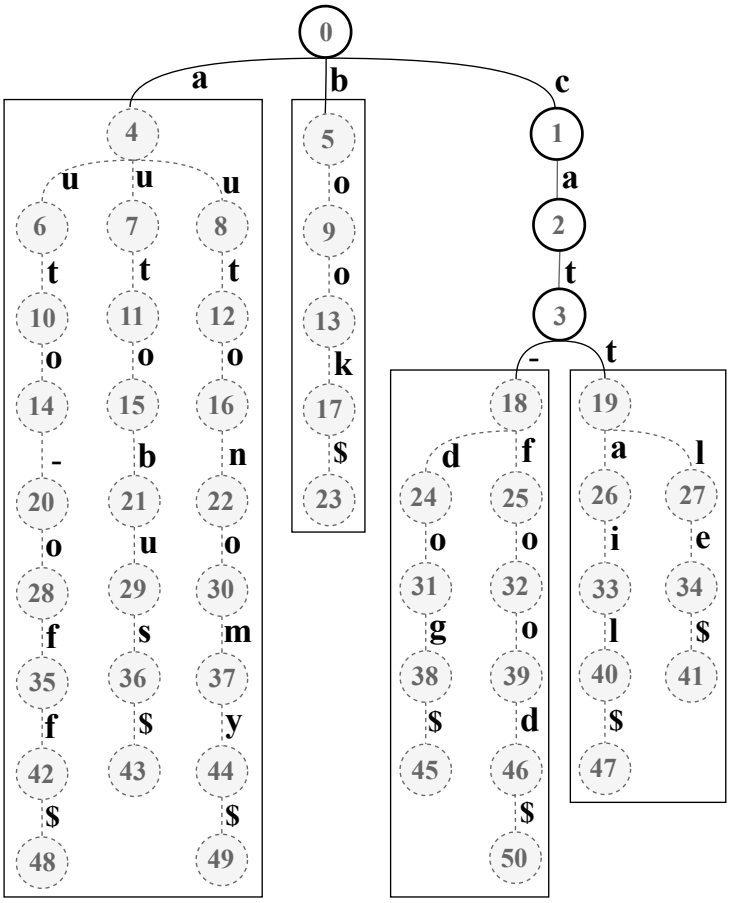


Figure 24 – Example of a burst trie limiting containers to 3 elements and representing the content in the containers as virtual trees.

6.2.1 Query processing in burst tries

In this section, we explain in more detail the query processing using burst tries adapted for ETQAC and the incremental computation of boundary active nodes. We use the edit vector automata to drive the traversal on the trie. Hence, a boundary active node of the prefix query is always associated with a state in the edit vector automata. Initially, the only boundary active node is the root node of the trie, associated with the initial state S_0.

Algorithm 3 Process prefix query p

```
1:  procedure MAINTAIN(c, |p|, B)
2:      updateBitmap(c)
3:      if |p| = 1 then
4:          B' ← ⟨r, S_0⟩                          ▷ r is the root of the trie
5:      else if |p| > τ then
6:          B' ← ∅
7:          for each ⟨n, S⟩ in B do
8:              if n.isVirtual then
9:                  B' ← B' ∪ findVirtualActiveNodes(|p|, ⟨n, S⟩)
10:             else
11:                 if n.isLeaf then
12:                     B' ← B' ∪ findVirtualActiveNodeFromActiveNodes(|p|, ⟨n, S⟩)
13:                 else
14:                     B' ← B' ∪ findActiveNodes(|p|, ⟨n, S⟩)
15:     return B'
```

The adaptation of the *maintain* function proposed in Zhou et al. (2016) is shown in the Algorithm 3. This function takes $c, |p|, \mathcal{B}$ as parameters, which represents the current character from prefix query, prefix query length, and list of boundary active nodes, respectively. Initially, in line 2, the global bitmaps are updated. When the prefix query length is 1, the only boundary active node is the root node associated with the initial state $\mathcal{S}_l$, as described in lines 3 and 4. The search for new boundary active nodes only starts when $|p| > \tau$, τ being the edit distance threshold, and this is checked on line 5. In line 6 the list of boundary active nodes is instantiated. In line 7, the iteration in the boundary active nodes of the previous prefix query is performed. If the current node is a virtual node (line 8) then the Algorithm 6 is called (line 9) to find virtual active nodes. Otherwise, if the current node is a leaf node (line 11) then the Algorithm 5 is called (line 12) to find virtual active nodes from the currently active node. Finally, the Algorithm 4 is called to traverse its descendants and search new boundary active nodes.

The Algorithm 4 is the same as presented in Zhou et al. (2016), changing only in lines 11 and 12, when the processing reaches a leaf node in the trie, then the Algorithm 5 is called because such leaf node does not have children nodes but this leaf node can have a suffix in the burst container. The answers are computed in the findVirtualActiveNodeFromActiveNodes and findActiveNodes algorithms independently but without duplicates.

In the Algorithm 5, the processing will be performed from the global static dataset determined by the range contained in the boundary active node of the previous prefix query, lines 3, 4, and 5. If we find any boundary active node in this step it will be processed in the next prefix query directly by the Algorithm 6 called from Algorithm 3.

The Algorithm 6 shows how the new virtual active nodes are computed. The processing is also performed incrementally, and the edit vector automata proposed by

Algorithm 4 Find active nodes

```
1: procedure FindActiveNodes(|p|, ⟨n, S⟩)
2:     B' ← ∅
3:     level ← n.level + 1
4:     k ← |p| − n.level
5:     for each child n' of n do
6:         b_n' ← buildBitmap(|p|, level, n'.char)
7:         S' ← f(S, b_n')
8:         if S' ≠ S_⊥ then                          ▷ S_⊥ is the final state
9:             if S'[τ + 1 + k] ≤ τ then
10:                B' ← B' ∪ ⟨n', S'⟩
11:            else if n'.isLeaf then
12:                B' ← B' ∪ findVirtualActiveNodeFromActiveNodes(|p|, ⟨n', S'⟩)
13:            else
14:                B' ← B' ∪ findActiveNodes(|p|, ⟨n', S'⟩)
15:    return B'
```

Algorithm 5 Find virtual active node from active nodes

```
1: procedure findVirtualActiveNodeFromActiveNodes(|p|, ⟨n, S⟩)
2:     B' ← ∅
3:     for each record in records[n.beginRange, n.endRange] do
4:         n' ← ⟨record.id, n.level⟩
5:         B' ← B' ∪ findVirtualActiveNodes(|p|, ⟨n', S⟩)
6:     return B'
```

Algorithm 6 Find virtual active nodes

```
1: procedure findVirtualActiveNodes(|p|, ⟨n, S⟩)
2:     B' ← ∅
3:     while n.level < |records[n.id]| do          ▷ records is the global static dataset
4:         n.level ← n.level + 1
5:         k ← |p| − n.level
6:         b_n' ← buildBitmap(|p|, n.level, records[n.id][level])
7:         S' ← f(S, b_n')
8:         if S' ≠ S_⊥ then                        ▷ S_⊥ is the final state
9:             if S'[τ + 1 + k] ≤ τ then
10:                n' ← ⟨n.id, n.level⟩
11:                B' ← B' ∪ ⟨n', S'⟩
12:                return B'
13:        else
14:            return B'
15:    return B'
```

Zhou et al. (2016) is used to calculate the edit distance between two strings. The processing is similar to the Algorithm 4, with the difference that we have a record from the static dataset instead of prefixes queries from trie. When the edit distance in the new state is within the τ, then a new virtual active node is computed as a response, and the processing terminates as described in lines 8 to 14. Otherwise, a new character must be checked.

As an example to show how to process queries that allow errors, Table 28 presents the nodes activated when searching for the prefix "cut". At the beginning, 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 of the dataset in their subtrees. After including the letter 't' to form the prefix "cut", nodes 10, 11, 12, and 3 become activated. This example is useful to illustrate how we adapt trie-based algorithms to perform search on burst tries. We can see that the search now activates more nodes since we add some redundancy to the tree when adding the virtual nodes. As we show in the experiments, such redundancy does not have much effect on the time for processing queries, while the usage of burst tries may significantly reduce the space required for storing the index.

Prefix Query	Active Node Set
∅	{0}
c	{0}
cu	{6, 7, 8, 1}
cut	{10, 11, 12, 3}

Table 28 – Query processing using the BEVA method with MCD+MCK and $\tau = 1$ to prefix query "cut" in a burst trie containing virtual nodes.

6.3 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.

6.3.1 Burst trie parameters selection

We start by discussing the parameter selection for burst tries and named 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 the trie-building strategies and analyze the relation 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 thus decided to only show results in JusBrasil to avoid too much redundant information, and also because it is the only real case query autocomplete 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 Figure 25. We report the variation in time for processing queries and memory usage as we vary the parameters.

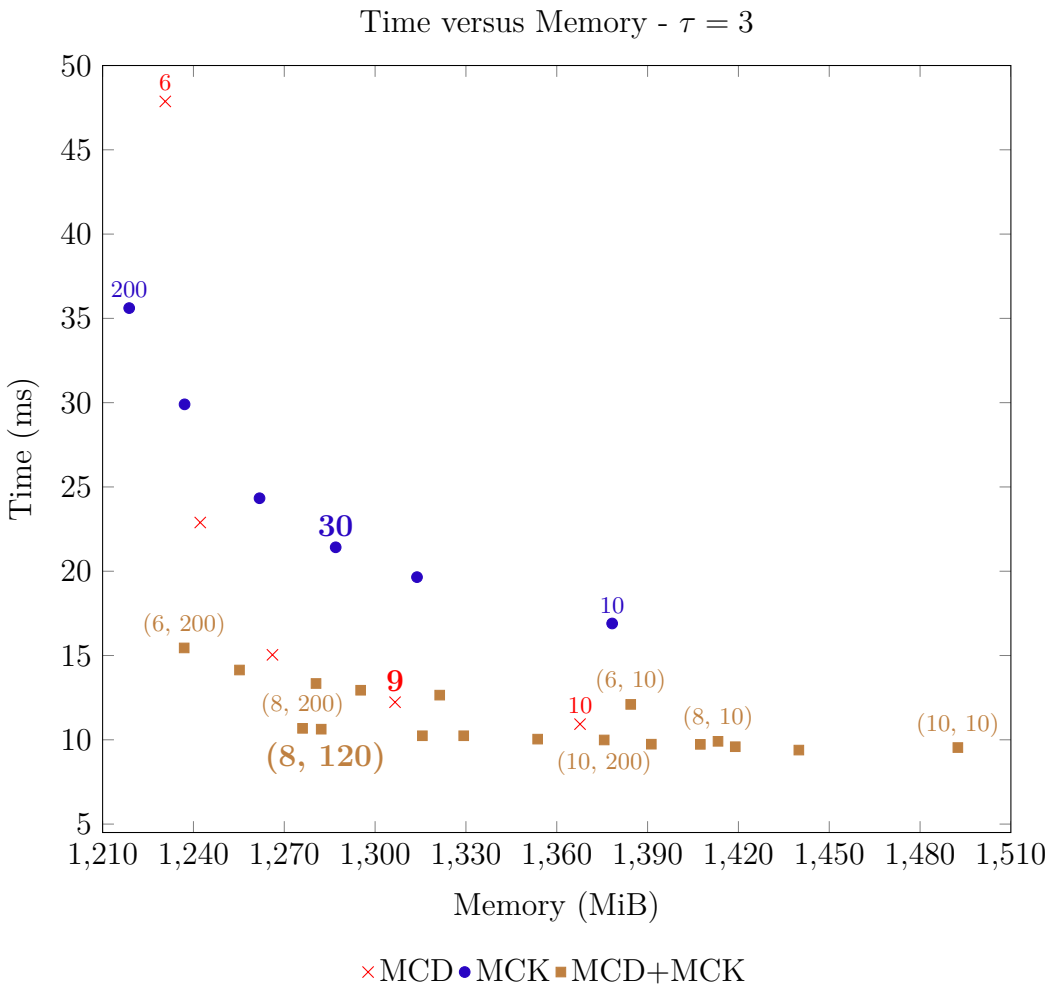


Figure 25 – Trade-off between time performance (ms) and memory usage (MiB) when processing queries with BEVA in JusBrasil dataset with data structures MCD with values varying from 6 to 10, MCK with values varying from 10 to 200 and MCD+MCK with MCD values 6, 8 and 10 and MCK varying also from 10 to 200.

Based on the results, we selected the parameters to be adopted in the experiments with each of the experimented variations for each dataset 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. 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 a 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 29.

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

Table 29 – Selected parameters for BEVA using MCD, MCK and MCD+MCK heuristic parameters in JusBrasil dataset.

Table 30 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 the 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.

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	1,282.13	0.2 ± 0.001	2.89 ± 0.015	21.84 ± 0.101
MCD+MCK	1,278.32	0.16 ± 0.001	1.65 ± 0.01	10.86 ± 0.066
dynamic-MCD	5,666.19	3.51 ± 0.119	23.88 ± 0.251	138.85 ± 0.837
dynamic-MCK	3,979.94	2.7 ± 0.013	29.43 ± 0.102	112.24 ± 0.392
dynamic-MCD+MCK	4,174.89	1.37 ± 0.009	21.99 ± 0.092	118.4 ± 0.419

Table 30 – 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.

6.3.2 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 MCD+MCK was adopted with the best parameters found in the previous section. Table 31 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.

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	1,590.41 ± 58.597

Table 31 – 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.

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. 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. In such cases, MCD+MCK processed queries in an average time of 10.86 milliseconds, while using full trie would result in a time of 9.34 milliseconds. These results mean MCD+MCK was 16.27% slower than the full trie. On the other hand, MCD+MCK used just about 26.0% of the memory requirement of full trie, reducing the memory requirement from 4,912.7 MiB to 1,278.32 MiB. Compared to CPT, MCD+MCK was not only faster, reducing the time from 18.82 milliseconds to 10.86 milliseconds, a reduction of about 42.29%, but also reduced the memory requirement from $1,820.1$ MiB to $1,278.32$ MiB, requiring only 70.23% of memory requirement required by CPT.

This positive result occurs because the BEVA algorithm traverses quite a minimized set of nodes and does not require maintaining such nodes after visiting them, which reduces 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 expressive and so 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 is not much variation among prefix query processing times when comparing the use of the full trie optimized for static collections and the burst trie variation also optimized for static collections, MCD+MCK, 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 adopted required that the suffix array pointed to each suffix of each query suggestion present in the dataset, thus making the suffix array demand a large amount of space in memory.

6.3.2.1 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). This experiment was performed using the JusBrasil dataset. Figure 26 presents the results of experiments, varying the prefix 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 performance of the methods 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 on the time performance when compared to the full trie. The good performance of the methods even for larger prefixes

occurs because we expect fewer additions 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.

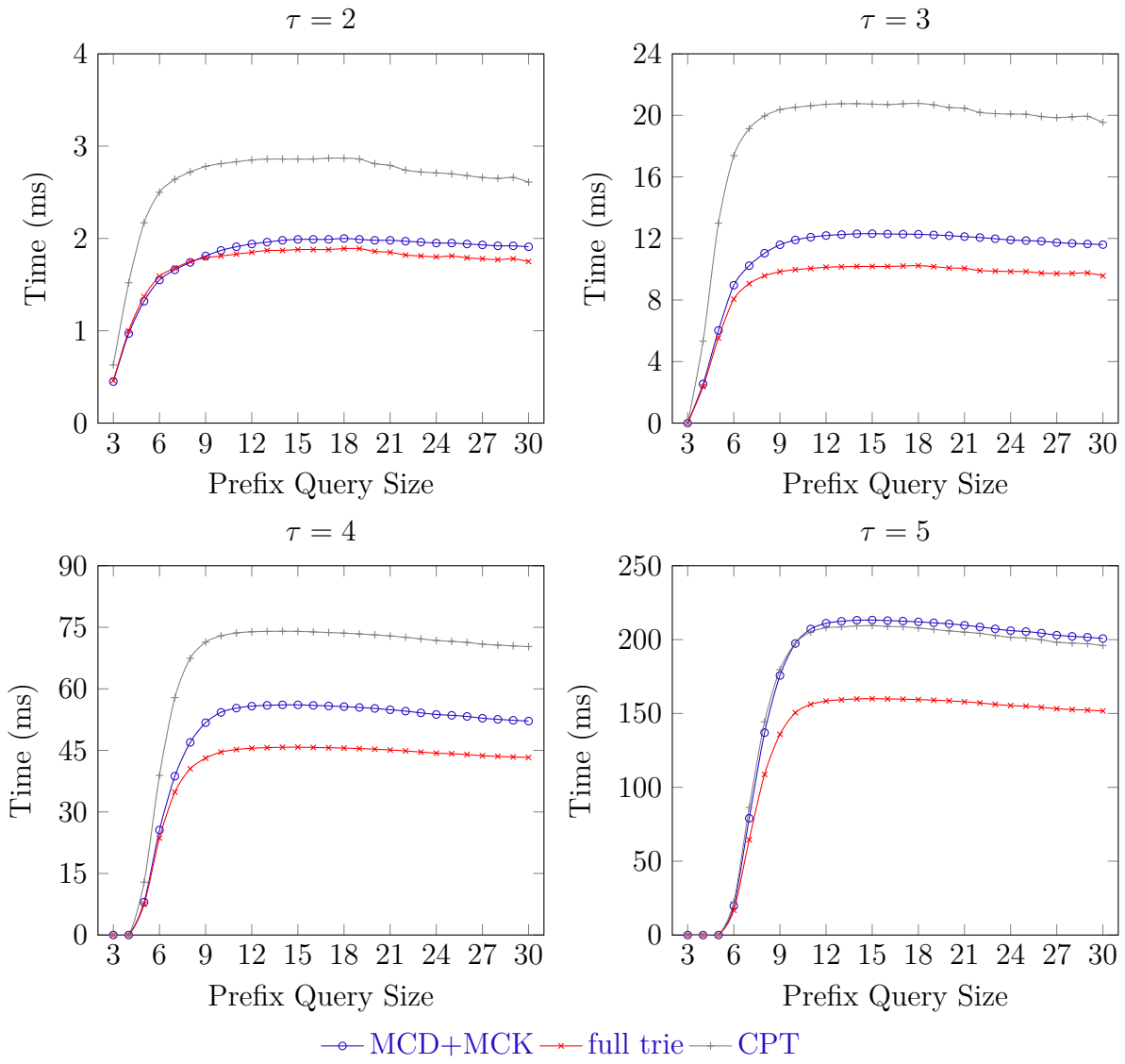


Figure 26 – Time performance (ms) of BEVA using MCD+MCK (set to 8 and 120, respectively), and full trie when processing prefix queries and varying the prefix query size and the number of errors allowed in JusBrasil dataset

As it 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 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 the 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 CTP 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 conclusions are the same, with the time performance of MCD+MCK and full trie being close in all experimented parameters and collections.

6.3.2.2 How do methods affect scalability?

A good question is whether the usage of burst static affects the throughput of systems to deal with high query workloads or not. We performed experiments with the methods submitting queries using Vegeta[1].

Figure 27 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, thus this restriction makes the experiment closer to real scenarios.

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 Figure 27, 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.

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

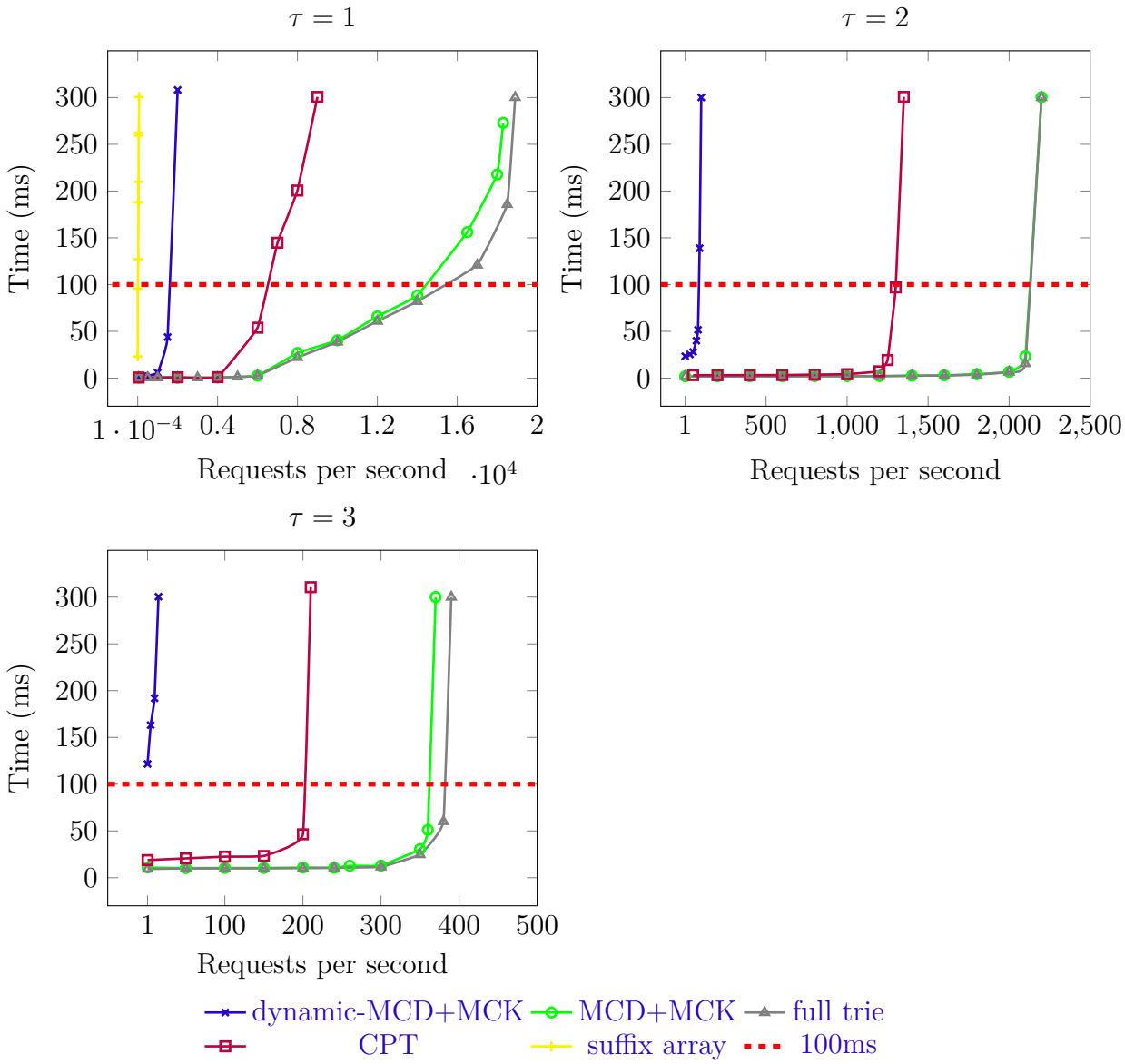


Figure 27 – Processing time (ms) as the number of requests per second increases for τ varying from 1 to 3 in the JusBrasil dataset.

6.3.2.3 Performance when increasing the size of the dataset

An important evaluation in the experiments is to verify the behavior of the proposed ideas for a dataset that frequently increases the size over time, a common scenario in real practical applications that have a large volume of logs being generated per day. Therefore, we evaluate our ideas in different portions of the same dataset, simulating the increase in size of this dataset over time. For this, we split the adopted datasets into portions of sizes of 20%, 40%, 60%, and 100% of the total size. These portions were created getting the items from the full dataset in a random order until reaching the portion limit.

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. Figure 28 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.

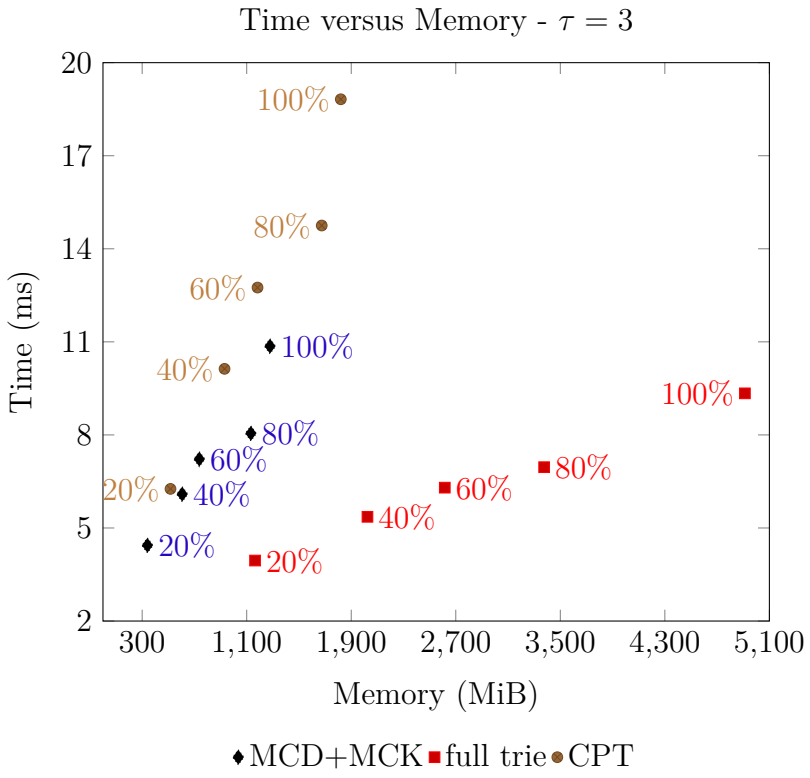


Figure 28 – Time performance (ms) and memory usage (MiB) when processing queries with BEVA and indexing increasing percentages of the JusBrasil dataset (20%, 40%, 60%, 80%, 100%) with data structures MCD+MCK (set to 8 and 120, respectively), full trie and CPT.

6.3.3 Comparing trie indexes in mode word by word

We have presented experiments up to now considering a scenario where the system performs matches between the whole prefix typed by the user and the complete string of each query suggestion, called modes 1 and 2 in the taxonomy presented by Krishnan et al. (2017a), but here we consider it allowing errors. Another possible usage is a scenario

where the match is performed word by word, which is mode 3 of the taxonomy, but here also includes 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 split into words. The first step is to compute the match between the words in the prefix typed by the user and the words 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. In the experiments in this environment, we present only the performance of the first step, since we are using data structures to perform prefix match operations.

Table 32 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, thus 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. The 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.

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.

6.3.4 Experiments with DBLP and UMBC datasets

Tables 34 and 35 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 33 presents the parameters chosen for the MCD+MCK on these two datasets. The 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

Methods	MEM (MiB)	Time (ms)		
		$\tau = 1$	$\tau = 2$	$\tau = 3$
MCD+MCK	54.43	0.08 ± 0.0	0.91 ± 0.004	4.85 ± 0.029
full trie	196.42	0.09 ± 0.002	0.95 ± 0.016	5.06 ± 0.101
CPT	53.86	0.13 ± 0.002	1.24 ± 0.022	7.05 ± 0.137
suffix array	186.29	1.9 ± 0.107	17.08 ± 0.483	46.74 ± 0.971

Table 32 – 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.

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

Table 33 – Selected parameters for BEVA using MCD+MCK in DBLP and UMBC datasets.

performance of the methods is compatible with the conclusions for experiments with the JusBrasil dataset. Again MCD+MCK presents a better 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 at 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, typed by the user.

Methods	MEM (MiB)	Time (ms) $\tau = 1$		$\tau = 2$		$\tau = 3$	
		9	17	9	17	9	17
MCD+MCK	505.63	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 34 – 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	18,194.58	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	19,909.1	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	-	-	-	-	-	-	-

Table 35 – 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.

7 Conclusion

This research aims to address the development of efficient error-tolerant query autocompletion systems. We have studied efficient data structures to represent the query suggestion dataset and perform fast error-tolerant prefix search operations on it.

We 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.

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 can 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 experiments, even when using parameters that resulted in less memory usage than CPTs. A general conclusion is that 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.

Regarding the algorithms for performing error-tolerant ETQAC, we proposed a method called BWBEV, which uses a bit parallelism approach. The experiments presented in this study have demonstrated that the *BWBEV* method is a new competitive method for performing approximate prefix search on large datasets. *BWBEV* was designed for scenarios where the dataset can be indexed to facilitate rapid prefix search. Our experiments compared the performance of BWBEV to state-of-the-art approximate prefix search methods, and the results indicated that *BWBEV* outperformed these methods across all tested scenarios. These findings suggest that *BWBEV* can enable the development of faster and more scalable search systems.

Also regarding efficient algorithms, we have proposed two pruning strategies for pruning nodes while processing the queries that can potentially speed up the query processing further.

7.1 Future works

Several avenues for future research can build upon the findings of this study:

1. **Exploring Different Match Modes** Building on the ideas of (Krishnan et al., 2017b), we plan to study the combination of various match modes and their impact on both query processing efficiency and the quality of query suggestions. This includes experiments with partial matches, prefix matches, and error-tolerant combinations to balance speed and result relevance.

2. **Learning-to-Rank (LTR) Models for Top-k Ranking** We intend to investigate top-k ranking functions using Learning-to-Rank models. LTR approaches could refine the quality of query suggestions by incorporating user interaction data and optimizing the ranking of suggestions based on relevance and click-through probabilities.

3. **Integration of Semantic Search Techniques** A promising direction is the integration of semantic search methods using embeddings to enhance prefix search. By combining trie-based approaches with embeddings, systems could handle misspellings, synonyms, and semantic variations more effectively.

4. **Parallel and Distributed Implementations** Future research could explore parallel or distributed implementations of BWBEV and other trie-based algorithms to scale ETQAC systems for massive datasets. Leveraging modern distributed computing frameworks such as Spark or cloud-native architectures could enable near real-time query processing.

5. **Dynamic Dataset Updates** Investigating efficient methods for dynamically updating the index without full reprocessing is another important step. This would allow autocompletion systems to adapt to evolving datasets in real time while preserving performance.

References

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004. Cited on page 30.

Anurag Acharya, Huican Zhu, and Kai Shen. Adaptive algorithms for cache-efficient trie search. In *Workshop on Algorithm Engineering and Experimentation*, pages 300–315. Springer, 1999. Cited on page 31.

Marwah Alaofi, Luke Gallagher, Dana Mckay, Lauren L. Saling, Mark Sanderson, Falk Scholer, Damiano Spina, and Ryen W. White. 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, 2022. Association for Computing Machinery. ISBN 9781450387323. doi: 10.1145/3477495.3531711. URL https://doi.org/10.1145/3477495.3531711. Cited on page 18.

Nikolas Askitis and Ranjan Sinha. Hat-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the 13th Australasian conference on Computer Science*, pages 97–105, 2007. URL http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV62Askitis.html. Cited on page 31.

Nikolas Askitis and Justin Zobel. Redesigning the string hash table, burst trie, and bst to exploit cache. *Journal of Experimental Algorithmics (JEA)*, 15:1–1, 2011. Cited on page 31.

Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, oct 1992. ISSN 0001-0782. doi: 10.1145/135239.135243. URL https://doi.org/10.1145/135239.135243. Cited 2 times on pages 48 and 68.

Ziv Bar-Yossef and Naama Kraus. 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, 2011a. Association for Computing Machinery. ISBN 9781450306324. doi: 10.1145/1963405.1963424. URL https://doi.org/10.1145/1963405.1963424. Cited on page 73.

Ziv Bar-Yossef and Naama Kraus. 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, 2011b. Association for Computing Machinery. ISBN 9781450306324. doi: 10.1145/1963405.1963424. URL https://doi.org/10.1145/1963405.1963424. Cited on page 46.

Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer, and Niklas Schnelle. Efficient sparql autocompletion via sparql. *arXiv preprint arXiv:2104.14595*, 2021. Cited on page 37.

Holger Bast and Ingmar Weber. 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, 2006. ISBN 1595933697. URL https://doi.org/10.1145/1148170.1148234. Cited on page 37.

Holger Bast, Christian W Mortensen, and Ingmar Weber. Output-sensitive autocompletion search. *Information Retrieval*, 11(4):269–286, 2008. URL https://doi.org/10.1007/s10791-008-9048-x. Cited on page 37.

Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *International Symposium on String Processing and Information Retrieval*, pages 159–172. Springer, 2010. Cited on page 32.

Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, page 165–173. Springer-Verlag, 2002. ISBN 3540441808. URL http://arxiv.org/abs/cs/0211010. Cited on page 32.

Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the International ACM SIGMOD Conference on Management of Data*, pages 521–534, 2018. URL https://doi.org/10.1145/3183713.3196896. Cited on page 32.

Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, page 426–434, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137230. doi: 10.1145/956863.956944. URL https://doi.org/10.1145/956863.956944. Cited on page 23.

Fei Cai and Maarten de Rijke. Learning from homologous queries and semantically related terms for query auto completion. *Information Processing & Management*, 52(4):628–643, 2016. ISSN 0306-4573. URL https://doi.org/10.1016/j.ipm.2015.12.008. Cited on page 46.

Fei Cai, Shangsong Liang, and Maarten de Rijke. Time-sensitive personalized query autocompletion. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, page 1599–1608, New York,

NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325981. doi: 10.1145/2661829.2661921. URL https://doi.org/10.1145/2661829.2661921. Cited on page 46.

Fei Cai, Ridho Reinanda, and Maarten De Rijke. Diversifying query auto-completion. *ACM Trans. Inf. Syst.*, 34(4), jun 2016. ISSN 1046-8188. doi: 10.1145/2910579. URL https://doi.org/10.1145/2910579. Cited on page 21.

Surajit Chaudhuri and Raghav Kaushik. Extending autocompletion to tolerate errors. In *ACM SIGMOD*, pages 707–718. Association for Computing Machinery, Inc., June 2009. URL https://doi.org/10.1145/1559845.1559919. Cited 8 times on pages 19, 21, 23, 36, 37, 40, 41, and 56.

David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1998. UMI Order No. GAXNQ-21335. Cited on page 29.

Silviu Cucerzan and Eric Brill. Spelling correction as an iterative process that exploits the collective knowledge of web users. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 293–300. Association for Computational Linguistics, July 2004. Cited on page 18.

Caio Moura Daoud, Edleno Silva Moura, David Fernandes, Altigran Soares Silva, Cristian Rossi, and Andre Carvalho. Waves: A fast multi-tier top-k query processing algorithm. *Inf. Retr.*, 20(3):292–316, jun 2017. ISSN 1386-4564. doi: 10.1007/s10791-017-9298-6. URL https://doi.org/10.1007/s10791-017-9298-6. Cited on page 23.

John J Darragh, John G Cleary, and Ian H Witten. Bonsai: a compact representation of trees. *Software: Practice and Experience*, 23(3):277–291, 1993. Cited on page 31.

Dong Deng, Guoliang Li, He Wen, H. V. Jagadish, and Jianhua Feng. Meta: An efficient matching-based method for error-tolerant autocompletion. *Proc. VLDB Endow.*, 9(10): 828–839, June 2016. ISSN 2150-8097. URL https://doi.org/10.14778/2977797.2977808. Cited 7 times on pages 19, 23, 36, 37, 38, 39, and 44.

Giovanni Di Santo, Richard McCreadie, Craig Macdonald, and Iadh Ounis. Comparing approaches for query autocompletion. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, page 775–778, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336215. doi: 10.1145/2766462.2767829. URL https://doi.org/10.1145/2766462.2767829. Cited 2 times on pages 46 and 47.

Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, page 993–1002, New York,

NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307574. doi: 10.1145/2009916.2010048. URL https://doi.org/10.1145/2009916.2010048. Cited on page 23.

Branislav Durian, Jan Holub, Hannu Peltola, and Jorma Tarhio. Tuning bndm with q-grams. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, page 29–37, USA, 2009. Society for Industrial and Applied Mathematics. Cited on page 68.

Berg Ferreira, Edleno Silva de Moura, and Altigran da Silva. Applying burst-tries for error-tolerant prefix search. *Inf. Retr.*, 25(4):481–518, dec 2022. ISSN 1386-4564. doi: 10.1007/s10791-022-09416-9. URL https://doi.org/10.1007/s10791-022-09416-9. Cited on page 53.

Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960. ISSN 0001-0782. URL https://doi.org/10.1145/367390.367400. Cited 3 times on pages 22, 31, and 36.

Chavoosh Ghasemi, Hamed Yousefi, Kang G Shin, and Beichuan Zhang. A fast and memory-efficient trie structure for name-based packet forwarding. In *Proceedings of the International Conference on Network Protocols*, pages 302–312, 2018. URL https://doi.org/10.1109/ICNP.2018.00046. Cited on page 32.

Simon Gog, Giulio Ermanno Pibiri, and Rossano Venturini. Efficient and effective query auto-completion. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2271–2280, 2020. Cited 2 times on pages 26 and 80.

Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. *Information Retrieval: Data Structures & Algorithms*, 66:82, 1992. Cited on page 30.

Korinna Grabski and Tobias Scheffer. 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, 2004. ISBN 1581138814. URL https://doi.org/10.1145/1008992.1009066. Cited on page 37.

Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, April 2002. ISSN 1046-8188. Cited 3 times on pages 22, 28, and 87.

Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*,

11(1):1–9, 2016. URL https://doi.org/10.1186/s13015-016-0066-8. Cited on page 32.

Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. An efficient algorithm for location-aware query autocompletion. *IEICE TRANSACTIONS on Information and Systems*, 101(1): 181–192, 2018. doi: 10.1587/transinf.2017EDP7152. Cited 2 times on pages 44 and 46.

Heikki Hyyrö. A bit-vector algorithm for computing levenshtein and damerau edit distances. In *Nordic Journal of Computing*, 2003. doi: 10.5555/846090.846095. URL https://api.semanticscholar.org/CorpusID:12608967. Cited on page 51.

Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Linked dynamic tries with applications to lz-compression in sublinear time and space. *Algorithmica*, 71(4):969–988, 2015. Cited on page 32.

Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, page 371–380, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584874. doi: 10.1145/1526709.1526760. URL https://doi.org/10.1145/1526709.1526760. Cited 10 times on pages 18, 19, 23, 36, 37, 38, 41, 44, 56, and 57.

Jyun-Yu Jiang, Yen-Yu Ke, Pao-Yu Chien, and Pu-Jen Cheng. Learning user reformulation behavior for query auto-completion. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, page 445–454, New York, NY, USA, 2014a. Association for Computing Machinery. ISBN 9781450322577. doi: 10.1145/2600428.2609614. URL https://doi.org/10.1145/2600428.2609614. Cited on page 21.

Jyun-Yu Jiang, Yen-Yu Ke, Pao-Yu Chien, and Pu-Jen Cheng. Learning user reformulation behavior for query auto-completion. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, page 445–454, New York, NY, USA, 2014b. Association for Computing Machinery. ISBN 9781450322577. doi: 10.1145/2600428.2609614. URL https://doi.org/10.1145/2600428.2609614. Cited on page 46.

Shunsuke Kanda, Dominik Köppl, Yasuo Tabei, Kazuhiro Morita, and Masao Fuketa. Dynamic path-decomposed tries. *Journal of Experimental Algorithmics (JEA)*, 25:1–28, 2020. Cited on page 32.

Young Mo Kang, Wenhao Liu, and Yingbo Zhou. Queryblazer: Efficient query autocompletion framework. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, WSDM '21, page 1020–1028. Association for Computing

Machinery, 2021. ISBN 9781450382977. URL https://doi.org/10.1145/3397271.3401432. Cited on page 45.

Unni Krishnan, Alistair Moffat, and Justin Zobel. A taxonomy of query auto completion modes. In *Proceedings of the 22nd Australasian Document Computing Symposium*, ADCS 2017, New York, NY, USA, 2017a. Association for Computing Machinery. ISBN 9781450363914. doi: 10.1145/3166072.3166081. URL https://doi.org/10.1145/3166072.3166081. Cited on page 101.

Unni Krishnan, Alistair Moffat, and Justin Zobel. A taxonomy of query auto completion modes. In *Proceedings of the 22nd Australasian Document Computing Symposium*, ADCS 2017, New York, NY, USA, 2017b. Association for Computing Machinery. ISBN 9781450363914. doi: 10.1145/3166072.3166081. URL https://doi.org/10.1145/3166072.3166081. Cited 2 times on pages 25 and 106.

VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966. Cited 2 times on pages 32 and 47.

Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20:617–640, 08 2011. URL https://doi.org/10.1007/s00778-011-0218-x. Cited 11 times on pages 19, 23, 36, 37, 38, 41, 44, 46, 56, 57, and 81.

Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993. Cited on page 30.

Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. URL https://doi.org/10.1145/321941.321946. Cited on page 29.

Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall Joint Computer Conference, part I*, pages 267–277, 1968. Cited 2 times on pages 61 and 99.

Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968. Cited on page 31.

Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, may 1999. ISSN 0004-5411. doi: 10.1145/316542.316550. URL https://doi.org/10.1145/316542.316550. Cited on page 51.

Arnab Nandi and H. V. Jagadish. Effective phrase prediction. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '07, page 219–230, 2007.

ISBN 9781595936493. URL http://www.vldb.org/conf/2007/papers/research/p219-nandi.pdf. Cited on page 37.

Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, mar 2001. ISSN 0360-0300. doi: 10.1145/375360.375365. URL https://doi.org/10.1145/375360.375365. Cited on page 48.

Gonzalo Navarro and Mathieu Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. Exp. Algorithmics*, 5:4–es, dec 2001. ISSN 1084-6654. doi: 10.1145/351827.384246. URL https://doi.org/10.1145/351827.384246. Cited on page 68.

Gonzalo Navarro, Erkki Sutinen, Jani Tanninen, and Jorma Tarhio. Indexing text with approximate q-grams. In *Annual Symposium on Combinatorial Pattern Matching*, pages 350–363. Springer, 2000. Cited 2 times on pages 30 and 58.

Gonzalo Navarro, Erkki Sutinen, and Jorma Tarhio. Indexing text with approximate q-grams. *Journal of Discrete Algorithms*, 3(2-4):157–175, 2005. Cited 2 times on pages 30 and 58.

Hannu Peltola and Jorma Tarhio. Alternative algorithms for bit-parallel string matching. In Mario A. Nascimento, Edleno S. de Moura, and Arlindo L. Oliveira, editors, *String Processing and Information Retrieval*, pages 80–93, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39984-1. Cited on page 68.

Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 615–624, 2017. Cited on page 80.

Jianbin Qin, Chuan Xiao, Sheng Hu, Jie Zhang, Wei Wang, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane. Efficient query autocompletion with edit distance-based error tolerance. *VLDB Journal*, pages 1–25, 2019. URL https://doi.org/10.14778/2536336.2536339. Cited 6 times on pages 19, 21, 36, 37, 40, and 56.

Milad Shokouhi. Learning to personalize query auto-completion. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, page 103–112, New York, NY, USA, 2013a. Association for Computing Machinery. ISBN 9781450320344. doi: 10.1145/2484028.2484076. URL https://doi.org/10.1145/2484028.2484076. Cited on page 21.

Milad Shokouhi. Learning to personalize query auto-completion. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, page 103–112, New York, NY, USA, 2013b. Association for

Computing Machinery. ISBN 9781450320344. doi: 10.1145/2484028.2484076. URL https://doi.org/10.1145/2484028.2484076. Cited on page 46.

Milad Shokouhi and Kira Radinsky. Time-sensitive query auto-completion. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, page 601–610, New York, NY, USA, 2012a. Association for Computing Machinery. ISBN 9781450314725. doi: 10.1145/2348283.2348364. URL https://doi.org/10.1145/2348283.2348364. Cited on page 21.

Milad Shokouhi and Kira Radinsky. Time-sensitive query auto-completion. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, page 601–610, New York, NY, USA, 2012b. Association for Computing Machinery. ISBN 9781450314725. doi: 10.1145/2348283.2348364. URL https://doi.org/10.1145/2348283.2348364. Cited 2 times on pages 46 and 47.

Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000. Cited on page 68.

Catherine L. Smith, Jacek Gwizdka, and Henry Feild. The use of query auto-completion over the course of search sessions with multifaceted information needs. *Information Processing & Management*, 53(5):1139–1155, 2017. ISSN 0306-4573. URL https://doi.org/10.1016/j.ipm.2017.05.001. Cited on page 45.

Alisa Strizhevskaya, Alexey Baytin, Irina Galinskaya, and Pavel Serdyukov. Actualization of query suggestions using query logs. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, page 611–612, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312301. doi: 10.1145/2187980.2188152. URL https://doi.org/10.1145/2187980.2188152. Cited on page 47.

Saedeh Tahery and Saeed Farzi. Customized query auto-completion and suggestion — a review. *Information Systems*, 87:101415, 2020. URL https://www.sciencedirect.com/science/article/pii/S0306437919303072. Cited on page 45.

Esko Ukkonen and Derick Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, November 1993. ISSN 0178-4617. URL https://doi.org/10.1007/BF01769703. Cited 3 times on pages 33, 52, and 76.

Jin Wang and Chunbin Lin. Fast error-tolerant location-aware query autocompletion. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1998–2001. IEEE, 2020. URL https://doi.org/10.1109/ICDE48307.2020.00223. Cited 2 times on pages 44 and 46.

Stewart Whiting and Joemon M. Jose. Recent and robust query auto-completion. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, page 971–982, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327442. doi: 10.1145/2566486.2568009. URL https://doi.org/10.1145/2566486.2568009. Cited on page 46.

Alden H. Wright. Approximate string matching using within-word parallelism. *Softw. Pract. Exper.*, 24(4):337–362, apr 1994. ISSN 0038-0644. doi: 10.1002/spe.4380240402. URL https://doi.org/10.1002/spe.4380240402. Cited on page 51.

Sun Wu and Udi Manber. Fast text searching: Allowing errors. *Commun. ACM*, 35 (10):83–91, oct 1992. ISSN 0001-0782. doi: 10.1145/135239.135244. URL https://doi.org/10.1145/135239.135244. Cited on page 49.

Chuan Xiao, Jianbin Qin, Wei Wang, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane. Efficient error-tolerant query autocompletion. *Proc. VLDB Endow.*, 6(6): 373–384, April 2013. ISSN 2150-8097. URL https://doi.org/10.14778/2536336.2536339. Cited 4 times on pages 19, 37, 39, and 56.

Gaogang Xie, Jingxiu Su, Xin Wang, Taihua He, Guangxing Zhang, Steve Uhlig, and Kave Salamatian. Index–trie: Efficient archival and retrieval of network traffic. *Computer Networks*, 124:140–156, 2017. URL https://doi.org/10.1016/j.comnet.2017.06.010. Cited on page 32.

Susumu Yata. Dictionary compression by nesting prefix/patricia trie. In *Proceedings of the 17th Annual Meeting of the Association for Natural Language*, 2011. Cited on page 31.

Qi Zhong, Jinyi Zhi, and Gang Guo. Dynamic is optimal: Effect of three alternative autocomplete on the usability of in-vehicle dialing displays and driver distraction. *Traffic injury prevention*, 23(1):51–56, 2022. doi: 10.1080/15389588.2021.2010052. Cited on page 18.

Xiaoling Zhou, Jianbin Qin, Chuan Xiao, Wei Wang, Xuemin Lin, and Yoshiharu Ishikawa. Beva: An efficient query processing algorithm for error-tolerant autocompletion. *ACM Trans. Database Syst.*, 41(1), March 2016. ISSN 0362-5915. URL https://doi.org/10.1145/2877201. Cited 21 times on pages 19, 22, 23, 33, 34, 35, 36, 37, 40, 41, 42, 44, 56, 57, 58, 67, 75, 76, 81, 91, and 93.

Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977. Cited on page 32.