Van Den Berg da Gama Ferreira

Optimizing BEVA with Two-Level Indexes

Manaus, AM - Brazil

2020

Van Den Berg da Gama Ferreira

Optimizing BEVA with Two-Level Indexes

Dissertation presented to the Program of Post-graduate in Informatics of the Federal University of Amazonas - UFAM as a requirement to obtain the degree of Masters in Informatics.

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

2020

Ficha Catalográfica

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

F383b

Ferreira, Van Den Berg da Gama
   Optimizing BEVA with Two-Level Indexes / Van Den Berg da Gama Ferreira . 2020
   87 f.: il. color; 31 cm.

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

   1. query processing. 2. error-tolerant. 3. autocompletion,. 4. two-level. 5. trie. I. Moura, Edleno Silva de. II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"Optmizing BEVA with two-level indexes"

VAN DEN BERG DA GAMA FERREIRA

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Edleno Silva de Moura
Prof. Edleno Silva de Moura - PRESIDENTE

[illegible]
Prof. Altigran Soares da Silva - MEMBRO INTERNO

[illegible]
Prof. Thierson Couto Rosa - MEMBRO EXTERNO

Manaus, 10 de Julho de 2020

This work is dedicated to my parents, brave warriors of the countryside, who when children could not have the opportunity to walk the path of studies, but knew the importance of the study and gave me full support to walk this path from an early age.

Acknowledgements

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

MY FAMILY, especially my wife and unconditional companion and my daughter Anallu, for being my 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, for also providing me with great challenges allowing me to evolve as a person and researcher. My thanks.

TO THE BDRI STUDENT GROUP, in the person of Anderson Pimentel, Lucas Castro and Felipe Franco, for the exchange of knowledge. Especially the availability of Anderson Pimentel to help in the start this journey. That was very important.

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.

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

Query autocompletion is an important component of modern search systems that suggests possible queries at each user keystroke to complete the query based on the prefix already typed in the search box. One of the most adopted and successful data structures for query autocompletion is the TRIE which is used to index the possible query suggestions. The TRIE is traversed based on the search prefix typed by the user in order to select suggestions that match the prefix. The use of TRIEs requires a large amount of extra memory for processing queries, which may increase the cost for processing queries and may limit the number of query suggestions indexed. In this work we propose optimized alternative implementations of BEVA algorithm, currently the state-of-the-art in the literature for autocompletion, in order to achieve a reduction in its memory consumption while keeping it efficient in query processing times.

First, we propose a novel strategy to build the TRIE, named level-at-a-time (laat), and compare its performance to the way TRIEs are usually built, the key-at-a-time (kaat). In the kaat strategy the index is built in depth-ward direction and in the laat strategy the index is built in breadth-ward direction. We implemented the proposed ideas and experimented them with several datasets, where we show that laat strategy allows a significant speedup in query processing of BEVA, being up to four times faster, with improvements achieved specially in queries with high number of errors, which are the most expensive ones.

Second, we study the use of two-level indexing and prefix processing approaches for query autocompletion also in BEVA method. Two-level approaches combine the use of indexes with sequential search in order to reduce memory requirements in search systems. In our study, we insert in the TRIE only part of each query inserted, and the leaf nodes reference to a set of queries where a sequential search is performed. We experimented two alternative ways of selecting the portion of each key that remains indexed in the first level and compare their performance. The two-level approach has shown to significantly reduce the memory requirements for storing the index with just a small variation in query processing times.

Keywords: query processing, error-tolerant, autocompletion, two-level, TRIE.

List of Figures

Figure 1 – Query autocompletion in search box. 23

Figure 2 – Representation of the TRIE data structure with the exact search and approximate search for the prefix query "undo". Nodes in blue color represent the nodes that exact search processing traverses. The green nodes represent the nodes that are traversed when the search is approximate and in this case the blue nodes can also be part. For query "undo" and $\tau = 1$. 37

Figure 3 – Computing the active nodes for $\mathcal{P}_\mathcal{Q}$ = "live" and $\tau = 1$. The strings "life", "live" and "love" are similar to $\mathcal{P}_\mathcal{Q}$. 38

Figure 4 – Representation of boundary active nodes (in blue) obtained step by step for the query "live" and $\tau = 1$. 40

Figure 5 – Difference between obtaining active nodes and boundary active nodes when changing the prefix query from "li" to "liv" 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. 41

Figure 6 – Representation of the boundary active nodes set computation as the prefix query changes. For this example we have the query "love", $\tau = 1$ and the prefixes indexed in the TRIE are "life", "live" and "love". For this example, although the edit vectors are represented in a different structure from the TRIE, the edit vectors are manipulated in the TRIE nodes. 43

Figure 7 – Edit vector automata for $\tau = 1$. 45

Figure 8 – Ways of building a TRIE index. 48

Figure 9 – Example of the indexing with updates in the intervals. 52

Figure 10 – Indexing with fixed size branches 4. 52

Figure 11 – Example of the indexing construction in LS method with $max_{slot} = 2$. . 55

Figure 12 – Indexing with Limited Slot Method. 55

Figure 13 – Architecture of our proposed methods using the two-level approach. On the left side we have our first level with part of the strings indexed in the TRIE, the size of the string to be indexed depends of the configured parameter. Each node in the tree has an interval referring to the complete strings that this node covers. When we reach a given node, we take its range and directly access the complete string in the list of strings. In this index we run the BEVA with adaptations. On the right side of the figure is our list of complete strings in lexicographic order, in which we obtain the complete string from the first level. We run the also the second level in this list of complete strings, but only in subsets covered by active nodes or covered by active word nodes. At the bottom of the figure we have the EVA structure that allows the efficient the edit distance calculation between two strings by storing states associated with the active nodes. Both levels use this structure. 57
Figure 14 – Representation of query processing using the LS method. Consider the prefix query “midd”. The boundary active prefix set is $\{n_4\}$. 60
Figure 15 – Amount of memory used (MB) for LD and LS methods in the USADDR dataset with 25%, 50%, 75% and 100% of the dataset portion and $\tau = 1$ 79
Figure 16 – Memory Consumption (MB) with portion of 25%, 50%, 75% and 100% of the dataset sizes in USADDR, MEDLINE and DBLP. 83

List of Tables

Table 1 – Edit distance calculation with classic dynamic programming. 34

Table 2 – The cell values in color grey from top to bottom represent the k-diagonal varying from -1 to 1 for $\tau = 1$. The vertical in grey dark is the 0-diagonal. 35

Table 3 – Edit vectors represented in yellow and green. 42

Table 4 – Query processing step by step with BEVA and prefix query “live”. . . . 46

Table 5 – Query processing with LS method for prefix query “midday”. The boundary active nodes set obtained in the first level, shown in step 1 to 4. And the word active node set obtained in the second level, shown in step 5 to 7. 61

Table 6 – Dataset Statistics 64

Table 7 – Query processing time (ms) of the BEVA with the TRIE Built in laat and kaat strategies with prefix queries of sizes 5, 7, 9, 13 and 17 and τ from 1 to 3. 65

Table 8 – Simulation of memory cache misses collected by the CacheGrind program in the USADDR dataset for $\tau = 3$. 66

Table 9 – Query processing time (ms) of the LD with the TRIE Built in laat and kaat strategies with queries of sizes 5, 9, 13 and 17 and max_{depth} with values 6, 8, 10, 12 and 14 for τ varying from 1 to 3 in the USADDR, MEDLINE and DBLP datasets. 66

Table 10 – Query processing time (ms) of the LS with the TRIE Built in laat and kaat strategies with queries of sizes 5, 9, 13 and 17 and max_{slot} with values 1, 2, 4, 6, 8 and 10 for τ varying from 1 to 3 in the USADDR, MEDLINE and DBLP datasets. 67

Table 11 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 3$. . . . 69

Table 12 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 3$. . . 69

Table 13 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 3$. 70

Table 14 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 1$. . . . 71
Table 15 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 1$. . . 72
Table 16 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 1$. 72
Table 17 – USADDR - Average number of operations per query (on the scale of thousands) in query processing in the 1º and 2º level in the prefix query length 5, 9, 13 and 17 for $\tau = 3$ in the LD and LS methods. 73
Table 18 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 4$. . . . 74
Table 19 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 4$. . . 74
Table 20 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 4$. 75
Table 21 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 2$. . . . 76
Table 22 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 2$. . . 76
Table 23 – Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 2$. 77

Table 24 – Number of nodes and memory usage in TRIE index build in the USADDR, MEDLINE and DBLP datasets 79
Table 25 – Query response time (ms) comparison with confidence interval between baselines methods varying τ from 1 to 4, $|\mathcal{P}| = 5$ in the USADDR, MEDLINE and DBLP datasets. 81
Table 26 – Query response time (ms) comparison with confidence interval between baselines methods varying τ from 1 to 4, $|\mathcal{P}| = 13$ in the USADDR, MEDLINE and DBLP datasets. 81
Table 27 – Query response time (ms) comparison with confidence interval between baselines methods, varying the prefix queries sizes 5, 9, 13 and 17 for $\tau = 3$ in the USADDR, MEDLINE and DBLP datasets. 82

List of Algorithms

Algorithm 1 – Insert Query Suggestions with kaat strategy 49
Algorithm 2 – Insert Query Suggestions with laat strategy 49
Algorithm 3 – Insert Query Suggestions with LD Method 53
Algorithm 4 – Insert Query Suggestions with LS Method 56
Algorithm 5 – Process prefix query 58
Algorithm 6 – Find active nodes 59
Algorithm 7 – Find word active nodes 59
Algorithm 8 – Calculate edit distance and save new word active nodes 59

Contents

1 Introduction **23**

2 Related Work **29**

3 Preliminaries **33**

3.1 Problem Definition 33

3.2 Alternative Solutions to Pattern Matching 34

3.2.1 Sequential Search 34

3.2.2 Index-Based Search 35

3.3 TRIE 36

3.4 Active Nodes 37

3.5 Two-level Search 38

3.6 BEVA 39

3.6.1 Boundary Active Prefix Set 39

3.6.2 Edit vector 41

3.6.3 Edit Vector Automata 43

3.6.4 Query Processing With Edit Vector Automata 45

4 Improving BEVA **47**

4.1 Index Building Strategies 47

4.2 Two-Level Approach 49

4.2.1 Limited Depth Method 50

4.2.2 Limited Slot Method 54

4.2.3 Query Processing with Two-Level Approach 56

4.2.3.1 Running a Query with LS method 60

5 Results **63**

5.1 Experiments Setup 63

5.1.1 Experienced Methods 63

5.1.2 Experimental Environment and Datasets 63

5.1.3 Settings 64

5.2 Improving Index Building 65

5.3 Selecting Parameters for the Proposed Methods 67

5.3.1 Query Response Time 68

5.3.2 Memory Consumption 78

5.4 Baselines Comparison 80

5.4.1 Varying Edit Distance Threshold 80
5.4.2 Varying Query Length 81
5.4.3 Varying Dataset Portions for Memory Consumption 82

6 Conclusion 85

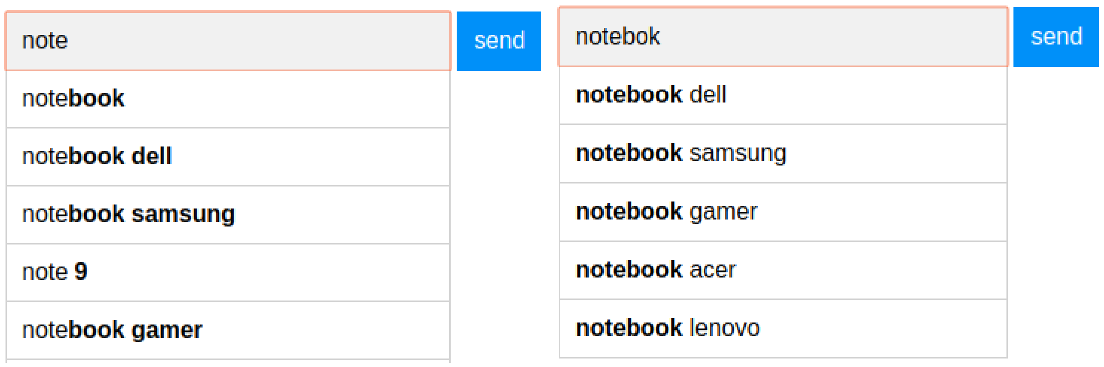
Bibliography 87

1 Introduction

Search systems are present is most of current programs available nowadays, including e-commerce services, web search engines, dialing applications in mobile phones, among others. One of the important feature of such systems is the property of performing query autocompletion, which is an essential component in the interaction between the user and the input interface of the search systems. Figure 1a shows an example where the user has typed the prefix "note" and the system suggests possible queries that match with it.

When searching in a system, the users can submit queries that contain typos that might result into unsatisfactory or even into empty query results. Because of this, recent works have propose the error-tolerant 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), where results within a small amount of errors are allowed, being the number of errors usually an edit distance between two sequences that represents the minimum number of operations that includes insertion, deletion or substitution of characters to make two sequences equal.

Error-tolerant query autocompletion can be seen as a mechanism to teach users about how to correctly spell difficult queries or to fix typos when the user is writing a prefix query. An example of a search system that allows error-tolerant query autocompletion is showed in Figure 1b, where the user receives suggestions "notebook dell", "notebook samsung", "notebook gamer", "notebook acer" and "notebook lenovo", all of them being answers that match with erroneous typed prefix "notebok".



(a) Example of Simple Query Autocompletion.

(b) Example of Error-Tolerant Query Autocompletion.

Figure 1: Query autocompletion in search box.

This mechanism can be seen as a specialized version of pattern matching problem. In the pattern matching problem, a search window with the size of the pattern is moved from left to right along the text and the pattern is searched within this window. There are two types of pattern matching problems: (1) The exact pattern matching and (2) the approximate pattern matching. The exact pattern matching is defined as all occurrences of an given pattern $P = p_1, ..., p_m$ in a text $T = t_1, ..., t_n$, with $m \leq n$, and each occurrence being a string equal to the pattern defined by a starting position in T and with the size of the pattern. The approximate pattern matching is defined as all occurrences of an given pattern $P = p_1, ..., p_m$ in a text $T = t_1, ..., t_n$, whose have a given edit distance within a given threshold k when compared to the pattern.

The approximate version of the string pattern matching problem has several practical applications, being applied, for example, when searching into DNA sequences or, as discussed here, to provide suggestions of the correct spelling of queries in search systems when the user makes any mistake while typing a query. In additional, the approximate pattern allows us to define more formally the problem treated in error-tolerant query autocompletion. Given an input pattern P of length m, a set $Qs = Q_1, ..., Q_n$ of n strings representing possible query suggestions and a threshold that limits the maximum number of errors τ, the error-tolerant query autocompletion problem can be defined as the problem of finding a subset of Q composed of elements whose prefixes match P with at most τ errors.

This problem has been addressed by several authors in recent research articles published in the literature Ji et al. (2009); Li et al. (2011); Xiao et al. (2013); Deng et al. (2016); Zhou et al. (2016). And the most successful current error-tolerant solutions adopt a TRIE data structure Fredkin (1960) to solve this problem. A TRIE is a search tree where the keys are usually strings with symbols belong to a predefined alphabet Σ, which each character of the string is stored as a key on a node. Each path from the root to a leaf of the TRIE represents a string, and each node in this path represents one of the characters of such string, with the root node representing an empty string. All equal prefixes in the TRIE share the same nodes.

While solutions based on data structure TRIEs represent the current state-of-the-art methods to perform fast query autocompletion computation allowing errors, these data structures and the proposed algorithms to search on them present a large memory consumption that makes the query autocompletion processing unfeasible for scenarios where the query dataset is large, which are quite common scenarios in practical applications. For instance, the BEVA algorithm proposed by Zhou et al. (2016), which is one of the fastest current solutions for query autocompletion, performs the pre-computation of large data through edit vectors that store them in a set of distinct states forming a structure called edit vector automata (EVA). This structure allow the query processing very efficient, but the memory consumption required to store both the TRIE and the edit vector automata is quite high.

On the other hand, time performance is also critical in autocompletion systems, once the query suggestions should quickly appear as the users are typing their queries, and the answers should be changed as each new character is typed. Miller (1968) suggest that each query needs to be completed in a total time of less than 100 ms to avoid noticeable delays during interactive search sessions.

We here study alternative forms of implementing query autocompletion methods. We focused our study on improving the BEVA method. We investigate alternative ways of using two-level indexing approaches to reduce the memory requirements for processing queries. We also propose and experiment a novel index building strategy for query autocompletion that allows faster query processing than Original BEVA by reducing memory cache misses when processing queries.

The two-level strategies studied perform the search in two steps. In the first step, we perform a search in the TRIE index, but this TRIE does not necessarily contain the full path to find the suggestion results. As a consequence, when the system reaches a leaf of the TRIE, this leaf may be associated to a set of strings, instead of a single string. If necessary, a second step is performed, and the matching processing should then continue to select the strings that match the query among the ones found by the TRIE in the first step. For these strings, the second step performs a sequential search. We name this approach as two-level processing approach.

We implemented the proposed ideas using the algorithm BEVA to process queries in the first level. In the second level, a sequential search is performed incrementally from the results of the first level. This sequential search continues the query processing by using BEVA. The only difference is that this second step does not use a TRIE, but just sequentially process all the strings selected by the first step. The key idea behind our method is that we adjust the amount of data indexed in the TRIE to be large enough to allow just a small number of strings need to be processed in the sequential second step. On the other hand, as smaller is the size of the paths indexed in the TRIE, as more advantageous is our method when compared to a method that uses the TRIE to index the full path of each string.

BEVA is one of the fastest methods in the literature, where it makes use of a structure called Edit Vector Automata (EVA) to calculate the edit distance between two strings. Such structure is built from edit vectors Ukkonen (1985). In addition, BEVA defines a set of active nodes, called boundary active nodes, which considerably reduces the number of nodes manipulated during query processing in previous work. Each active node in the TRIE is associated with a state in the automaton. We chose BEVA as the basis for our study because it is the state-of-the-art method in the literature.

We studied two ways of reducing the number of nodes in the first level of the TRIE. In the first, named as Limited Depth (LD), the size of this prefix is controlled by a parameter called max_{depth}. In the second, named as Limited Slots (LS), we limit the number of entries associated to each leaf of the TRIE, controlling it by a parameter called max_{slot}. Notice that in LD we do not control the number of entries in the second level associated to each leaf, while in LS we do not control the maximum size of a path in the TRIE.

In addition, we studied two strategies to build the TRIE. In the first strategy the index is built in depth-ward direction, that is, we insert one key in the TRIE, and thus we named it as key-at-a-time (kaat). Kaat is the natural insertion strategy for keys in a TRIE and is the one usually described or assumed in autocompletion articles Ji et al. (2009); Li et al. (2011); Xiao et al. (2013); Deng et al. (2016); Zhou et al. (2016). In second strategy, the index is built in breadth-ward direction, that is, we insert a position of all keys at a time, creating a new level in the TRIE. Only after that we start to insert the flowing position of each key, thus creating the following level. We repeat the process until inserting the whole keys in the TRIE. We named this strategy as level-at-a-time (laat), and as far as we know, it is a novel insertion strategy that was not considered before in the literature.

As we show in the experiments, our methods with two-level approach results in a considerable reduction of memory requirements for processing error-tolerant query autocompletion. For instance, our method requires only about 5% of the memory required by original BEVA algorithm for processing queries of dataset MEDLINE[1], a dataset usually adopted in experimental evaluation of error-tolerant query autocompletion methods, while keeping a time performance close to BEVA.

Our experiments also show a large difference in query processing times when we comparing laat and kaat strategies. The main algorithms in the literature, such as: Ji et al. (2009); Li et al. (2011); Zhou et al. (2016) process queries in breadth-ward direction, which makes laat the best insertion strategy to fastening the query processing. In our experiments, laat strategy was able to make BEVA up to four times faster.

Our contributions can be summarized as:

- We studied and compare two strategies for TRIE index building, the laat and the kaat strategies.
- We propose and study two alternative ways of using BEVA with two-level indexes: LD and LS methods.

The remaining of this dissertation is organized as follows. In Section 2 we present the related work and present the error-tolerant query autocompletion algorithms available in the literature. In section 3, we formally define the problem and also some important definitions necessary for easy understanding in our proposed method. In Section 4 we

[1] `https://www.nlm.nih.gov/databases/download/pubmed_medline.html`

present different ways to build the trie index, with the kaat and laat strategies. In addition, we present details about the proposed two-level approach methods , and also show how to implement them using BEVA, which one of the fastest methods for error-tolerant query autocompletion. In Section 5 we present experiments performed with our two strategies to build the TRIE index and our proposed methods, as well as comparison of our methods with baselines. Finally, in section 6, we present our conclusions and possible future research directions.

2 Related Work

The widespread use of query autocompletion in search systems has attracted attention to this problem in the literature. Grabski and Scheffer (2004) studied the query autocompletion and discuss a retrieval model to select sentences to be shown to users from the ones that autocomplete the query user. Their model is motivated by administrative and call center environments, in which users have to write documents with a certain repetitiveness. Bast and Weber (2006) propose the Hyb data structure, which basically uses inverted indexes, computing for each prefix, the union of inverted lists of all words that complete such prefix. Nandi and Jagadish (2007) also studied the query autocompletion problem as a multi-word "phrase" and introduced a FussyTree structure to select autocomplete phrases for a given prefix.

We here address the problem of error-tolerant query autocompletion. It was first studied by Chaudhuri and Kaushik (2009) and Ji et al. (2009), which proposed solutions based on incrementally maintaining a set of *active nodes* on a TRIE Fredkin (1960). In their approach, each character typed by the user is adopted to update the list of active nodes. After updating such list, the result can be reported by taking all the leaf nodes that can be reached from the active nodes in the TRIE. While both 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 precompute the answer active nodes for all these classes. This strategy changes the time complexity to maintain the list of active nodes, and while the time complexity of each maintenance step is $O(\tau \cdot (|A| + |A'|))$ for Ji et al. (2009), the time is $O(|A| + |A'|)$ for Chaudhuri and Kaushik (2009), where $|A|$ and $|A'|$ are the numbers of active nodes before and after the maintenance, respectively. The number of active nodes can be extremelly high when performing error-tolerant query autocompletion, which can slow down the search process. For instance, this number can be up to $O((|Q| + \tau)^{\tau} \cdot |\Sigma|^{\tau})$ in the proposal of Ji et al. (2009).

Several authors have presented improvements in the idea of using a TRIE and computing active nodes. Li et al. (2011) improved the method proposed by Ji et al. (2009) to reduce memory consumption and query response time by only considering the subset of active nodes with the last characters being neither substituted nor deleted, reducing the number of active nodes to $O((|Q| + \tau - 1)^{\tau} \cdot |\Sigma|^{\tau})$, which still can become quite high.

In another effort to reduce the costs for computing active nodes, Deng et al. (2016) propose the method META. One additional feature proposed in META is the ability of supporting top-k queries. Authors designed a compact tree index to maintain active nodes in order to avoid the redundant computations that occurs in previous proposals, such as ICAN (Ji et al. (2009)) and ICPAN (Li et al. (2011)). They also devised an incremental method to efficiently answer top-k queries.

Zhou et al. (2016) propose an even more efficient evaluation strategy for the active nodes that speedups the query processing by entirely eliminating ancestor-descendant relationships among active nodes. They show how to implement their new strategy by using a data structure named edit vector automaton (EVA). Their experiments show that their complete method, named BEVA, outperforms existing approaches in both space and time efficiency. While any of the above mentioned methods could be adopted as the first level in the two-level query autocompletion method proposed here, we here adopted BEVA as the method for the first step query processing in our work, given its high performance.

In an alternative direction when compared to the method BEVA, Xiao et al. (2013) propose the method incNGTrie, a proposal focused in speed up the query processing while increasing the amount of memory required for processing queries. It builds a TRIE for all the τ-deletion variants of the data strings and process the query by a simple matching procedure. This change results in performance gains when compared to other methods. However, its indexes sizes become several times larger than the other baselines, while the effective gain in time performance is not so expressive. 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 still their method requires more memory than methods BEVA and META, their new proposal reduces such difference, at a price of a significant increasing in the time for indexing the databases.

We here study the combination of a TRIE with sequential search for performing error-tolerant query autocompletion, deriving a method that is at the same time fast and memory efficient. Two-level strategies for indexing and search large string sets were previously adopted in the literature. Manber et al. (1994); Baeza-Yates and Navarro (2000); Navarro et al. (2000) studied the possible combination between full inverted index and sequential search with no indexing. In these works, the full search into text collections is performed by using an index that points to blocks, rather than each position in the text. The search is performed first in this index to detect blocks that might match the query, and then a second level with sequential search is performed to find the real list of occurrences in the text.

The application of TRIES and other structures into a two-level index has also being considered in previous work. Sussenguth Jr (1963) studied the use of a two-level structure to access file systems where the first level index was a TRIE. In their application, the mixed strategy decreased the quantity of TRIE nodes by a factor of six without high variation in the running time. Heinz et al. (2002) propose the so called Burst TRIES which can be seen as another example of two-level index applied to a distinct context. The Burst TRIES are collections of small data structures, called containers. Such containers represent the second level index, that are accessed via a conventional TRIE, which can be considered as the first level index.

3 Preliminaries

3.1 Problem Definition

Let Σ be a finite alphabet of symbols or characters. A string s is an ordered array of symbols drawn from Σ with length denoted by $|s|$ and $s[i]$ being the ith character of s, starting from 1. The substring of s is denoted by $s[i..j]$ between positions i and j. Given two strings s and s', where $s' = s[1..i], 1 \leq i \leq |s|$ the $s' \leq s$ denotes that s' is a prefix of s.

The definition of error-tolerant query autocompletion is given by a string $\mathcal{P}_{\mathcal{Q}}$ representing the prefix of the query already typed by the user, a set of strings $\mathcal{D}$ representing the possible suggestions to complete the prefix query that the user is typing, and a maximum edit distance threshold (maximum number of errors) represented by symbol τ. The problem is to given a prefix query $\mathcal{P}_{\mathcal{Q}}$, a dataset $\mathcal{D}$, and an edit distance threshold τ, the error-tolerant query autocompletion task is to: (1) return all $d \in \mathcal{D}$ sentences that are similar to $\mathcal{P}_{\mathcal{Q}}$ in at most τ edit distance operations, and (2) be able to efficiently process the subsequent prefix queries when additional characters are appended to $\mathcal{P}_{\mathcal{Q}}$.

Given two character strings s_1 and s_2, the edit distance between them defined as $ed(s_1, s_2)$ is the minimum number of edit operations required to transform s_1 into s_2 or vice versa. Most commonly, the edit operations allowed for this purpose follow the so called Levenshtein edit distance proposed by Levenshtein (1966), and include the following operations:

1. **Insertion** of a single character. If $a = xz$, then inserting the symbol y produces $a = xyz$. This can also be denoted by $\epsilon \to y$, using ϵ to denote the empty string.

2. **Deletion** of a single character changes $a = xyz$ to $a = yz$. This can also be denoted by $x \to \epsilon$.

3. **Substitution** of a single character x for a symbol $y \neq x$ changes $a = xyz$ to $a = yyz$. This can also be denoted by $x \to y$.

Chaudhuri and Kaushik (2009) propose for the first time the query autocompletion algorithm that allow a small amount of errors in input query and consequently increase the usability significantly of systems that has search interfaces mainly in mobile applications that naturally has an bad usability.

3.2 Alternative Solutions to Pattern Matching

The pattern matching solutions can be divided into two subgroups, being solved either using algorithms that index the dataset to be searched before proceeding the search, the so called index-based approach, or using algorithms that do not index the dataset, the so called sequential search approach.

3.2.1 Sequential Search

Algorithms that do not have indexes are also known as sequential search algorithms, in which a sequential search is performed comparing the pattern with each string in the dataset searched. As we are interested here in the query autocompletion allowing errors, a sequential search comparing the prefix to each string in the dataset searched should be performed. We describe below a solution for sequential search that is based on dynamic programming. While it is not the best solution for solving the sequential string matching problem, it is important because the idea is the basis for other more sophisticated pattern matching algorithms we study here.

A well known method to compute the edit distance between two strings d and Q (of length n and m, respectively) is the dynamic programming algorithm that fills in a matrix M of size $(n+1)\cdot(m+1)$. Each cell $M[i,j]$ records the edit distance between the prefixes of lengths i and j of the two strings, respectively. The cell values can be computed in one pass in row-wise or column-wise order based on the following recurrence equation:

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

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

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

Table 1: Edit distance calculation with classic dynamic programming.

Ukkonen (1985) observed that, based on the dynamic programming algorithm to calculate the distance between two strings d and Q, it is possible to obtain the same result by pruning the matrix values. This idea still uses dynamic programming but avoids keeping previously unnecessary values for a certain edit distance threshold. Such improvement was called of k-diagonal of the matrix and was defined as all the cells $M[i, j]$ such that $j - i = k$. To determine if the edit distance from d to Q is within τ, the threshold edit distance algorithm in Ukkonen (1985) only needs to compute the k-diagonals of the matrix, where $k \in [-\tau, \tau]$, according to **Table** 2, where cells in gray vertically represent the k-diagonal. The complexity is $O(\tau \cdot min(n, m))$. In addition, Ukkonen (1985) show the following lemma: $\forall\ M[i, j], M[i, j] \geq M[i - 1, j - 1]$.

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

Table 2: The cell values in color grey from top to bottom represent the k-diagonal varying from -1 to 1 for $\tau = 1$. The vertical in grey dark is the 0-diagonal.

However, even considering the best alternatives available in this paradigm, the sequential search is not an acceptable solution in most of the query autocompletion scenarios, since the dataset of query suggestions is usually too large and the time for performing sequential search becomes higher than the average time the users take to type their queries. Thus, sequential search is not considered as an option for solving the query autocompletion problem in the literature.

3.2.2 Index-Based Search

Index-based algorithms use specialized data structures to speed up the query processing. There are several alternative specialized index structures that are applied according to the characteristics of the problem addressed, including inverted index Baeza-Yates and Ribeiro-Neto (1999), signature files, suffix TRIEs and TRIEs. Among these data structures, the TRIEs are the ones that are currently adopted by most of the state-of-the-art query autocompletion methods that allow errors in the search as Zhou et al. (2016) and Deng et al. (2016).

While sequential search is not a viable solution, da Costa Xavier (2019) studied the possibility of combining it with index-based approaches for producing new solutions to the query autocompletion problem that are fast and reduce the amount of memory required for processing queries when compared to index-based approaches. This approach, named as two-level, adopts an index to reduce the search space and then proceeds a sequential search in this reduced space. Our proposal here is to further study this alternative.

3.3 TRIE

TRIE is a data structure usually applied to store strings whose symbols belong to a predefined alphabet Σ, which each character of the string is stored as a key on a node. A TRIE is a rooted tree formed as follows. It starts with a root node representing an empty string. Each node has references to a set of children nodes containing at most $|\Sigma|$ children, $|\Sigma|$ being the size of the alphabet. The insertion of a new string NS 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 $NS[pos]$. When finding this child, making $curr$ point to it and increasing *pos* by 1. When not finding, a new child node of $curr$ is created with the value of $NS[pos]$ 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 procedure described above, except that the search returns a fail when not finding a child node equal to $NS[pos]$. 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 $O(\Sigma \cdot m)$, where Σ is the size of the alphabet and m is the size of the searching 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, specially when searching for a large set of string, as it occurs in the problem of query autocompletion.

The exact search and approximate search can be performed using the TRIE data strucuture. Figure 2 show how the exact search and approximate search occurs in TRIE for the prefix query “undo”. The exact search occurs when the TRIE traversal is performed in prefixes that match exactly with the prefix query searched. When we processing also the approximate search, we need to analyze adjacent branches in order to find prefixes that are within a certain threshold. This feature increases the complexity of processing queries. Here we will approach the approximate search that is controlled by an edit distance threshold. The TRIE prefixes that are within this threshold are called of active nodes and then the processing is performed incrementally from the active nodes of the previous prefix query.

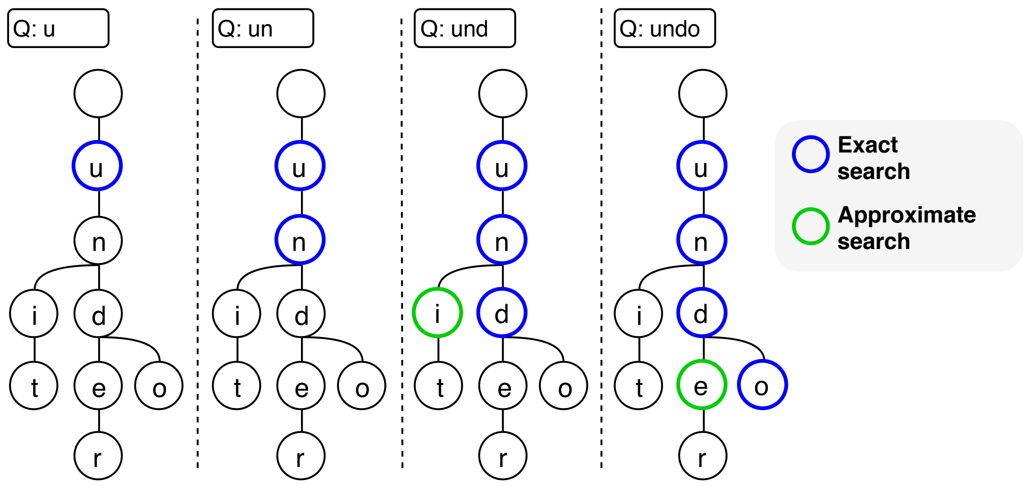


Figure 2: Representation of the TRIE data structure with the exact search and approximate search for the prefix query "undo". Nodes in blue color represent the nodes that exact search processing traverses. The green nodes represent the nodes that are traversed when the search is approximate and in this case the blue nodes can also be part. For query "undo" and $\tau = 1$.

3.4 Active Nodes

An active node is characterized by the edit distance between a prefix from dataset and a prefix query that is within the edit distance threshold, and the active node set is formally defined as: $\mathcal{V}_{\mathcal{P}_{\mathcal{Q}}} = \{\mathcal{P}_d \mid \mathcal{P}_d \in \mathcal{P}(\mathcal{D}) \land ed(\mathcal{P}_d, \mathcal{P}_{\mathcal{Q}}) \leq \tau\}$, where $\mathcal{P}(\mathcal{D})$ is prefix set from dataset $\mathcal{D}$, $\mathcal{P}_d \in \mathcal{P}(\mathcal{D})$ and it is one prefix from prefix set and $\mathcal{P}_{\mathcal{Q}}$ is the prefix query.

In Figure 3 we show the steps to computing the active nodes set as the prefix query changes. Consider the query "live" and $\tau = 1$. Initially, when we have the empty query ϵ, where ϵ represent the empty string, we consider the prefixes ϵ and "l" as active nodes because $ed(\epsilon, \epsilon) = 0$ (dashed circle) and $ed(l, \epsilon) = 1$ (bold circle). When the prefix query 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 represented by the ϵ and "l" prefixes and also their children. The stopping criterion is when we find a prefix that is outside the edit distance threshold. So, we get the prefixes ϵ, "l", "li" and "lo" 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 prefixes have an edit distance greater than ours edit distance threshold 1 and therefore are not part of the active nodes set for the prefix query "l". When the prefix query changes to "li" the root node is no longer an active node for this prefix query because $ed(\epsilon, li) = 2$. The rest of the active nodes from the previous prefix query remain as active nodes for the prefix query "li" and their edit distances are updated. In addition, the prefixes "lif" and "liv" become active nodes. This algorithm is repeated for the other prefixes query, until reaching the prefix query "live", in which the active nodes set is represented by the prefixes "life", "live" and "love".

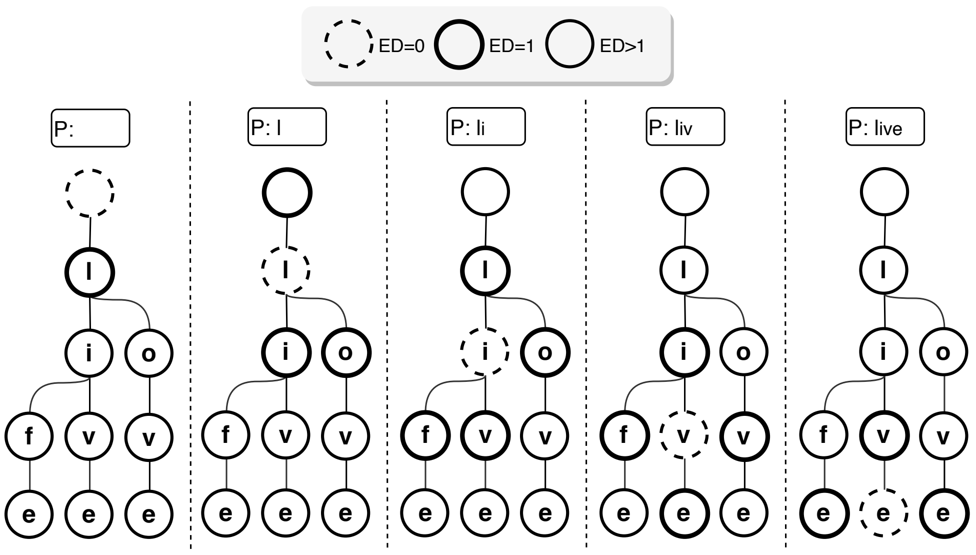


Figure 3: Computing the active nodes for $\mathcal{P}_Q$ = “live” and $\tau = 1$. The strings “life”, “live” and “love” are similar to $\mathcal{P}_Q$.

3.5 Two-level Search

The two-level search consists of a combination of two approaches that together combine the search final result. It is an alternative to one level search because it has some advantages, such as: reduction in memory usage and efficient query processing, since less data in memory do processing more efficient.

Broder et al. (2003) present an efficient query evaluation method based on a two-level approach. At the first level, the method iterates in parallel over query term postings and identifies candidate documents using an approximate evaluation taking into account only partial information on term occurrences and no query independent factors; at the second level, promising candidates are fully evaluated and their exact scores are computed. In this context, it is unnecessary to fully evaluate candidates who would not be part of the final result. Therefore a partial evaluation is done, in which it avoids a greater volume of processing. That said, the experiments in Broder et al. (2003) show that the algorithm reduces significantly the total number of full evaluations by more than 90%, without any loss in precision or recall.

da Costa Xavier (2019) proposes a error-tolerant query autocompletion method that use the two-level approach. The TRIE data structure with the ICAN algorithm (Ji et al. (2009)) were used in the first level with branches of the TRIE with fixe sizes 8, 9 and 10. When the processing reaches a leaf node the sequential search is performed based on the results of the first level. This work shows a significant reduction in memory consumption when compared to baselines. However, in some scenarios this method is not efficient for processing large volumes of data at the second level.

Therefore, our proposal in this work is to use the two-level approach in query autocompletion. Thus, it is believed that by indexing less data in the TRIE, fewer nodes are generated, and therefore fewer active nodes in query processing, saving memory space and improving query processing efficiency. When the partial indexing of the dataset in TRIE is not sufficient to determine the autocompletion result, the second level processing is used, which consists of the sequential search among the set of items resulting from the processing in the first level.

3.6 BEVA

In Chaudhuri and Kaushik (2009) and Ji et al. (2009) all prefixes that satisfy the edit distance constraint are kept as active nodes. Li et al. (2011) maintain a subset 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.

Besides that, the computation to obtain the edit distance between two strings s and t can be costly even with the best dynamic programming methods existing in the literature, due to the amount of comparisons in the TRIE that is performed in large datasets. Zhou et al. (2016) propose an efficient structure called edit vector automata (EVA), in which it computes the edit distance between two strings efficiently and also helps to maintain only a minimum set of active nodes, called Boundary Active Nodes Set.

3.6.1 Boundary Active Prefix Set

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: $\mathcal{B}_{\mathcal{P}_Q} = \{v \mid v \in \mathcal{V}_{\mathcal{P}_Q} \wedge (\nexists v' \in \mathcal{V}_{\mathcal{P}_Q} \wedge |v'| \leq |v|)\}$, according to Zhou et al. (2016).

In Figure 4 we show how the active nodes and boundary active nodes are obtained for the prefix query "live" and $\tau = 1$ in a TRIE structure with the strings "life", "live" and "love" indexed. Initially, we have the prefix query "l", the nodes represented by the prefixes ϵ (where ϵ represent the empty string in the root node), "l", "li" and "lo" are the nodes that have the edit distance for the prefix query within the edit distance threshold, that is, $ed(\epsilon, l) = 1$, $ed(l, l) = 0$, $ed(li, l) = 1$ and $ed(lo, l) = 1$, therefore are considered active nodes. The root node is 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 "li" as a prefix query, 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 the nodes represented by the prefixes: “l”, “li”, “lo”, “lif” and “liv”. 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 the node represented by the prefix “l”. And so on, until reaching the end of the prefix query.

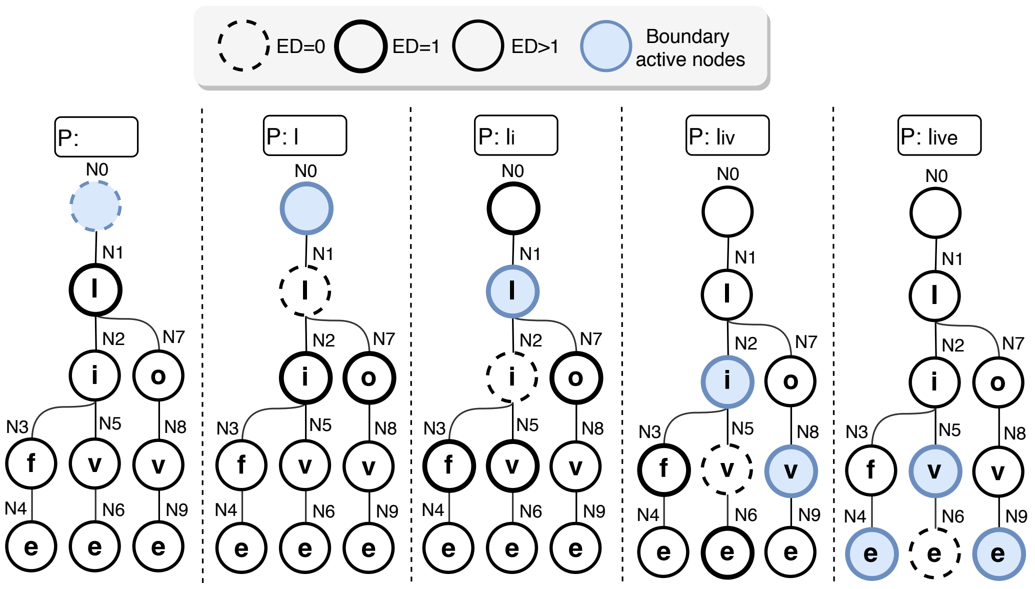


Figure 4: Representation of boundary active nodes (in blue) obtained step by step for the query “live” and $\tau = 1$.

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 this 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 5 we have the computation of the active nodes (in green) and the computation of the boundary active nodes (in blue) for the prefix queries “li” and “liv” , $\tau = 1$ and the prefixes indexed in the TRIE are “life”, “live” and “ love”. As we explained earlier, the active nodes for the prefix query “li” are represented by the prefixes: “l”, “li”, “lif”, “liv” and “lov”. Analyzing the boundary active nodes in this same step we have only one boundary active node represented by the prefix “l”, without having to save the other prefixes as boundary active nodes because it may cause duplicate results when taking the complete suggestions. In the active nodes computation, if we not keep the prefix “lo” for the next prefix query “liv” we not be able to obtain the active node represented by the prefix "lov", though this must be part of the result, because when analyzing the child nodes of "l" we obtain $ed(li, lov) = 2$, 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 an edit vector automaton. This also allows to maintain only the Boundary Active Nodes Set.

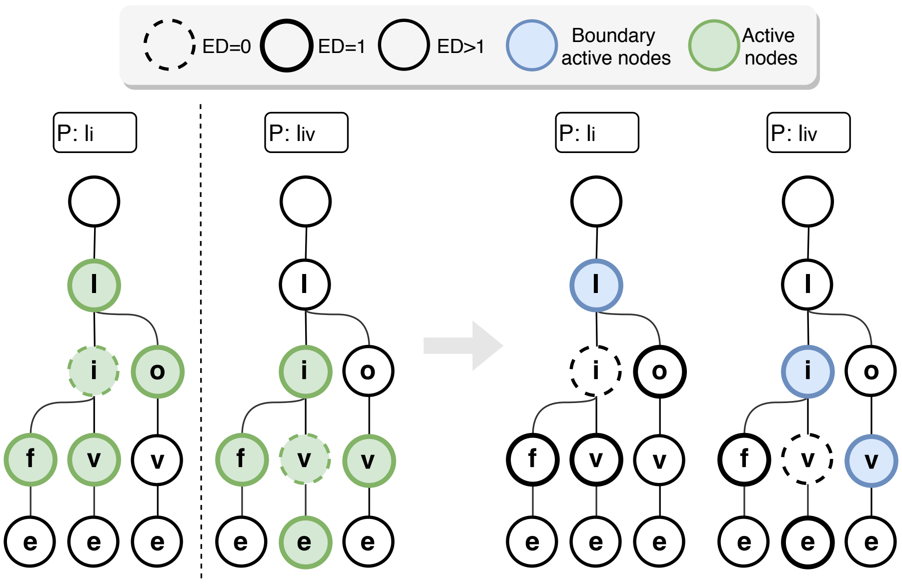


Figure 5: Difference between obtaining active nodes and boundary active nodes when changing the prefix query from "li" to "liv" and $\tau = 1$. On the left side we have the active nodes in green and on the right side the boundary active nodes in the same step.

3.6.2 Edit vector

A raw edit vector v_j with respect to τ is a column vector of size $2\tau + 1$ at the jth column of M. It is unnecessary to keep the actual value of cells whose value is larger than τ. No values that are greater than τ are saved. Therefore, the values of these cells are replaced with a special symbol # to generate the edit vector. The edit vector of column 0 always has the form $[\underbrace{\tau, \tau-1, \ldots, 1}_{\tau}, 0, \underbrace{1, 2, \ldots, \tau}_{\tau}]$, because the word in column 0 is empty. This characteristic is named as the **initial edit vector**. Similarly, the vector with all the symbols #, which is, $[\underbrace{\#, \#, \ldots, \#}_{2\tau+1}]$ is named as the **final edit vector**. From Table 3, we have four edit vectors: $[1, 0, 1]$, $[1, 0, 1]$, $[1, 1, 1]$, and $[\#, \#, \#]$.

The threshold edit distance computation is essentially computing the subsequent jth edit vector starting from $j = 0$. The computation is performed from the top-row cell to the bottom cell of the column, using the recurrence equation adapted from recurrence equation of the dynamic programming to obtain the edit distance between two strings:

$$v_{j+1}[i] = min(v_j[i] + \delta(d[j+1], Q[j-\tau+i]), v_j[i+1]+1, v_{j+1}[i-1]+1), \forall 1 \leq i \leq 2\tau+1.$$

Finally, the edit distance between Q and a data string d is given by $v_{|d|}[\tau+1+(|Q|-|d|)]$ when $|Q| \in [|d|-\tau, |d|+\tau]$ or more than τ otherwise.

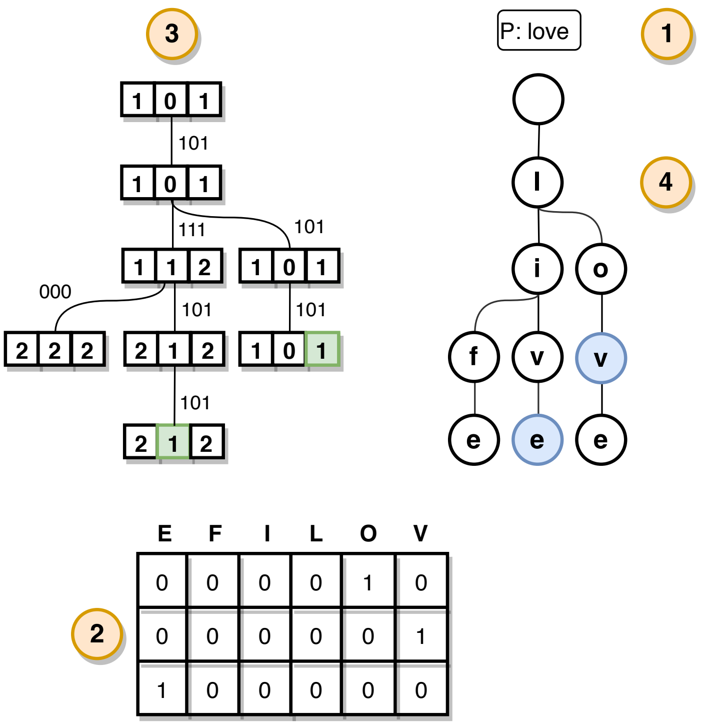
		0	1	2	3
		1	**m**	**i**	**d**
0		0	1		
1	**m**	1	0	1	
2	**a**		1	1	#
3	**i**			1	#
4	**n**				#
		V_0	V_1	V_2	V_3

Table 3: Edit vectors represented in yellow and green.

Zhou et al. (2016) modeled the computation of the subsequent edit vector called of v_{j+1} as a function f that receives a set of input parameters. The edit vector v_{j+1} was modeled as $f(v_j, \mathcal{B})$, where $\mathcal{B}$ is a binary *bitmap* of $2\tau+1$ bits and $\mathcal{B} = \neg\delta(d[j+1], Q[j-\tau+i])$, for $\forall 1 \leq i \leq 2\tau+1$. For example, suppose the edit vectors of **Table** 3 for calculating the transition function $\neg\delta(m, \epsilon) = \mathbf{0}$, $\neg\delta(m, m) = \mathbf{1}$, $\neg\delta(m, a) = \mathbf{0}$, the bitmap $\mathcal{B} = \mathbf{010}$. Then $f([1, 0, 1], 010) = [1, 0, 1]$. The authors from BEVA observed that no matter what characters $d[j+1]$ and $Q[j-\tau+i]$ are, it is whether they match or not that affects the resulting edit vector. The bitmaps are used only to compare the prefix label from TRIE with each character of the prefix query, in which the comparison in prefix query size varies from $[|n|-\tau, |n|+\tau]$, where $|n|$ represent the current prefix size from TRIE.

The control of the bitmaps can be performed in a table $\mathcal{H}$ that represents the characters that appeared or not in the prefix query. The size of $\mathcal{H}$ can be at most the size of the alphabet. The update of $\mathcal{H}$ is performed as the prefix query changes. Supposing $\tau = 1$ and the last $2\tau+1$ characters of the prefix query "lov" (Figure 6), the dynamic table of bitmap $\mathcal{H}$ maps "l" to 100, "o" to 010, "v" to 001 and all the other characters to 000. If a new character "e" is appended to the prefix query, we will delete the character "l" in the table of bitmap (delete means set the bit to 000) and update the bitmap of "o" and "v" by shifting 1 bit leftwards and then add "e" = 001.

In addition, we need to construct also a new boundary active node set from the descendants of current boundary active nodes (represented by the prefix "liv" and "lo"). Hence, we iterate over all the current boundary active nodes to traverse its descendants, and for each child node, the corresponding bitmap is generated to obtain the next edit vector. The corresponding bitmap is generated using the formula $\mathcal{H}(n'.char) \ll (\tau - |\mathcal{P}_Q| - |\mathcal{P}_d|)$, where n' is the current node from TRIE, $|\mathcal{P}_\mathcal{Q}|$ is the prefix query size and $|\mathcal{P}_d|$ is the size of the prefix from TRIE, according to Zhou et al. (2016). Finally we get the new boundary active nodes represented by the prefixes "live" and "lov".



	E	F	I	L	O	V
	0	0	0	0	1	0
	0	0	0	0	0	1
	1	0	0	0	0	0

Figure 6: Representation of the boundary active nodes set computation as the prefix query changes. For this example we have the query "love", $\tau = 1$ and the prefixes indexed in the TRIE are "life", "live" and "love". For this example, although the edit vectors are represented in a different structure from the TRIE, the edit vectors are manipulated in the TRIE nodes.

Zhou et al. (2016) realized that it is not necessary to compute the next edit vector during the boundary active nodes set computation. So the authors from BEVA proposed a structure called Edit Vector Automata or just EVA that pre-computes all the edit vectors that can be generated based on bitmaps. And from then on, every boundary active node is associated with a state in this automaton.

3.6.3 Edit Vector Automata

The key idea of the Edit Vector Automata (EVA) algorithm is to store all edit distances between the k-diagonals where $k \in [-\tau, \tau]$. Thus, the idea of edit vector can be encoded as a state in an automaton edit vector. The total number of precomputed table entries will be upper bounded by $|\mathcal{V}(\tau)| \cdot 2^{2\tau+1}$, where $|\mathcal{V}(\tau)|$ represents the number of distinct states and hence does *not* depend on the size of the alphabet Σ - the number of entries is only 41,344 for $\tau = 3$, according to Zhou et al. (2016).

Zhou et al. (2016) define the algorithm to compute of the edit vector automaton (EVA) through of the following steps:

1. Add $\mathcal{S}_0$ state associated with the initial edit vector inside the empty queue.

2. While the queue is not empty, the state $\mathcal{S}_i$ of the queue is removed. Compute $\mathcal{S}' = f(\mathcal{S}_i, \mathcal{B})$ for every $2^{2\tau+1}$ possible values of $\mathcal{B}$. Record these transitions in the automaton. Finally, if $\mathcal{S}'$ is a new state, add it into the queue.

   Each new state is pushed into the queue exactly once. For each state propped from the queue, are computed $2^{2\tau+1}$ transitions, each taking $O(\tau)$ time. Testing if a state is new takes $O(1)$ time. In total the complexity of the algorithm is $O(\tau \cdot 6^{2\tau})$, according to Zhou et al. (2016).

Finally, Zhou et al. (2016) define formally the edit vector automaton with respect to τ as an 5-tuple $(\mathcal{S}, \mathcal{B}, f, \{\mathcal{S}_0\}, \{\mathcal{S}_\perp\})$.

- $\mathcal{S}$ is the set of states and each state is associated with a single edit vector.
- $\mathcal{B} = \{0,1\}^{2\tau+1}$, is the set of all bitmaps of size $2\tau + 1$ that drives the state transitions.
- f is the transition function of the edit vector.
- $\mathcal{S}_0 \in \mathcal{S}$, is the only *initial state* associated with the initial edit vector.
- $\mathcal{S}_\perp \in \mathcal{S}$, is the only *final state* associated with the final edit vector.

After executing the automaton construction algorithm, the edit vector automaton is obtained, as shown in Figure 7. In Figure 7, we have the EVA structure for $\tau = 1$. As we can see in this figure, the initial state represented by $[1, 0, 1]$ is always associated with the initial edit vector previously defined. For each state, the other states that can be generated for the $2^{2\tau+1}$ possible transitions for the bitmap of size $2\tau + 1$ are calculated. Although we show a large number of transitions, most of them can generate the same edit vector and thus have a transition for itself in the automaton. In the initial state, bitmaps 010, 011, 110 and 111 led to the same state $S0$ associated with the edit vector $[1, 0, 1]$. Bitmaps 001 and 101 led to the state $S1$ associated with the edit vector $[1, 1, 1]$.

Let's take bitmap 001 and show as occurs $f([1, 0, 1], 001) = [1, 1, 1]$. For this we need to apply the recurrence equation that generates the edit vectors: $v_{j+1}[i] = min(v_j[i] + \delta(d[j+1], Q[j-\tau+i]), v_j[i+1]+1, v_{j+1}[i-1]+1), \forall 1 \leq i \leq 2\tau+1$. With that, we have as input for this equation the previous edit vector $v_j = [1, 0, 1]$ and we will replace $d[j+1], Q[j-\tau+i]$ by the bitmap 001 with the corresponding position inverted. Thus, we can substitute in the equation, as: $v_{j+1}[0] = min(1+1, 0+1, \# + 1) = 1, v_{j+1}[1] = min(0+1, 1+1, 1+1) = 1, v_{j+1}[2] = min(1+0, \# + 1, 1+1) = 1$. Zhou et al. (2016) use the special symbol # to denote out-of-boundary cell values and also define three natural rules regarding the essential computations on #: (i) $\tau + 1 = \#$, (ii) $\# + 1 = \#$, and (iii) $\# > \tau$. Thus, $f([1, 0, 1], 001) = [1, 1, 1]$.

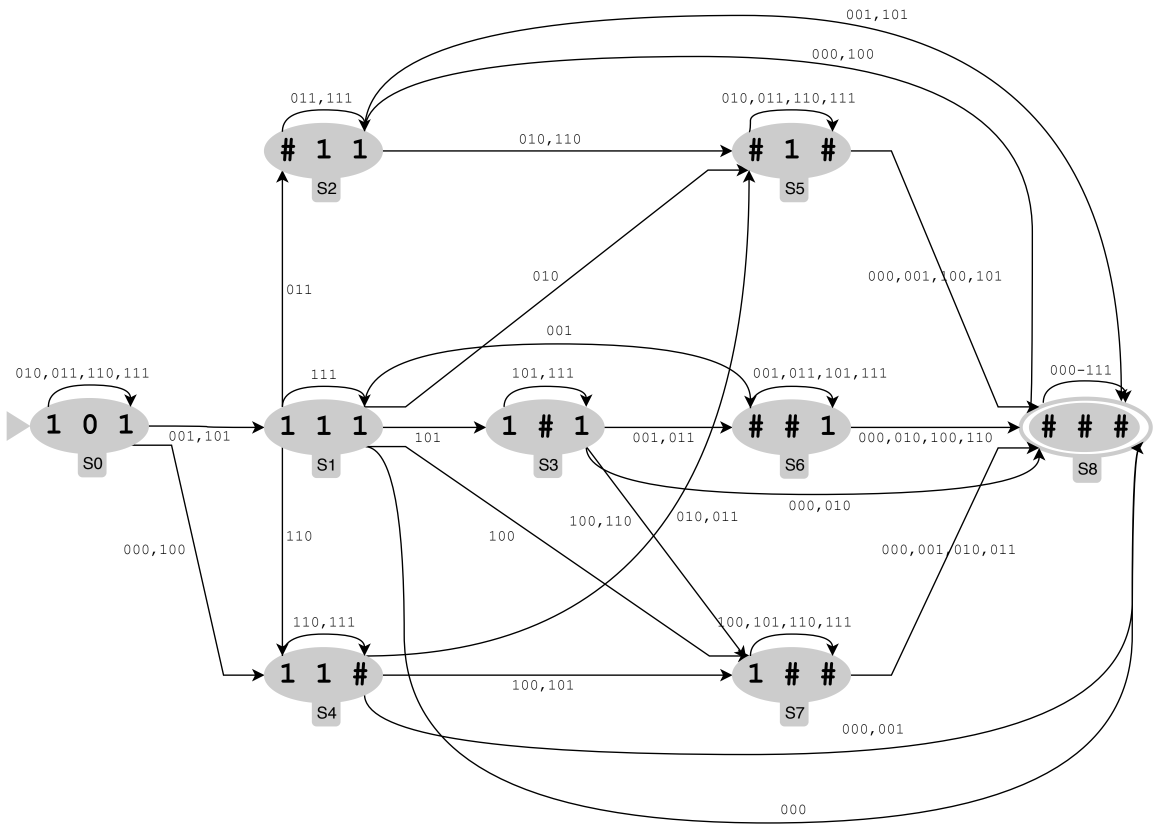


Figure 7: Edit vector automata for $\tau = 1$.

3.6.4 Query Processing With Edit Vector Automata

BEVA processes queries incrementally by using boundary active nodes from the previous query. Boundary active nodes are associated with a state in the edit vector automaton. The processing starts from root node of the TRIE and on each of its descendants. The edit distance is computed for each prefix in the TRIE. The edit distance calculation is performed through the edit vector automata (EVA), in which it has all the edit distances pre-computed. If the edit distance is within the edit distance threshold then the node is computed as a response and added to the list of boundary active nodes to be used in the next query, otherwise, the descendants of this node are analyzed and this process repeats until that a boundary active node is found or that the final state of the automaton is reached. In the latter case, it is no longer possible to find boundary active nodes at the lowest levels of the TRIE and then the computation is terminated.

An interesting consequence of definition of boundary active nodes is that if n is an active node of the current prefix query $\mathcal{P}_\mathcal{Q}$, then this cannot be an active node of the subsequent query $\mathcal{P}'_\mathcal{Q}$ and only its first descendant nodes in path of the subtrie that satisfy the edit distance constraint will be the active nodes of $\mathcal{P}'_\mathcal{Q}$. Zhou et al. (2016) define two lemma, that best describes these rules:

1. *For a given τ, none of the active nodes of a query $\mathcal{P}_\mathcal{Q}$ is an active node of the query*

$\mathcal{P}'_Q = \mathcal{P}_Q \circ c$, where $\circ$ denotes a concatenation, and a is an appended character

2. *For a given τ, a node n is an active node of a query $\mathcal{P}'_Q = \mathcal{P}_Q \circ c$ if and only if (1) one of n's ancestors is an active node of the query $\mathcal{P}_Q$, (2) none of its ancestor nodes is an active node of the query $\mathcal{P}'_Q$, and (3) it satisfies the edit distance constraint with respect to $\mathcal{P}'_Q$.*

There is a dynamic table of bitmaps values $\mathcal{H}$, where each bitmap is associated with a character, this table size is equal to the size alphabet. Adding a character to the input prefix query all bitmaps in the table are updated, i.e, the bitmaps have shifting 1 bit leftwards. New characters are added to $\mathcal{H}$ with bitmap 001, and in iterations followed it is only updated.

For example, consider the TRIE shown in the Figure 4 that contains the strings: "life", "live" and "love", $\tau = 1$ and characters of $\mathcal{P}_Q$ are "liv", the dynamic Table $\mathcal{H}$ maps l to 100, i to 010, v to 001, and all the other characters to 000. If a new character e is appended to query $\mathcal{P}_Q$, we will update the dynamic table to $\mathcal{H}(l) = 000$, $\mathcal{H}(i) = 100$ and $\mathcal{H}(v) = 010$ by shifting 1 bit leftwards and add $\mathcal{H}(e) = 001$. We show the boundary active nodes and their states associated with the edit vector for each step of the prefix query in **Table 4**.

Step	$\mathcal{P}_Q$	**Boundary Active Node Set** $\mathcal{B}_{\mathcal{P}_Q}$
1	ϵ	$\{[n_0, S_0]\}$
2	l	$\{[n_0, S_0]\}$
3	i	$\{[n_1, f(S_0, 010) = S_0]\}$
4	v	$\{[n_2, f(S_0, 010) = S_0], [n_8, f(S_0, 010) = S_5]\}$
5	e	$\{[n_4, f(S_4, 010) = S_5], [n_5, f(S_0, 010) = S_0],$ $[n_9, f(S_5, 010 = S_5)]\}$

Table 4: Query processing step by step with BEVA and prefix query "live".

4 Improving BEVA

In this chapter we show in detail our proposals for building TRIE index and processing prefix queries with low memory consumption and respecting the speed restriction in query processing time.

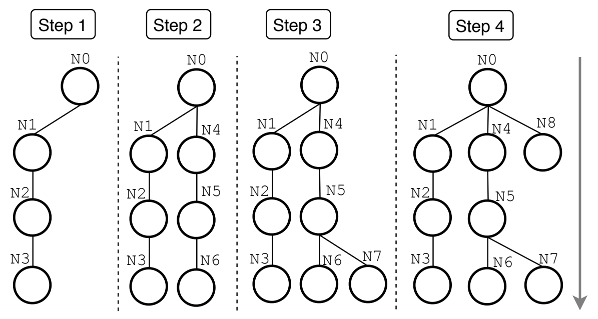
4.1 Index Building Strategies

Depth First Search (or just DFS) and Breadth First Search (or just BFS) are algorithms that determine the direction of search in a tree or graph, the only difference from one structure to the other is that the tree does not admit cycles and therefore the algorithm does not require a boolean array to mark visited nodes. Here we will use these concepts in the context of a TRIE data structure. However, the way of indexing strings in a TRIE is not addressed in the literature, but the depth-ward direction is more natural for indexing. As it is about insertions and not search, we will not treat such indexing strategy as DFS algorithm.

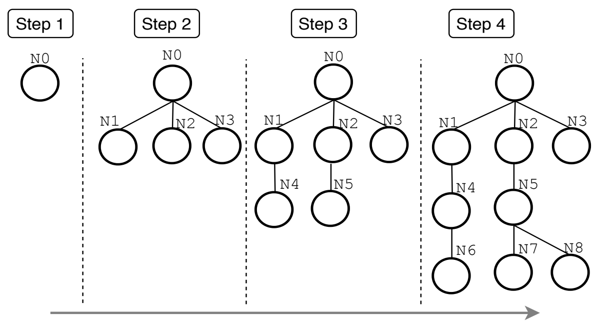
In DFS, the query processing in the TRIE is performed in depth-ward direction, that is, one prefix is entirely analyzed and then the next one entirely until all the prefixes be analyzed. In BFS, the query processing in the TRIE is performed in breadth-ward direction, that is, all prefixes have their first character analyzed and then all prefixes have the second character analyzed until all prefixes be analyzed. Currently, the main error-tolerant query autocompletion algorithms available in the literature perform the query processing using the BFS algorithm (Chaudhuri and Kaushik (2009); Ji et al. (2009); Zhou et al. (2016); Deng et al. (2016)) and the prefixes are indexed in the TRIE in depth-ward direction, indexing one prefix at a time, until all the prefixes of the dataset are indexed.

When implementing BEVA algorithm, we have found an important practical issue that affects not only BEVA performance, but also may affect the performance of other TRIE based algorithms proposed in literature. We realized that the way the TRIE is built in memory may largely affect the performance of the algorithm due to the cache effects in the memory hierarchy. The autocompletion algorithms, such as BEVA, are implemented using BFS search, since they need to find the new set of answers for each key typed by the user. Authors of BEVA have discussed an implementation of BEVA that uses DFS for being used to speedup situations where user types a query too fast, such as when the query is copied to the search box. Their experiments has shown that BEVA DFS can be sligthly faster for these situations, but difference in performance was small and BFS is the default strategy for searching if considering a user typing one character at a time.

When building the TRIE, the more natural way of inserting keys is to insert them in a key-at-a-time (*kaat*) approach, which creates all the nodes for a key as soon as the key is inserted. In the Figure 8a we show the direction in which the keys are inserted in the TRIE for the kaat strategy, in this case the nodes are inserted in the following order: N0, N1, N2, N3, N4, N5, N6, N7 and N8. A side effect of this strategy is that nodes of lower levels for new keys tend to become far from each other in the memory system, since they tend to be separated by the several nodes created to represent the full previous keys inserted. This distance can be decreased if we change the approach to a level-at-a-time (*laat*) TRIE construction, which inserts all keys in parallel. In the laat, we start by inserting the first character of all keys, then insert the second and so on. In the Figure 8b we show the direction in which the keys are inserted in the TRIE for the laat strategy, in this case the nodes are inserted in the following order: N0, N1, N2, N3, N4, N5, N6, N7 and N8. We argue that laat approach may reduce the distance between low level nodes of the TRIE in memory, which allows a better use of the machine cache system when processing queries.



(a) TRIE Index Building with kaat strategy. The TRIE is built from the top to the bottom in each step.



(b) TRIE Index Building with laat strategy. The TRIE is built from the left to right in each step.

Figure 8: Ways of building a TRIE index.

Algorithm 1 describes the TRIE index building with the kaat strategy. This algorithm receive $\mathcal{D}$ as parameter, in which represent the dataset. In line 2, all strings in the dataset are processed at a time. In line 3, the root node is obtained. In line 4, all characters from query are processed and in line 5 this character is inserted in the TRIE as a node. In the lines 6 to 8 the node interval is updated. Finally, the node is marked as leaf node in line 9.

Algorithm 2 describes the TRIE index building with the laat strategy. This algorithm receive $\mathcal{D}$ as parameter, in which represent the dataset. Initially, the maximum level of the TRIE is instantiated with the size of the first query from dataset and this variable can be updated as that found levels more depth in the TRIE, as describes in lines 7 and 8. In line 3 the iteration through of the levels of the TRIE is performed, this iteration control the level that the character of the records are inserted. In line 4 and 5 all characters from

Algorithm 1 Insert Query Suggestions with kaat strategy

```
1: procedure TrieIndexBuildingWithKaat($\mathcal{D}$)
2:     for each $\mathcal{Q}$ in $\mathcal{D}$ do
3:         $node \leftarrow r$
4:         for each $c$ in $\mathcal{Q}$ do
5:             $node \leftarrow insert(c)$
6:             if $node.beginRange$ = -1 then
7:                 $node.beginRange \leftarrow \mathcal{Q}_{id}$
8:             $node.endRange \leftarrow \mathcal{Q}_{id}$
9:         $node.isLeaf \leftarrow$ True
```

query are processed and the current level is checked to be less than the query size. In line 6 the last node known for the query (in the first time the only node known is the root node) is retrieved. In line 9 the node associated with the current character is created and inserted in the TRIE and the node interval is updated in lines 10 to 12. Finally, the last node created is saved as last node known for the query and the node is marked as leaf node if the current level reached the query size, lines 13, 14 and 15.

Algorithm 2 Insert Query Suggestions with laat strategy

```
1: procedure TrieIndexBuildWithLaat($\mathcal{D}$)
2:     $maxLevel \leftarrow |\mathcal{Q}_0|$                                  ▷ for $\mathcal{Q}_0 \in \mathcal{D}$
3:     for each level until maxLevel do
4:         for each $\mathcal{Q}$ in $\mathcal{D}$ do
5:             if $level \leq |\mathcal{Q}|$ then
6:                 $parent \leftarrow$ retrieve last node known for $\mathcal{Q}$
7:                 if $level > maxLevel$ then
8:                     $maxLevel \leftarrow level$
9:                 $node \leftarrow insert(c)$
10:                if $node.beginRange$ = -1 then
11:                    $node.beginRange \leftarrow \mathcal{Q}_{id}$
12:                $node.endRange \leftarrow \mathcal{Q}_{id}$
13:                $parent \leftarrow$ keeps last node known for $\mathcal{Q}$
14:                if $level = |\mathcal{Q}|$ then
15:                    $node.isLeaf \leftarrow$ True
```

4.2 Two-Level Approach

Query autocompletion methods based on TRIEs present a considerably high memory requirement to store the index. We here describe our solution to reduce this overhead with no significant increasing in query processing times. The main idea is quite simple: we index only small prefixes of the queries in the TRIE and use such index to select subset of queries that are candidate for matching the prefix query already typed by the user. We then perform a sequential search in this subset to find the actual list of suggestions that match the prefix typed by the user. The query processing is performed partially using the TRIE, the first level, and partially performing sequential search in the query results, the second level. We name this approach as two-level.

For implementing the two-level approach, we have selected BEVA, which is one of the fastest algorithms available in literature. BEVA has also the advantage of using the Edit Vector Automata (EVA) to perform the match in the fist level. While performing the second level, we take advantage of the states already computed by BEVA, and keeping using the EVA to performing the prefix matches in the second level of our algorithm. We show that this combination allows achieving a significant reduction in the size of the index, while time performance is quite close to the ones achieved by BEVA.

We have considered two alternative ways for implementing the two-level approach, namely: (1) Limited Depth or LD and (2) Limited Slot or LS. The limited depth method does not allow TRIE nodes above a specified maximum threshold level. The limited slot method limits the number of query suggestions associated to each leaf node in the TRIE. As a consequence, LS allows query suggestions with less frequent prefixes in the data set to have fewer characters indexed in the TRIE, whereas query suggestions that have very frequent prefixes have more characters indexed in the TRIE. This method aims at adjusting the number of levels stored according to the popularity of the prefix, which may have impact in the final performance. The hypothesis is that prefix that have more strings in common may also have more chance of being typed. Further, in such cases, fixing a maximum depth could result in a larger set of strings to be searched in the second level.

4.2.1 Limited Depth Method

Our first two-level approach lies on the strategy of limiting the maximum depth of the TRIE. The boundary indexing uses a TRIE structure, where a TRIE has a defined maximum depth called max_{depth} for all branches, starting at the root node. Using this approach, we may use any algorithm to search in the TRIE, such as BEVA or IncNGTRIE and, if the prefix searched is larger than max_{depth}, get the set of results as candidates for matching the prefix queries. We can then perform a sequential search in the set of candidates in order to select the suggestions that really match the query. Any method could be used for searching the TRIE in the first level. However, when using BEVA, we can take advantage the EVA states of each candidate to accelerate the query processing at the second level. We thus need to just continue to process the candidates from the starting point where the EVA automaton stopped in the first level.

A first observation about the proposed approach is that the query can be processed just using the first level if the given prefix is smaller than the upperbound $max_{depth} - \tau$, where τ is the number of errors allowed in the search task. This fact is important if considering that good query autocompletion systems provide relevant suggestions for small prefix, thus increasing the chance of users to click in a suggestion by just typing a few characters. If the prefix query typed by the user is bigger than $max_{depth} - \tau$, the second level is started and performs a sequential search in the suggestions selected by the first level.

When creating the TRIE, we insert the strings in a lexicographic order and index up to max_{depth} characters of each query suggestion. In such case, each node of the TRIE represents a set of all query suggestions that share a unique prefix represented by the node. We thus include in such node integer pointers to the beginning and to the end positions of the query suggestions associated to the node in the full list of query suggestions. Thus each node stores a range of integer values referring to the first and last position of its associated query suggestions in the dataset. Notice that all entries associated to a node will be in contiguous positions at the full list of query suggestions, since we sort them in lexicographic order before indexing.

For example, suppose we will index the "midday" and "midfield" strings in the TRIE, and their IDs are 1 and 2, respectively. When indexing the character "m", it is created a new node in the TRIE with the value "m" and interval [1, 1]. Then we index the character "i", and a new node is created with value "i" and same interval [1, 1] and so on until the end of the word "midday". When indexing new prefixes in a branch that already exists in the TRIE, only the intervals will be updated. For instance, when inserting the query suggestion "midfield", the character "m" is processed and it is detected that the character already exists in the TRIE. In this case, only its interval is updated from [1, 1] to [1, 2] also encompassing the prefix "midfield", and so on until the end of the query suggestion, as shown in **Figure** 9. This technique allows instant access to the list of query suggestions associated to the node, which significantly reduces the time for fetching the results.

Query processing starts at the first level with BEVA. When the query prefix is larger than the maximum prefix max_{depth}, the processing goes to the second level. For example, suppose the query suggestion "midday" has been indexed until the fourth position (second occurrence of "d") and the user has typed the query "m"-"i"-"d" (trace (-) is to simulate each keystroke) in this way the processing is performed entirely on the first level through BEVA. If the user had typed the prefix query "m"-"i"-"d"-"d"-"a", we had processed the query "midd" using BEVA and had taken all the results to be sequentially inspected to check for the occurrence of the complement "a".

In LD we index only part of the prefix. The size of the prefix to be indexed in the TRIE should be large enough to allow that the second level search is performed into a small set of items, so that this second level, which performs sequential search, takes an acceptable query processing time to be performed. On the other hand, it should be short enough to allow a significant reduction in the total amount of memory taken by the TRIE.

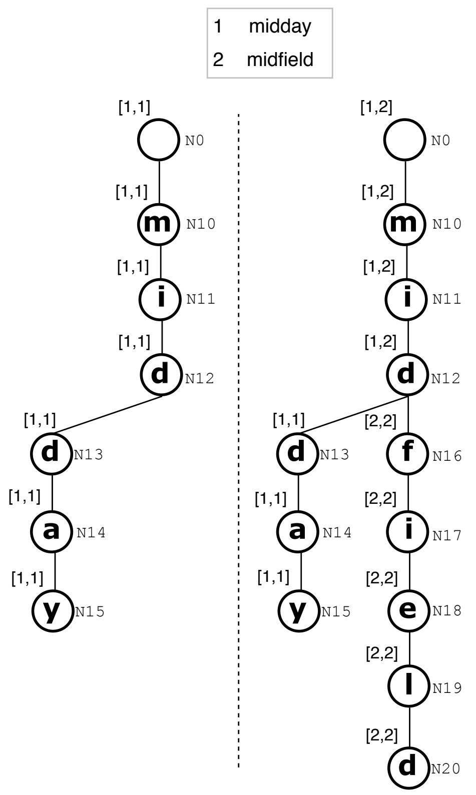


Figure 9: Example of the indexing with updates in the intervals.

The upper bound should be set through experiments to balance these two values according the necessity of either more performance or less space requirement. **Figure** 10 shows an example of TRIE using a fixed depth threshold for limiting the prefix indexed in the TRIE. In the example, only the 4 first characters of the query suggestions are inserted into the TRIE.

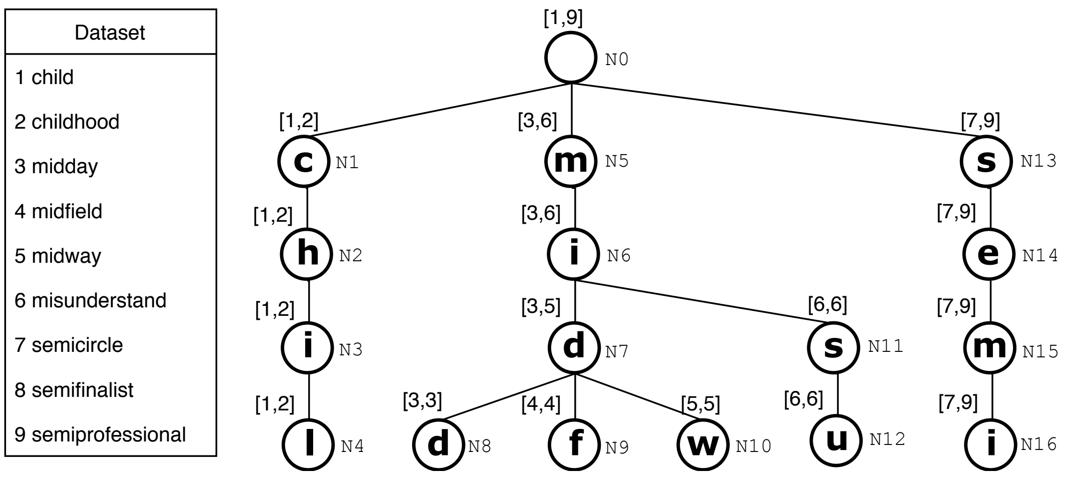


Figure 10: Indexing with fixed size branches 4.

Algorithm 3 describes the insertion of queries in the TRIE when using the Limited Depth method. The algorithm received as parameters: r, $\mathcal{Q}$, $\mathcal{Q}_{id}$ and max_{depth}, where represent the root node, query, query identifier and max depth in the TRIE, respectively.

Each query suggestion is inserted character by character in the TRIE, as shown in line 5. In lines 7 and 8 the query ranges are updated in the node. In line 9, the value of the variable *depth* is incremented, where *depth* is the current depth in the TRIE. In line 10 it is verified if *depth* is greater than the max_{depth}, to determine when stopping the insertion procedure.

Algorithm 3 Insert Query Suggestions with LD Method

```
 1: procedure InsertSuggestion($r$, $\mathcal{Q}$, $\mathcal{Q}_{id}$, $max_{depth}$)
 2:     $node \leftarrow r$
 3:     $depth \leftarrow 0$
 4:     for each $c$ in $\mathcal{Q}$ do
 5:         $node \leftarrow insert(c)$
 6:         if $node.beginRange$ = -1 then
 7:             $node.beginRange \leftarrow \mathcal{Q}_{id}$
 8:         $node.endRange \leftarrow \mathcal{Q}_{id}$
 9:         $depth \leftarrow depth + 1$
10:         if $depth \geq max_{depth}$ then
11:             break
12:     $node.isLeaf \leftarrow$ True
```

As we will show in the experiments, the naive two-level approach using the limited depth method already allows a significant decreasing in memory usage. For comparison, we can see in **Figures** 4 and 10 an example comparing the number of nodes in TRIE that the baseline BEVA and others algorithm in literature achieve when inserting a small set of strings in the TRIE. Figure 10 shows the number of nodes in TRIE that two-level approach requires when using the Limited Depth method. In the example, there is a reduction from 64 to 16 nodes, that is, 48 nodes less in the TRIE. The list of strings in lexicographic order is shown on the left side of the figure, the TRIE with the respective strings is shown on the right side, but all are limited to the same size as determined by the max_{depth} parameter. To access the complete string we use the ranges of nodes that correspond to the position of the complete string in the list of strings.

However, for prefixes larger than threshold that share too many query suggestions, the second level processing may become slow. In the worst case, the number of strings to be searched for in the second level may be $O(n)$, where n is the number of suggestions in the dataset, which means performing a linear sequential search in the whole dataset. While this worst case is not expected to occur in practice, this theoretical limitation is useful to explain how slow might be the LD method if the query contains a prefix that has a high number of occurrences in the dataset. Further, the max_{depth} threshold may vary in different datasets and a poor choice of this parameter may result either in a larger number of nodes in the TRIE or in higher query processing times.

4.2.2 Limited Slot Method

In the two-level approach in which we use the Limited Depth method it is necessary to set a fixed size for the size of the prefix to be indexed. This size can vary in different dataset and a poor choice of prefix size can cause some disadvantages. For example, if the size of the chosen prefix is too small, few nodes will be indexed in the TRIE, however, the list in the second level will be very large. A search for results on the second level in a very large list can cause slowness due to the nature of the sequential search. In an attempt to solve the performance limitations of LD method, we studied a second alternative for use the two-level approach, called Limited Slot or just LS. The idea is to determine the stopping criteria to go to the second level based on the number of nodes to be searched in the second level, instead of fixing a maximum depth for the nodes in the TRIE.

When creating the TRIE in this second approach, we use the parameter *max second level slot size* (max_{slot}), which determines the maximum size of query suggestions associated to a leaf node of the TRIE. In this method the indexing for a string stop with a few letters indexed, according to the thresholds configured, but as the number of query suggestions that share the same prefix increases, expansions in the TRIE occurs until that the parameter max_{slot} is respected. For instance, in the **Figure** 11 we have $max_{slot} = 2$. Initially we index the string “semicircle” and as the threshold max_{slot} is not exceeded, only a single node is added to the TRIE. Next, we index the string “semifinalist”, in which it shares the same prefix as the previously added string, but only the range of node $N1$ is updated, as both strings do not exceed $max_{slot} = 2$. Finally, we index the string “semiprofessional”, which has a similar prefix with the words “semicircle” and “semifinalist”. When we update the node interval, this size exceeds the threshold 2, so we must expand the strings until we have a leaf node interval smaller or equal to $max_{slot} = 2$. In this way, we make 4 expansions until we reach the nodes with intervals that are within our threshold. Notice that query suggestions that share large prefixes with several other query suggestions tend to have larger paths indexed in the resulting TRIE, since now we limit the cost for second level, instead of limiting the size of the paths in the TRIE. We experimented with constant values to set the value of max_{slot}, so that the final time complexity for searching when using BEVA is maintained in LS.

For example, suppose that the parameter max_{slot} has the value 2. And we want to index the words: “child”, “childhoold”, “midday”, “midfield”, “midway”, “misunderstand”, “semicircle”, “semifinalist”, and “semiprofessional”. All queries that have a similar prefix with more than two queries suggestion, according to the value of max_{slot}, will have a larger amount of their characters indexed in the TRIE until their current prefix has a list in second level less than or equal to max_{slot}. For example, the words “midday”, “midfield” and “midway”, and also “semicirle”, “semifinalist” and “semiprofessional”, as shown in **Figure** 12.

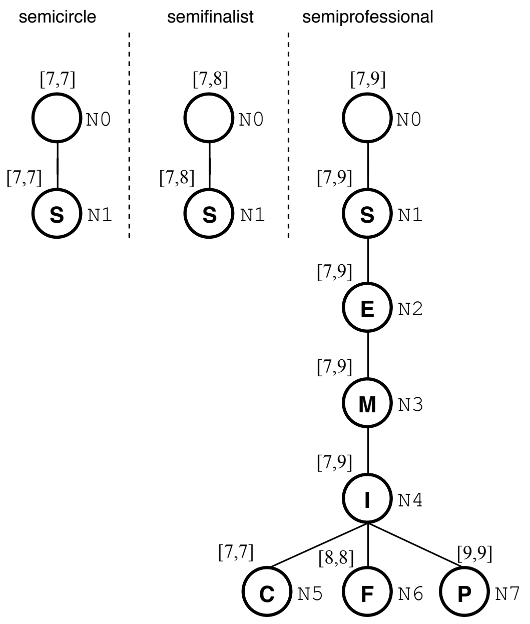


Figure 11: Example of the indexing construction in LS method with $max_{slot} = 2$.

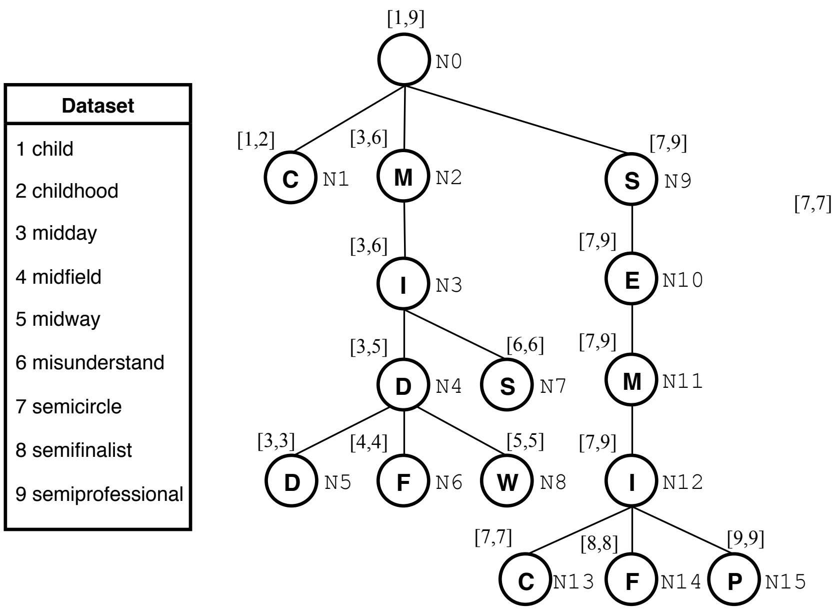


Figure 12: Indexing with Limited Slot Method.

The mechanism for indexing prefixes with varied size branches is very simple and is described in **Algorithm** 4. The function received as parameters: r, $\mathcal{Q}$, $\mathcal{Q}_{id}$ and max_{slot}, where this parameters represent the root node, query, query identifier and max slot from interval of the queries covered in the leaf nodes, respectively. The query suggestions is inserted character by character in line 5. In lines 7 and 8 the query suggestion ranges are updated in the node. In line 9, the value of the variable *depth* is incremented. In line 10, the list size in the second level is obtained for the current prefix query. In line 11, it is checked if this list is less or equal than the variable max_{slot}, in case true, the indexing is stopped. In line 13 the nodes is marked as leaf.

Algorithm 4 Insert Query Suggestions with LS Method

```
1: procedure InsertSuggestion($r$, $\mathcal{Q}$, $\mathcal{Q}_{id}$, $max_{slot}$)
2:     $node \leftarrow r$
3:     $depth \leftarrow 0$
4:     for $each\ c\ in\ \mathcal{Q}$ do
5:         $node \leftarrow insert(c)$    ▷ If $max_{slot}$ is exceeded, the expansion of strings occurs
6:         if $node.beginRange$ = -1 then
7:             $node.beginRange \leftarrow \mathcal{Q}_{id}$
8:         $node.endRange \leftarrow \mathcal{Q}_{id}$
9:         $depth \leftarrow depth + 1$
10:        $range \leftarrow node.endRange - node.beginRange + 1$
11:        if $range \leq max_{slot}$ then
12:            break
13:    $node.isLeaf \leftarrow$ True
```

In the second level a new search is performed for query suggestions that are similar to the prefix query typed by the user. We use the boundary active nodes results from the search in the first level to create the list of candidate results that should be inspected in the second level. Whenever a leaf node in the TRIE becomes a boundary active node, we extract the range of query suggestions covered by it and associate the edit vector automata (EVA) state of such node to each of the query suggestion found, inserting these suggestion in the list of candidates to be inspected at the second level of query processing. We then continue process new characters typed by the user, always updating the list of boundary active nodes and sequentially evaluating all candidate suggestions using the EVA states inferred from the first level. Finally, query suggestions that have edit distance within the desired threshold are computed as responses to the query. The architecture of the two-level methods presented can be seen in detail in **Figure** 13.

4.2.3 Query Processing with Two-Level Approach

In this section we explain in more details the query processing with the first and second level and the incremental computation of boundary active nodes or word active nodes. BEVA is used on the first level to process queries and its edit vector automata is reused on the second level to perform the edit distance calculation between words and prefix query. We use the edit vector automata to drive the traversal on the TRIE. Hence, an active node of the query is always associated with a state in the edit vector automata. Initially, the only active node is the root node of the TRIE, associated with the initial state S_0.

The adaptation of the *maintain* function proposed in Zhou et al. (2016) is shown in the **Algorithm** 5. This function takes $c, |\mathcal{P}|, \mathcal{A}$, and $\mathcal{W}$ as parameters, which represents the current character from prefix query, prefix query length, list of active nodes and list of

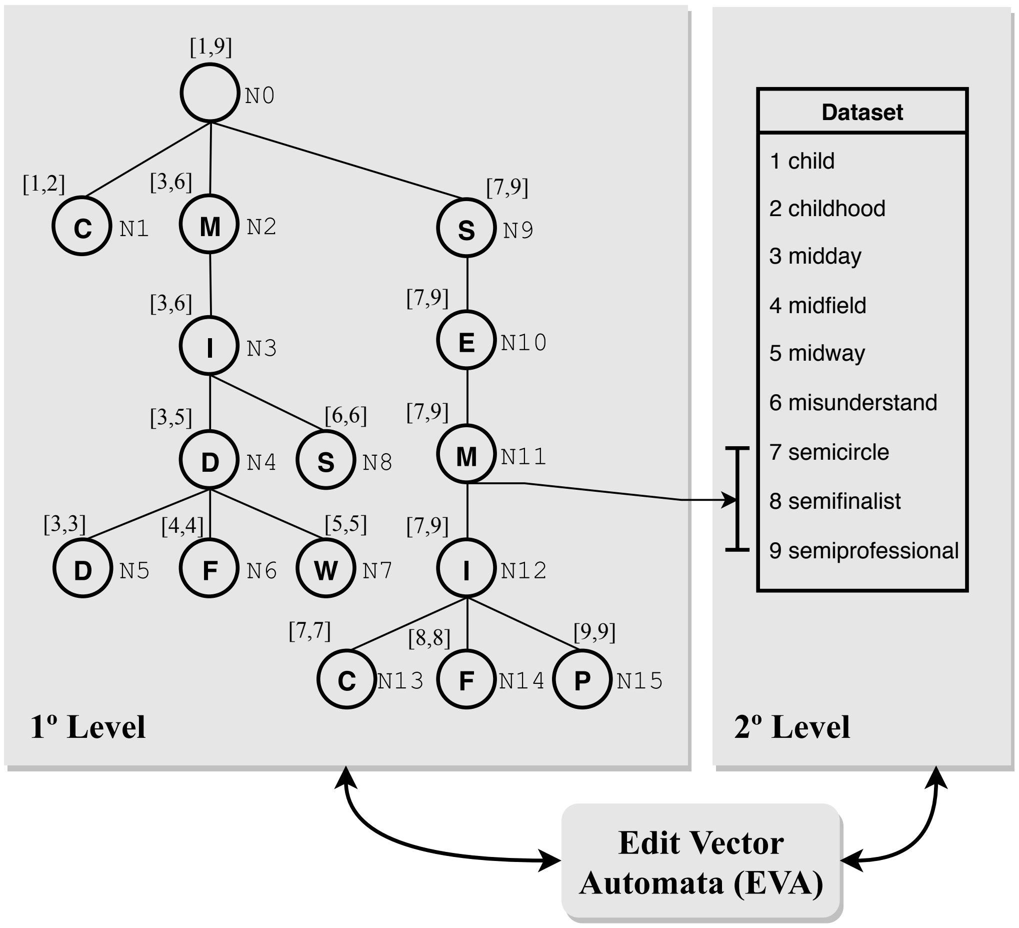


Figure 13: Architecture of our proposed methods using the two-level approach. On the left side we have our first level with part of the strings indexed in the TRIE, the size of the string to be indexed depends of the configured parameter. Each node in the tree has an interval referring to the complete strings that this node covers. When we reach a given node, we take its range and directly access the complete string in the list of strings. In this index we run the BEVA with adaptations. On the right side of the figure is our list of complete strings in lexicographic order, in which we obtain the complete string from the first level. We run the also the second level in this list of complete strings, but only in subsets covered by active nodes or covered by active word nodes. At the bottom of the figure we have the EVA structure that allows the efficient the edit distance calculation between two strings by storing states associated with the active nodes. Both levels use this structure.

word active nodes, respectively. Initially, in line 2, the global bitmaps are updated. When the prefix query length is 1, the only active node is the root node associated with the initial state $\mathcal{S}_I$, as described in lines 3 and 4. The search for new active nodes only starts when $|\mathcal{P}| \geq \tau$, τ being the *edit distance threshold*, and this is checked on line 5. In line 6 and 7 are instantiated the list of active nodes and the list of word active nodes, both empty. If the list of word active nodes from the previous prefix query is not empty, the Algorithm 8 is called to find new word active nodes from word active nodes from previous prefix query and these nodes are concatenated in the new list of word active nodes, as

Algorithm 5 Process prefix query

```
1: procedure MAINTAIN($c, |\mathcal{P}|, \mathcal{A}, \mathcal{W}$)
2:     updateBitmap($c$)
3:     if $|\mathcal{P}| = 1$ then
4:         $\mathcal{A} \leftarrow \langle r, \mathcal{S}_0 \rangle$
5:     else if $|\mathcal{P}| \geq \tau$ then
6:         $\mathcal{A}' \leftarrow \varnothing$
7:         $\mathcal{W}' \leftarrow \varnothing$
8:         for each $\langle w, \mathcal{S} \rangle$ in $\mathcal{W}$ do
9:             $\mathcal{W}' \leftarrow \mathcal{W}' \cup$ findWordActiveNodes($|\mathcal{P}|, \langle w, \mathcal{S} \rangle$)
10:        for each $\langle n, \mathcal{S} \rangle$ in $\mathcal{A}$ do
11:            if $n.isLeaf$ then
12:                $\mathcal{W}' \leftarrow \mathcal{W}' \cup$ findWordActiveNodes($|\mathcal{P}|, \langle n, \mathcal{S} \rangle$)
13:            else
14:                $\mathcal{A}', \mathcal{W}' \leftarrow \mathcal{A}', \mathcal{W}' \cup$ findActiveNodes($|\mathcal{P}|, \langle n, \mathcal{S} \rangle$)
15:        $\mathcal{A} \leftarrow \mathcal{A}'$
16:        $\mathcal{W} \leftarrow \mathcal{W}'$
17:    return $\mathcal{A}, \mathcal{W}$
```

can be seen in lines 8 and 9. In line 10, the iteration in the boundary active nodes of the previous prefix query in the first level is performed. Finally, the **Algorithm** 6 is called to transverse its descendants and search new boundary active nodes. If the processing at the first level reaches a leaf node in the TRIE, then the **Algorithm** 7 is called to transfer the processing to the second level, because for such leaf node does not have children nodes but this leaf node can have an suffix in second level, so does not have all the answers in first level and for this reason the second level is triggered in order to get all the results and the results are concatenated in the new lists of active nodes and word active nodes and returned, as shown in lines 10 to 15.

The **Algorithm** 6 is derived from **Algorithm 2: FindActive** proposed by Zhou et al. (2016). The difference is that if a given node is a leaf node, the processing that started at the first level will be transferred to the second level (both processes will happen together in case of LS method). The answers are computed in the findWordActiveNodes and findActiveNodes functions independently, but without duplicates.

In the **Algorithm** 7, the processing will be performed from the list of global queries determined by the range contained in the active node of the previous prefix query, lines 3, 4 and 5. This algorithm is only used when we are processing at the first level and we transfer the processing to the second level. If we find any word active nodes in this step it will be processed in the next prefix query directly by the Algorithm 8 called from *Maintain* algorithm 5. This processing is performed incrementally, as in the first level, using the active nodes of the previous prefix query typed by the user. For instance, if the user typed a query “abcd”, and then type an “e” to complement it, the “abcde” prefix will

Algorithm 6 Find active nodes

```
1: procedure FindActiveNodes($|\mathcal{P}|, \langle n, \mathcal{S} \rangle$)
2:     $\mathcal{A} \leftarrow \varnothing$
3:     $\mathcal{W} \leftarrow \varnothing$
4:     $level \leftarrow n.level + 1$
5:     for each child n' of n do
6:         $\mathcal{B}_{n'} \leftarrow$ buildBitmap($|\mathcal{P}|, level, n'.char$)
7:         $\mathcal{S}' \leftarrow f(\mathcal{S}, \mathcal{B}_{n'})$
8:         if $\mathcal{S}' \neq \mathcal{S}_\perp$ then
9:             if $\mathcal{S}'[|\mathcal{P}| - level] \leq \tau$ then
10:                $\mathcal{A} \leftarrow \mathcal{A} \cup \langle n', \mathcal{S}' \rangle$
11:            else if n'.isLeaf then
12:                $\mathcal{W} \leftarrow \mathcal{W} \cup$ findWordActiveNodes($|\mathcal{P}|, \langle n', \mathcal{S}' \rangle$)
13:            else
14:                $\mathcal{A}, \mathcal{W} \leftarrow \mathcal{A}, \mathcal{W} \cup$ findActiveNodes($|\mathcal{P}|, \langle n', \mathcal{S}' \rangle$)
15:    return $\mathcal{A}, \mathcal{W}$
```

Algorithm 7 Find word active nodes

```
1: procedure findWordActiveNodes($|\mathcal{P}|, \langle n, \mathcal{S} \rangle$)
2:     $\mathcal{W} \leftarrow \varnothing$
3:     for each record in records[n.beginRange, n.endRange] do
4:         $w' \leftarrow \langle record.id, n.level \rangle$
5:         $\mathcal{W} \leftarrow \mathcal{W} \cup$ CalculateEDWords($|\mathcal{P}|, \langle w', \mathcal{S} \rangle$)
6:     return $\mathcal{W}$
```

be computed taking the active nodes computed when processing the query up to "abcd".

Algorithm 8 Calculate edit distance and save new word active nodes

```
1: procedure CalculateEDWords($|\mathcal{P}|, \langle w, \mathcal{S} \rangle$)
2:     $\mathcal{W} \leftarrow \varnothing$
3:     while $n.level < |records[w.id]|$ do
4:         $n.level \leftarrow n.level + 1$
5:         $\mathcal{B}_{n'} \leftarrow$ buildBitmap($|\mathcal{P}|, n.level, records[w.id][level]$)
6:         $\mathcal{S}' \leftarrow f(\mathcal{S}, \mathcal{B}_{n'})$
7:         if $\mathcal{S}' \neq \mathcal{S}_\perp$ then
8:             if $\mathcal{S}'[|\mathcal{P}| - n.level] \leq \tau$ then
9:                 $w' \leftarrow \langle w.id, n.level \rangle$
10:                $\mathcal{W} \leftarrow \mathcal{W} \cup \langle w', \mathcal{S}' \rangle$
11:                break
12:    return $\mathcal{W}$
```

The **Algorithm** 8 shows how the new word active nodes are computed. The processing is also performed incrementally, and the edit vector automata proposed by Zhou et al. (2016) is used to calculate the edit distance between two strings. The processing is similar to the Algorithm 6, with the difference that we have words instead of prefixes queries from TRIE. When the edit distance in the new state is within the *edit distance* *threshold*, then a new word active node is computed as a response, and the processing terminates as described in lines 8 to 11. Otherwise, a new character must be checked.

4.2.3.1 Running a Query with LS method

Here we will show an example of execution. Consider that the prefix query is “midd”, the edit distance threshold is $\tau = 1$ and the strings indexed in the TRIE are: “child”, “childhood”, “midday”, “midfield”, “midway”, “misunderstand”, “semicircle”, “semifinalist“ and “ semiprofessional“, as shown in **Figure** 14. We will show query processing for LS method and the value from max_{slot} is 3. For the prefix query “midd” the processing was performed at the first level with the BEVA algorithm and the boundary active node computed is only n_4, as shown in gray in **Figure** 14. This boundary active node is associated with the state of the automaton as follows: $\{[n_4, f(S_0, 011) = S_0]\}$, as shown in the **Table** 5 until step 5. When the prefix query is incremented and a character “a” is appended to the previous prefix query, a new set of active nodes must be computed for the new prefix query “midda”. The computation of the new set will be performed incrementally starting the boundary active node of the previous prefix query.

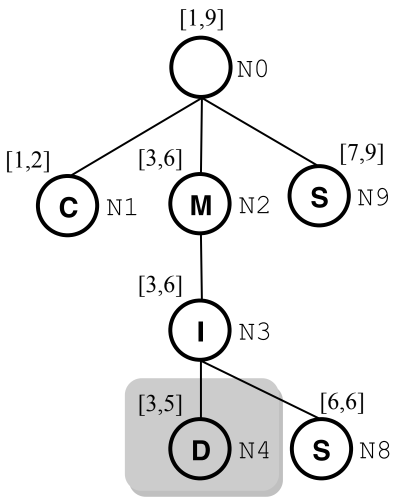


Figure 14: Representation of query processing using the LS method. Consider the prefix query “midd”. The boundary active prefix set is $\{n_4\}$.

The new set of active nodes is computed in **Algorithm** 5. The new set of active nodes of the prefix query “midda” is computed starting from node n_4. Thus, we process the node n_4, as the node n_4 is a leaf node, the **Algorithm** 7 is called to process the set of active nodes in the second level.

We obtain the range of ids covered by node n_4 in **Algorithm** 7. Such range can access the list of words: “midday”, “midfield”, and “midway” from the list of all words in the dataset. For each string in this list, we will calculate the edit distance between the string and the prefix query using the edit vector automata, through the **Algorithm** 8. The edit distances of the prefix query for each string are shown as follows: $ed(midda, midd) = 1$,

$ed(midda, midfi) = 2$ and $ed(midda, midwa) = 1$. Thus, the strings *midday* and *midway* analyzed are within the *edit distance threshold* and therefore, will be part of the new list of active nodes, the called word active nodes, in which each node stores the word and the state associated in the automaton.

Step	$\mathcal{P}$	**Boundary Active Node Set and Word Active Node Set**
1	ϵ	$\{[n_0, S_0]\}$
2	m	$\{[n_0, S_0]\}$
3	i	$\{[n_2, f(S_0, 010) = S_0]\}$
4	d	$\{[n_3, f(S_0, 010) = S_0]\}$
5	d	$\{[n_4, f(S_0, 011) = S_0]\}$
6	a	$\{[id_3, f(S_0, 110) = S_0], [id_5, f(S_4, 111) = S_5]\}$
7	y	$\{[id_3, f(S_0, 001) = S_0], [id_5, f(S_4, 000) = S_4]\}$

Table 5: Query processing with LS method for prefix query “midday”. The boundary active nodes set obtained in the first level, shown in step 1 to 4. And the word active node set obtained in the second level, shown in step 5 to 7.

Finally, the character “y” will be added to the previous prefix query, forming the new prefix query “midday”. Such prefix query will be initially processed in the **Algorithm** 5 and then it will be verified that the list of nodes from the first level is empty, so the **Algorithm** 7 will be called direct in line 9 of the **Algorithm** 5. And then the process for computing the edit distances will be repeated starting from list of word active nodes of the previous prefix query. After doing the edit distance calculations, the strings “midday” and “midway” will be added to the new list of word active nodes. The complete process is show in the **Table** 5.

5 Results

In this chapter, we analyze and report the results of the study of parameters for the proposed methods, their advantages, disadvantages and scenarios in which they can be better used. We also compare our proposed methods with the baseline methods and we proposed improvements to such methods in literature. At the end of these experiments, we hope to answer the following question: (1) The alternative implementations of BEVA are able to reduce the memory requirements for performing query autocompletion ? (2) What is the impact of the new index building strategies proposed (laat) in overall performance of the query autocompletion methods studied ? (3) How the two alternative two-level methods proposed perform in practice ?

5.1 Experiments Setup

5.1.1 Experienced Methods

The following methods are analyzed in the experiments:

- **META** It is a method that uses indexing in compact trees, being able to reduce redundant calculations in query autocompletion processing. Deng et al. (2016).
- **BEVA** is method based on the boundary maintenance strategy and edit vector automata Zhou et al. (2016).
- **LD** Is BEVA with two-level approach and branches from TRIE with limited depth.
- **LS** Is BEVA with two-level approach and leaf nodes from TRIE with references to lists in the second level with limited size or slot.

5.1.2 Experimental Environment and Datasets

All the experiments were carried out on a machine with Intel® Xeon E5-4617 2.90 GHz and 64GB RAM, running Ubuntu 18.04.1 LTS. We implemented all the algorithms in C++ and compiled with gcc 7.4.0.

We select three publicly available dataset:

- **USADDR**[1] Is a set with 10 million addresses and places of the United States, extracted from the collection *SimpleGeo CC0*. From this database, the place names

[1] http://archive.org/download/2011-08-SimpleGeo-CC0-Public-Spaces/

were extracted and inserted into a single text file, where each row is equivalent to one item.

- **MEDLINE**[2] Is the main bibliographic database of the National Library (NLM), which contains more than 14 million references to articles in health science journals, with emphasis on biomedicine. MEDLINE is the online version of MEDLARS (System of Analysis and Recovery of Medical Literature), originated in 1964. The title of each article was extracted, removing duplications. Each title extracted is one item in the file.

- **DBLP**[3] About 4.5 million computer science publication records. For the experiments, we extracted only title of each publication and output it line-by-line into a file.

Dataset	**Items**	**Words**	**Avg len words**	**Avg len items**
USADDR	10,251,121	32,408,910	5.8	20.7
MEDLINE	14,419,464	168,912,499	6.7	90.1
DBLP	4,544,402	43,937,367	6.9	75.7

Table 6: Dataset Statistics

5.1.3 Settings

For MEDLINE, USADDR and DBLP we follow Chaudhuri and Kaushik (2009) to randomly sample 1,000 strings as prefix queries. Such prefix queries received a number of errors that are within range $0 - \tau$ at any position in the prefix query, where the number of errors and the error position in the prefix query were randomly chosen. The query processing and memory consumption were experimented in separated executions because the method to collect memory consumption causes overhead in query processing time.

We measure (1) the **query response time**, which consists of the **searching time** (separated at 1^o and 2^o level for proposed methods) and **result fetching time** (2) **TRIE index size** and (3) the memory consumption (of each parameter experienced for proposed methods). All the measures are averaged over 1,000 prefix queries. The query response time was experimented with prefix query of sizes 5, 9, 13 and 17. The query response time was experimented with all datasets described in this dissertation for edit distance threshold 1, 2, 3 and 4. Edit distance thresholds greater than 4 are not common in real systems, therefore, we not have performed with such edit distance threshold.

[2] `https://www.nlm.nih.gov/databases/download/pubmed_medline.html`
[3] `https://dblp.uni-TRIEr.de/faq/How+can+I+download+the+whole+dblp+dataset`

5.2 Improving Index Building

We decided to evaluate the TRIE build in the same direction that the strings are searched in query processing. Our hypothesis with this is that the nodes in the TRIE will be indexed in nearby memory addresses. Thus, we expect that the query processing times will decrease due to the better performance of the cache policies of the machine. We performed experiments to compare the performance of BEVA and also in the proposed methods - which use the BEVA in the first level when running over TRIES built in the two strategies, key-at-a-time (kaat) and level-at-a-time (laat). We have the processing time for prefix queries sizes 5, 9, 13 and 17, varying the edit distance from 1 to 3 in the USADDR, MEDLINE and DBLP datasets.

In Table 7, for USADDR dataset and $\tau = 1$, we have a decrease of approximately 50% in the processing time of prefix queries of sizes 5, 9, 13 and 17 when the TRIE Built is performed in laat strategy for the BEVA. When we increase the τ to 2, this time difference is almost three times smaller when compared to the processing time when we use the kaat strategy in the TRIE building and this difference remains when we analyze the results for $\tau = 3$. In the DBLP dataset, the difference is even greater than in the USADDR dataset, for prefix queries of sizes 5, 9, 13 and 17 and $\tau = 1$ we have the Beva-laat method more than twice faster than the BEVA-kaat method, and when we increase the τ to 3, for example, we have the BEVA-laat method more than three times faster than the BEVA-kaat method. A really considerable decrease in time using a state-of-the-art method. In the comparisons with the baselines methods (Section 5.3) we will use the BEVA-laat method.

	$\tau = 1$				$\tau = 2$				$\tau = 3$			
	$\|\mathcal{P}\| = 5$	$\|\mathcal{P}\| = 9$	$\|\mathcal{P}\| = 13$	$\|\mathcal{P}\| = 17$	$\|\mathcal{P}\| = 5$	$\|\mathcal{P}\| = 9$	$\|\mathcal{P}\| = 13$	$\|\mathcal{P}\| = 17$	$\|\mathcal{P}\| = 5$	$\|\mathcal{P}\| = 9$	$\|\mathcal{P}\| = 13$	$\|\mathcal{P}\| = 17$
USADDR												
BEVA-laat	0.2	0.2	0.2	0.2	2.4	3.0	3.1	3.1	12.7	20.4	20.7	20.8
BEVA-kaat	0.4	0.4	0.5	0.5	6.7	7.6	7.6	7.7	36.2	48.7	49.0	49.1
MEDLINE												
BEVA-laat	0.2	1.5	1.5	1.5	1.8	2.0	2.1	2.1	6.6	9.4	9.5	9.6
BEVA-kaat	0.6	2.3	2.3	2.3	7.0	7.5	7.6	7.7	28.3	34.5	34.7	34.9
DBLP												
BEVA-laat	0.2	0.2	0.2	0.2	1.6	1.9	1.9	1.9	5.8	8.2	8.3	8.3
BEVA-kaat	0.5	0.5	0.5	0.5	5.0	5.4	5.4	5.5	21.0	25.7	25.9	25.9 5

Table 7: Query processing time (ms) of the BEVA with the TRIE Built in laat and kaat strategies with prefix queries of sizes 5, 7, 9, 13 and 17 and τ from 1 to 3.

The reason for this significant decrease in times when using BEVA-laat is due to the lower number of cache misses that occur when processing with BEVA-laat when compared to BEVA-kaat. In Table 8 we present the numbers of cache misses at level 1 and the last level from machine cache hierarchy, which are shown in columns D1 and DL, respectively. We can see a reduction in the number of caches misses at level 1 from 652,070,680 to 439,034,841 and at the last level from 469,774,543 to 231,654,885. A considerable decrease in the number of cache misses, being these cache misses responsible for the time difference between the kaat and laat strategies. This cache misses numbers represent only a simulation of memory cache misses, simulated by the CacheGrind program [4], it simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). In a real scenario, the number of cache misses may be even higher for others cache levels, however we were unable to do this full test.

	Memory Cache Misses	
	D1	**DL**
BEVA-kaat	652,070,680	469,774,543
BEVA-laat	439,034,841	231,654,885

Table 8: Simulation of memory cache misses collected by the CacheGrind program in the USADDR dataset for $\tau = 3$.

In Tables 9 and 10 we show the experiments result for max_{depth} with values 6, 8, 10, 12 and 14 and max_{slot} with values 1, 2, 4, 6, 8 and 10, for LD and LS methods, respectively. We quickly realized that unanimously the query processing times is shorter when the TRIE is built using laat approach when compared to kaat strategy. For example, for $\tau = 3$, $|\mathcal{P}| = 5$ and MEDLINE dataset, the query processing times for the LD6 method were 0.4 ms and 0.1 ms for the index built in kaat and laat strategies, respectively. As we increase the value of max_{depth} this trend continues. When we increase the prefix query size to 17 or the edit distance threshold, we also see the same trend in the LD method (see Table 9).

	$\tau = 1$								$\tau = 2$								$\tau = 3$							
	$\|\mathcal{P}\| = 5$		$\|\mathcal{P}\| = 9$		$\|\mathcal{P}\| = 13$		$\|\mathcal{P}\| = 17$		$\|\mathcal{P}\| = 5$		$\|\mathcal{P}\| = 9$		$\|\mathcal{P}\| = 13$		$\|\mathcal{P}\| = 17$		$\|\mathcal{P}\| = 5$		$\|\mathcal{P}\| = 9$		$\|\mathcal{P}\| = 13$		$\|\mathcal{P}\| = 17$	
	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat
USADDR																								
LD6	0.3	0.2	0.7	0.6	0.9	0.7	0.9	0.8	4.8	2.7	6.9	4.9	7.2	5.2	7.2	5.2	20.7	17.2	42.7	39.8	43.3	40.5	43.4	40.6
LD8	0.4	0.2	0.4	0.3	0.6	0.4	0.6	0.4	5.8	2.6	6.6	3.4	6.8	3.6	6.9	3.7	26.0	13.4	35.5	22.3	36.2	23.1	36.3	23.1
LD10	0.4	0.2	0.4	0.3	0.5	0.3	0.5	0.3	6.3	2.6	7.0	3.2	7.2	3.4	7.2	3.4	31.3	13.3	41.8	21.0	42.4	21.6	42.5	21.7
LD12	0.4	0.2	0.4	0.2	0.5	0.3	0.5	0.3	6.5	2.6	7.2	3.2	7.3	3.3	7.3	3.4	33.0	13.4	44.0	21.0	44.3	21.4	44.4	21.5
LD14	0.4	0.2	0.5	0.2	0.5	0.2	0.5	0.3	6.7	2.6	7.5	3.2	7.5	3.3	7.6	3.3	34.3	13.5	45.8	21.0	46.0	21.3	46.1	21.4
MEDLINE																								
LD6	0.4	0.2	2.9	2.7	4.5	4.3	4.9	4.8	3.7	2.1	9.0	7.4	11.4	9.7	12.0	10.4	12.8	11.5	35.0	34.1	38.8	38.0	40.0	39.2
LD8	0.5	0.2	1.0	0.7	2.7	2.5	3.1	2.9	4.4	1.9	5.2	2.8	7.3	4.9	7.9	5.5	14.8	7.6	20.1	12.8	23.0	15.7	24.2	16.9
LD10	0.5	0.2	0.6	0.3	1.5	1.2	1.9	1.6	4.8	1.9	5.2	2.3	6.3	3.3	6.9	3.9	18.3	7.6	22.0	10.7	23.3	12.1	24.5	13.2
LD12	0.5	0.2	0.6	0.3	0.8	0.5	1.2	0.9	5.0	1.9	5.4	2.2	5.7	2.5	6.2	3.1	19.8	7.5	24.0	10.5	24.4	11.0	25.3	11.9
LD14	0.5	0.2	0.6	0.3	0.6	0.3	0.9	0.6	5.3	1.9	5.7	2.2	5.8	2.3	6.1	2.6	20.6	7.5	25.0	10.5	25.1	10.7	25.6	11.2
DBLP																								
LD6	0.3	0.2	1.0	0.8	1.3	1.2	1.4	1.2	2.9	1.8	4.6	3.6	5.2	4.1	5.3	4.2	9.0	8.2	17.8	17.2	18.8	18.3	19.0	18.5
LD8	0.4	0.2	0.5	0.3	0.8	0.6	0.9	0.7	3.5	1.7	3.9	2.1	4.3	2.6	4.5	2.7	11.3	6.4	14.7	9.8	15.5	10.6	15.8	10.8
LD10	0.4	0.2	0.4	0.2	0.6	0.4	0.7	0.5	3.9	1.7	4.1	2.0	4.4	2.2	4.5	2.4	14.4	6.4	17.3	8.8	17.7	9.3	18.0	9.6
LD12	0.4	0.2	0.4	0.2	0.5	0.3	0.6	0.4	4.0	1.7	4.3	1.9	4.4	2.0	4.5	2.2	16.0	6.4	19.2	8.8	19.4	9.1	19.6	9.3
LD14	0.4	0.2	0.4	0.2	0.4	0.2	0.5	0.3	4.1	1.7	4.4	1.9	4.4	2.0	4.5	2.1	16.7	6.5	20.1	9.0	20.2	9.2	20.4	9.3

Table 9: Query processing time (ms) of the LD with the TRIE Built in laat and kaat strategies with queries of sizes 5, 9, 13 and 17 and max_{depth} with values 6, 8, 10, 12 and 14 for τ varying from 1 to 3 in the USADDR, MEDLINE and DBLP datasets.

The difference between query processing time in the TRIE built with the laat and kaat strategies increases when we increase the amount of information in the first level, as when we increase the value of max_{depth} or decrease the value of max_{slot} for the LD and

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

	$\tau = 1$								$\tau = 2$								$\tau = 3$							
	$\lvert\mathcal{P}\rvert = 5$		$\lvert\mathcal{P}\rvert = 9$		$\lvert\mathcal{P}\rvert = 13$		$\lvert\mathcal{P}\rvert = 17$		$\lvert\mathcal{P}\rvert = 5$		$\lvert\mathcal{P}\rvert = 9$		$\lvert\mathcal{P}\rvert = 13$		$\lvert\mathcal{P}\rvert = 17$		$\lvert\mathcal{P}\rvert = 5$		$\lvert\mathcal{P}\rvert = 9$		$\lvert\mathcal{P}\rvert = 13$		$\lvert\mathcal{P}\rvert = 17$	
	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat	kaat	laat
USADDR																								
LS-1	0.4	0.2	0.5	0.3	0.5	0.3	0.5	0.3	7.7	4.4	9.0	5.6	9.1	5.7	9.2	5.7	49.7	34.3	70.3	52.4	70.9	52.9	71.0	53.0
LS-2	0.4	0.2	0.5	0.3	0.5	0.3	0.5	0.4	7.8	5.1	9.3	6.5	9.4	6.6	9.4	6.6	53.2	41.2	75.6	61.8	76.2	62.4	76.3	62.4
LS-4	0.4	0.3	0.5	0.4	0.5	0.4	0.5	0.4	8.2	5.9	9.8	7.4	9.9	7.5	9.9	7.5	56.4	47.2	79.6	69.3	80.2	69.9	80.3	70.0
LS-6	0.4	0.3	0.5	0.4	0.5	0.4	0.5	0.4	8.4	6.4	10.0	7.9	10.1	8.1	10.1	8.1	58.6	50.3	82.1	72.7	82.7	73.3	82.8	73.4
LS-8	0.4	0.3	0.5	0.4	0.5	0.4	0.5	0.4	8.5	6.6	10.1	8.2	10.3	8.3	10.3	8.3	59.2	52.1	82.5	74.5	83.1	75.1	83.1	75.1
LS-10	0.4	0.3	0.5	0.4	0.5	0.4	0.6	0.4	8.7	6.9	10.3	8.5	10.5	8.6	10.5	8.6	59.8	53.5	83.0	75.9	83.5	76.5	83.6	76.6
MEDLINE																								
LS-1	0.5	0.3	0.7	0.4	0.7	0.4	0.7	0.4	6.8	4.3	7.6	5.0	7.7	5.0	7.8	5.1	36.3	27.6	44.9	35.1	45.3	35.5	45.4	35.6
LS-2	0.5	0.3	0.6	0.4	0.7	0.4	0.7	0.5	7.1	5.0	7.8	5.7	7.9	5.8	8.0	5.8	39.3	33.0	48.5	41.5	48.9	41.8	49.1	42.0
LS-4	0.5	0.3	0.6	0.4	0.6	0.4	0.6	0.4	7.5	5.8	8.3	6.6	8.4	6.7	8.5	6.8	42.4	37.4	52.1	46.4	52.5	46.8	52.7	47.0
LS-6	0.5	0.3	0.6	0.4	0.6	0.4	0.6	0.5	7.8	6.3	8.6	7.0	8.7	7.1	8.8	7.2	43.5	39.4	53.3	48.7	53.7	49.1	53.9	49.3
LS-8	0.5	0.4	0.6	0.4	0.6	0.5	0.7	0.5	8.0	6.6	8.9	7.4	9.0	7.5	9.0	7.6	44.2	40.5	54.0	49.8	54.4	50.2	54.6	50.5
LS-10	0.5	0.4	0.6	0.4	0.6	0.5	0.7	0.5	8.2	6.8	9.0	7.6	9.1	7.7	9.2	7.8	44.4	41.0	54.3	50.4	54.7	50.8	54.9	51.1
DBLP																								
LS-1	0.4	0.3	0.5	0.3	0.5	0.3	0.5	0.3	5.6	4.0	6.2	4.6	6.2	4.6	6.3	4.7	32.1	25.8	39.5	32.5	39.8	32.8	39.9	32.9
LS-2	0.4	0.3	0.5	0.3	0.5	0.4	0.5	0.4	6.1	4.8	6.7	5.4	6.8	5.5	6.9	5.5	34.2	30.8	42.1	38.2	42.4	38.5	42.5	38.6
LS-4	0.4	0.3	0.5	0.4	0.5	0.4	0.5	0.4	6.6	5.6	7.2	6.2	7.3	6.3	7.4	6.4	36.1	34.0	44.0	41.7	44.3	41.9	44.4	42.1
LS-6	0.4	0.4	0.5	0.4	0.5	0.4	0.6	0.5	6.9	6.0	7.6	6.7	7.6	6.8	7.7	6.8	36.7	34.7	44.5	42.3	44.8	42.6	44.9	42.7
LS-8	0.5	0.4	0.5	0.4	0.5	0.4	0.6	0.5	7.0	6.3	7.7	6.9	7.8	7.0	7.8	7.1	36.6	35.0	44.3	42.5	44.6	42.8	44.7	42.9
LS-10	0.5	0.4	0.5	0.4	0.6	0.5	0.6	0.5	7.1	6.5	7.8	7.1	7.9	7.2	7.9	7.2	36.2	34.8	43.8	42.2	44.1	42.5	44.2	42.6

Table 10: Query processing time (ms) of the LS with the TRIE Built in laat and kaat strategies with queries of sizes 5, 9, 13 and 17 and max_{slot} with values 1, 2, 4, 6, 8 and 10 for τ varying from 1 to 3 in the USADDR, MEDLINE and DBLP datasets.

LS methods, respectively. This is because, in the first level, node references are handled more frequently, such references receive an addresses in memory during indexing. When we build the TRIE with the kaat strategy, theses addresses of the nodes in memory are more distant between each node, on the other hand, when we build the TRIE with the laat strategy, the addresses of the nodes in memory are close. This proximity between references facilitates the cache policies of the machine during query processing. And this is more evident when: (1) we increase the processing volume, that is, when we increase the value of τ, (2) the index in the first level is higher, that is, there are more references of nodes in memory.

Thus, we conclude that the index built with the laat strategy is better than index built with the kaat strategy. This implementation is relatively simple and does not affect the size of the index in memory, the change is only in the location of the node addresses in memory, which it also given the significant gain in performance of laat TRIE building. We have adopted the laat TRIE building strategies for all the remaining experiments.

5.3 Selecting Parameters for the Proposed Methods

In this section, we analyze the selection of parameters for the proposed methods. The LD and LS methods use the two-level approach, which consists of the query processing in two step. Both LD and LS methods use the BEVA for query processing in the first level. In the second level, sequential search is performed on the first level results and reuses the edit vector automaton to calculate the edit distance between two sequences.

LD was analyzed with various thresholds to check query response time with different amounts of characters indexed in the first level. This threshold is called max_{depth} with values varying from 4 to 15. Values of max_{depth} less than 4 not were analyzed because the query processing occurs frequently only in the second level. In same way, values of max_{depth} greater than 15 not were analyzed because, in this scenario, the query processing occurs frequently only in the first level and the aim here is analyze the query processing using the first and second level.

LS allows control the amount of information added to the second level. This control is performed by a threshold called max_{slot}, which consists of the string interval length to be processed. The max_{slot} threshold avoids the size of the list of words processed in the second level to become large, ie, if the list size exceed this threshold, the indexing process does not stop until we reach a level in the TRIE where it is not exceeded. We experimented with values 1, 2, 4, 6, 8 and 10. Values of max_{slot} greater than 10 not were analyzed because we don't want long processing in the second level.

The aim in this section is to study different values for max_{depth} and max_{slot}, their advantages and disadvantages and try to choose the best value of max_{depth} and max_{slot} for LD and LS methods, respectively, considering the query processing time and memory consumption used for each dataset and edit distance threshold.

5.3.1 Query Response Time

Query response time is a very important metric for error-tolerant query autocompletion algorithms. This metric helps us to identify fast and non-fast algorithms and thus propose the best user experience in the computer systems, avoiding delays in the interactive search session and showing responses at each user keystroke in a fluid way. The query response times may vary depending of the prefix query characteristic, an example is the prefix query length, ie, short prefix queries are processed fast when computing the results, but present slower fetching times, on the other hand, long prefix queries are processed fast when computing the search results and when fetching results.

We measure the searching time separated by levels when experimenting the two-level strategies. The sum of values of the columns 1º and 2º in the Tables of the experiments represent the searching time, fetch column represents the result fetching time and total column represents the query response time (searching and fetching together). We present the times separated by level to have a better perception of the time spent in each level. In this way, we quickly see that there is a big difference in the query processing times spent at each level as we vary the values of max_{depth} for the LD method. The processing times spent on small values of max_{depth} such as 4, 5 and 6 is very high because small indexed prefixes in TRIE tend to have large lists in the second level, and processing for large lists in the second level is not efficient, as we can observe in Tables 11, 12 and 13 for LD4, LD5 and LD6 methods. This same behavior is not seen in prefix query length 5 in the LD5 and LD6 methods because the prefix query length is smaller than the TRIE level, so the processing occurs with less frequency in the second level or only in the first level.

	$\tau = 3$																
method	$\vert\mathcal{P}\vert = 5$				$\vert\mathcal{P}\vert = 9$				$\vert\mathcal{P}\vert = 13$				$\vert\mathcal{P}\vert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	20.5	136.5	118.6	275.7	23.8	184.4	0.7	208.8	23.8	185.0	-	208.8	23.8	185.1	-	208.9	2041.7
LD5	17.6	26.8	118.5	163.0	26.0	59.9	0.7	86.6	26.0	60.5	-	86.6	26.0	60.6	-	86.6	2047.5
LD6	14.2	3.0	119.2	136.4	23.1	16.8	0.7	40.5	23.1	17.4	-	40.6	23.1	17.5	-	40.6	2076.4
LD7	13.5	0.2	118.9	132.6	21.5	4.5	0.7	26.7	21.6	5.1	-	26.7	21.6	5.2	-	26.8	2138.1
LD8	13.4	-	118.5	131.9	21.1	1.2	0.7	23.0	21.2	1.8	-	23.1	21.2	1.9	-	23.2	2230.0
LD9	13.5	-	118.5	132.0	21.1	0.3	0.7	22.0	21.4	0.8	-	22.2	21.4	0.9	-	22.3	2360.1
LD10	13.3	-	118.5	131.8	20.9	-	0.7	21.6	21.2	0.4	-	21.7	21.2	0.5	-	21.7	2527.6
LD11	13.3	-	117.6	130.9	20.9	-	0.7	21.5	21.2	0.2	-	21.4	21.2	0.3	-	21.5	2729.9
LD12	13.4	-	117.7	131.1	21.0	-	0.7	21.7	21.3	0.1	-	21.4	21.3	0.2	-	21.5	2959.6
LD13	13.3	-	117.5	130.8	20.9	-	0.7	21.6	21.2	-	-	21.3	21.2	0.1	-	21.3	3207.6
LD14	13.5	-	120.3	133.7	21.0	-	0.7	21.7	21.3	-	-	21.4	21.4	0.1	-	21.4	3465.2
LD15	13.3	-	117.9	131.2	21.0	-	0.7	21.6	21.3	-	-	21.3	21.3	-	-	21.4	3725.2
LS-1	20.7	13.6	115.6	149.9	31.2	21.2	0.7	53.0	31.5	21.4	-	53.0	31.5	21.4	-	53.0	2867.8
LS-2	22.9	18.3	116.1	157.3	33.8	28.0	0.7	62.4	34.1	28.3	-	62.4	34.1	28.3	-	62.5	2486.8
LS-4	24.2	23.0	116.6	163.8	34.8	34.5	0.7	70.0	35.1	34.8	-	70.0	35.1	34.8	-	70.0	2290.0
LS-6	24.7	25.6	116.6	166.9	35.0	37.8	0.7	73.4	35.2	38.1	-	73.4	35.2	38.1	-	73.4	2218.6
LS-8	24.8	27.3	116.7	168.8	34.7	39.8	0.7	75.2	35.0	40.1	-	75.1	35.0	40.2	-	75.1	2180.1
LS-10	24.8	28.7	116.6	170.2	34.4	41.5	0.7	76.6	34.6	41.9	-	76.5	34.7	41.9	-	76.6	2155.7

Table 11: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 3$.

	$\tau = 3$																
method	$\vert\mathcal{P}\vert = 5$				$\vert\mathcal{P}\vert = 9$				$\vert\mathcal{P}\vert = 13$				$\vert\mathcal{P}\vert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	15.4	232.6	263.4	511.4	18.1	314.0	3.6	335.7	18.2	317.7	1.1	337.0	18.2	318.8	0.2	337.2	5905.6
LD5	10.3	35.3	259.7	305.3	14.2	84.0	3.7	101.9	14.3	87.6	1.1	103.1	14.3	88.7	0.2	103.3	5915.1
LD6	8.2	3.3	262.0	273.5	11.8	22.4	3.7	37.8	11.9	26.1	1.1	39.1	11.9	27.3	0.2	39.4	5928.5
LD7	7.7	0.2	268.1	276.0	11.0	6.8	3.7	21.5	11.1	10.2	1.1	22.4	11.1	11.3	0.2	22.6	5952.3
LD8	7.6	-	268.4	276.0	10.7	2.1	3.7	16.5	10.8	4.9	1.1	16.8	10.9	6.0	0.2	17.0	5987.8
LD9	7.6	-	264.9	272.4	10.6	0.4	3.7	14.7	10.7	2.5	1.1	14.3	10.8	3.6	0.2	14.6	6034.4
LD10	7.6	-	267.4	275.0	10.6	0.1	3.7	14.4	10.8	1.3	1.1	13.2	10.8	2.4	0.2	13.4	6094.2
LD11	7.6	-	265.7	273.3	10.7	-	3.6	14.3	10.8	0.7	1.1	12.6	10.9	1.7	0.2	12.7	6177.3
LD12	7.5	-	264.0	271.6	10.5	-	3.6	14.1	10.7	0.3	1.1	12.1	10.7	1.1	0.2	12.0	6287.7
LD13	7.6	-	265.1	272.7	10.6	-	3.7	14.2	10.8	0.1	1.1	12.0	10.8	0.7	0.2	11.8	6424.9
LD14	7.5	-	265.7	273.3	10.5	-	3.6	14.1	10.7	-	1.1	11.7	10.7	0.4	0.2	11.3	6592.0
LD15	7.7	-	265.0	272.7	10.6	-	3.7	14.3	10.8	-	1.1	11.9	10.9	0.2	0.2	11.3	6790.7
LS-1	14.6	12.9	260.4	288.0	18.8	16.3	3.6	38.8	19.0	16.4	1.1	36.5	19.1	16.5	0.2	35.8	8184.1
LS-2	16.3	16.7	262.7	295.7	20.6	20.9	3.6	45.1	20.8	21.0	1.1	42.9	20.9	21.1	0.2	42.2	6990.4
LS-4	17.2	20.1	265.3	302.7	21.4	25.0	3.6	50.0	21.6	25.2	1.1	47.9	21.7	25.3	0.2	47.2	6477.0
LS-6	17.5	21.9	263.0	302.4	21.6	27.1	3.6	52.3	21.8	27.3	1.1	50.2	21.9	27.4	0.2	49.5	6305.2
LS-8	17.5	22.9	267.2	307.6	21.4	28.4	3.7	53.5	21.6	28.6	1.0	51.3	21.7	28.8	0.2	50.6	6216.1
LS-10	17.4	23.6	264.9	305.9	21.2	29.3	3.6	54.1	21.3	29.5	1.1	51.9	21.4	29.6	0.2	51.2	6160.9

Table 12: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 3$.

As we have seen, LD does not behave well when it has to process a large amount of information in the second level. This is natural and occurs because we migrate (in part) the processing from a TRIE structure to sequential search. The TRIE structure group several words that have the common prefix, in this way, the processing is performed in one node, but this single node can cover several words. On the other hand, the processing in the second level is performed in a sequential list, which does not allow grouping of words, so it is necessary to process several words, even if they have the common prefix. To try avoid slow processing in the second level, we proposed the LS method that controls the amount of information that is added to the second level.

The LS-1 especially allows a TRIE with few incomplete branches, so we have a method that has the same complexity of the BEVA - which runs at the first level. Thus, we expect that this method will have the query processing times similar to BEVA, our premise to this is that we have an index closes to BEVA index. In addition, we also expect to have less memory consumption than the BEVA because the LS method does not have a complete TRIE.

LS better controls the amount of information that is added to the second level. Thus, we does not have a large disparity of time in the second level processing as we increase the values of max_{slot}, as it happens for the LD method in short prefixes indexed in the TRIE. In Tables 11, 12 and 13 it is possible to observe that the processing times taken in the second level grows slightly as we increase the values of max_{slot}. In contrast, there are always processing in the second level, even for small prefix queries, such as the prefix query length 5. This is because there are several words in the TRIE that may be indexed with a prefix smaller than the prefix query length. For example, the word "schwarzenegger" may not have a common prefix with any word in a given dataset, so this word may have only one character indexed in the TRIE and a reference to a list with a single word in the second level.

	$\tau = 3$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1º	2º	fetch	total	1º	2º	fetch	total	1º	2º	fetch	total	1º	2º	fetch	total	(MB)
LD4	13.9	76.7	92.4	183.0	15.8	102.2	1.0	119.0	15.9	103.1	0.2	119.2	15.9	103.4	-	119.3	1774.8
LD5	8.6	12.7	91.8	113.2	11.8	28.7	1.0	41.6	11.9	29.7	0.2	41.7	11.9	29.9	-	41.8	1777.0
LD6	6.7	1.5	91.8	100.0	9.5	7.8	1.0	18.3	9.5	8.7	0.2	18.5	9.5	9.0	-	18.6	1786.0
LD7	6.4	0.2	91.4	98.0	9.0	2.5	1.0	12.6	9.1	3.4	0.2	12.7	9.1	3.7	-	12.8	1801.0
LD8	6.4	-	90.7	97.0	8.9	0.9	1.0	10.8	9.0	1.6	0.2	10.8	9.0	1.8	-	10.9	1822.0
LD9	6.4	-	92.7	99.1	8.9	0.2	1.0	10.1	9.0	0.8	0.2	10.0	9.0	1.0	-	10.1	1849.2
LD10	6.4	-	90.8	97.2	8.8	0.1	1.0	9.8	8.9	0.4	0.2	9.5	8.9	0.7	-	9.6	1883.6
LD11	6.4	-	91.9	98.2	8.8	-	1.0	9.8	8.9	0.2	0.2	9.3	8.9	0.4	-	9.4	1926.5
LD12	6.4	-	92.5	98.9	8.8	-	1.0	9.8	9.0	0.1	0.2	9.3	9.0	0.3	-	9.3	1979.1
LD13	6.3	-	91.2	97.5	8.8	-	1.0	9.8	8.9	-	0.2	9.1	8.9	0.2	-	9.2	2042.7
LD14	6.5	-	91.2	97.7	9.0	-	1.0	10.0	9.1	-	0.2	9.3	9.2	0.1	-	9.4	2117.8
LD15	6.5	-	91.8	98.2	9.0	-	1.0	10.0	9.1	-	0.2	9.3	9.2	0.1	-	9.3	2205.0
LS-1	13.4	12.5	90.1	115.9	16.9	15.6	1.0	33.5	17.0	15.7	0.2	32.9	17.1	15.8	-	32.9	2756.2
LS-2	14.9	15.9	90.4	121.2	18.4	19.8	1.0	39.2	18.6	19.9	0.2	38.7	18.6	19.9	-	38.6	2105.2
LS-4	15.5	18.4	91.6	125.6	18.9	22.7	1.0	42.7	19.1	22.9	0.2	42.1	19.1	22.9	-	42.1	1936.7
LS-6	15.4	19.2	90.9	125.5	18.7	23.7	1.0	43.3	18.8	23.8	0.2	42.8	18.8	23.9	-	42.7	1886.8
LS-8	15.2	19.8	91.2	126.2	18.2	24.3	1.0	43.5	18.4	24.4	0.2	43.0	18.4	24.5	-	42.9	1860.8
LS-10	14.9	19.9	91.4	126.2	17.8	24.4	1.0	43.2	17.9	24.6	0.2	42.7	17.9	24.7	-	42.6	1845.0

Table 13: Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 3$.

For $|\mathcal{P}| = 5$ and $\tau = 1$ the LS and LD methods presented very close time in the USADDR and DBLP datasets. When we compare LD4 and LD5 with the LS method (for any value of max_{slot}) the LS method was faster (Tables 14 and 16). In this scenario, the LS method is faster due to better control of the amount of information added to the second level, ie, there are always little information in the second level, unlike the LD4 and LD5 methods, which processes large lists of words in the second level. When we increase the prefix query length to 9, 13, and 17 the LD method becomes quite competitive and, for large prefixes, it is slightly better than LS (for any value of max_{slot}). This is because the second level is triggered for smaller lists, until processing does not exist in this level, as we can see in Tables 14 and 16, $|\mathcal{P}| = 9$ and max_{depth} varying from 10 to 15.

In the MEDLINE dataset and $\tau = 1$ (Table 15) we see an advantage for the LS method. For $|\mathcal{P}| = 5$, the searching time was better in the LD method, but the fetching time was better in the LS, and the fetching time dominated the response time for this prefix query. For $|\mathcal{P}| = 9$, the searching time was better for the LS method, but the fetching time was better in the LD method, as the fetching time dominated, the response time for this prefix query was better in the LD method. For $|\mathcal{P}| = 13$ and $|\mathcal{P}| = 17$ the fetching time is almost negligible and so the LS method is better than LD method, as we increase the values of max_{slot}, this is because the first level time decreased considerably when we increased the size of the lists in the second level and as we've already seen, the expected is that we don't have a long time in the second level for LS method, while in the LD method we had an increase in the first level time and the decrease in the second level time was not sufficient to compensate such increasing.

	$\tau = 1$																
method	$\|\mathcal{P}\| = 5$				$\|\mathcal{P}\| = 9$				$\|\mathcal{P}\| = 13$				$\|\mathcal{P}\| = 17$				Mem
	1°	2°	fetch	total	1°	2°	fetch	total	1°	2°	fetch	total	1°	2°	fetch	total	(MB)
LD4	0.2	1.2	1.0	2.4	0.2	2.2	0.1	2.5	0.2	2.3	-	2.5	0.2	2.3	-	2.5	2021.9
LD5	0.2	0.1	0.9	1.2	0.2	0.8	0.1	1.1	0.2	0.9	-	1.2	0.2	0.9	-	1.2	2035.6
LD6	0.2	-	1.0	1.1	0.2	0.4	0.1	0.7	0.2	0.5	-	0.7	0.2	0.5	-	0.8	2068.8
LD7	0.2	-	0.9	1.1	0.2	0.2	0.1	0.5	0.2	0.3	-	0.5	0.2	0.3	-	0.5	2131.2
LD8	0.2	-	1.0	1.1	0.2	0.1	0.1	0.4	0.2	0.2	-	0.4	0.2	0.2	-	0.4	2225.1
LD9	0.2	-	0.9	1.1	0.2	-	0.1	0.3	0.2	0.1	-	0.4	0.2	0.1	-	0.4	2355.0
LD10	0.2	-	0.9	1.2	0.2	-	0.1	0.3	0.3	0.1	-	0.3	0.3	0.1	-	0.3	2522.5
LD11	0.2	-	0.9	1.1	0.2	-	0.1	0.3	0.2	-	-	0.3	0.2	-	-	0.3	2724.7
LD12	0.2	-	1.0	1.1	0.2	-	0.1	0.3	0.3	-	-	0.3	0.3	-	-	0.3	2954.4
LD13	0.2	-	0.9	1.1	0.2	-	0.1	0.3	0.2	-	-	0.3	0.3	-	-	0.3	3202.4
LD14	0.2	-	0.9	1.1	0.2	-	0.1	0.3	0.2	-	-	0.3	0.2	-	-	0.3	3460.0
LD15	0.2	-	0.9	1.1	0.2	-	0.1	0.3	0.2	-	-	0.3	0.2	-	-	0.3	3720.0
LS-1	0.2	-	0.9	1.2	0.2	0.1	0.1	0.4	0.3	0.1	-	0.3	0.3	0.1	-	0.3	2862.5
LS-2	0.2	-	0.9	1.2	0.2	0.1	0.1	0.4	0.3	0.1	-	0.4	0.3	0.1	-	0.4	2481.5
LS-4	0.2	0.1	0.9	1.2	0.3	0.1	0.1	0.4	0.3	0.1	-	0.4	0.3	0.1	-	0.4	2284.7
LS-6	0.2	0.1	0.9	1.2	0.3	0.1	0.1	0.5	0.3	0.1	-	0.4	0.3	0.1	-	0.4	2213.3
LS-8	0.2	0.1	0.9	1.2	0.3	0.1	0.1	0.5	0.3	0.2	-	0.4	0.3	0.2	-	0.4	2174.7
LS-10	0.2	0.1	0.9	1.3	0.3	0.2	0.1	0.5	0.3	0.2	-	0.4	0.3	0.2	-	0.4	2150.2

Table 14: Query processing time (at 1° and 2° level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 1$.

When we have a higher volume of processing, as occurs when we increase the edit distance threshold to 2, 3 or 4 we see that LD is faster than LS as we increase the values of

	$\tau = 1$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	0.2	3.4	13.4	17.1	0.3	10.0	1.5	11.9	0.4	11.6	0.4	12.4	0.4	12.0	0.1	12.5	5900.4
LD5	0.2	0.5	13.4	14.1	0.3	5.2	1.5	7.0	0.4	6.8	0.4	7.6	0.4	7.2	0.1	7.7	5907.9
LD6	0.2	-	13.5	13.7	0.3	2.4	1.5	4.2	0.3	4.0	0.4	4.8	0.3	4.4	0.1	4.8	5922.7
LD7	0.2	-	13.4	13.6	0.3	1.2	1.5	3.0	0.4	2.8	0.4	3.6	0.4	3.2	0.1	3.6	5946.9
LD8	0.2	-	13.7	13.9	0.3	0.4	1.5	2.2	0.4	2.1	0.4	2.9	0.4	2.5	0.1	3.0	5982.5
LD9	0.2	-	13.3	13.5	0.3	0.1	1.5	1.9	0.3	1.4	0.4	2.2	0.4	1.8	0.1	2.2	6029.1
LD10	0.2	-	13.4	13.6	0.3	-	1.5	1.8	0.3	0.9	0.4	1.6	0.3	1.3	0.1	1.7	6089.4
LD11	0.2	-	13.7	13.9	0.3	-	1.5	1.8	0.3	0.5	0.4	1.3	0.3	0.9	0.1	1.4	6172.8
LD12	0.2	-	13.6	13.8	0.3	-	1.5	1.8	0.3	0.2	0.4	0.9	0.3	0.6	0.1	1.0	6283.5
LD13	0.2	-	13.4	13.6	0.3	-	1.5	1.8	0.3	-	0.4	0.7	0.3	0.4	0.1	0.8	6420.5
LD14	0.2	-	13.4	13.6	0.3	-	1.5	1.8	0.3	-	0.4	0.7	0.3	0.2	0.1	0.6	6587.5
LD15	0.2	-	13.4	13.6	0.3	-	1.5	1.8	0.3	-	0.4	0.7	0.3	0.1	0.1	0.5	6786.0
LS-1	0.2	0.1	13.3	13.6	0.3	0.1	1.5	1.9	0.4	0.1	0.4	0.8	0.4	0.1	0.1	0.5	8179.2
LS-2	0.2	0.1	13.4	13.7	0.3	0.1	1.5	1.9	0.3	0.1	0.4	0.8	0.4	0.1	0.1	0.5	6985.6
LS-4	0.2	0.1	13.5	13.8	0.3	0.1	1.6	2.0	0.3	0.1	0.4	0.8	0.3	0.1	0.1	0.5	6472.2
LS-6	0.2	0.1	13.3	13.6	0.3	0.2	1.6	2.0	0.3	0.2	0.4	0.8	0.3	0.2	0.1	0.5	6300.4
LS-8	0.2	0.1	13.5	13.9	0.3	0.2	1.6	2.0	0.3	0.2	0.4	0.9	0.3	0.2	0.1	0.6	6211.3
LS-10	0.2	0.1	13.4	13.8	0.3	0.2	1.6	2.0	0.3	0.2	0.4	0.9	0.3	0.2	0.1	0.6	6155.9

Table 15: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 1$.

	$\tau = 1$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	0.2	0.9	2.7	3.8	0.2	2.7	0.3	3.2	0.2	3.0	0.1	3.3	0.3	3.1	-	3.3	1764.0
LD5	0.2	0.1	2.7	3.0	0.2	1.4	0.3	1.9	0.2	1.7	0.1	2.0	0.2	1.8	-	2.0	1770.1
LD6	0.2	-	2.7	2.9	0.2	0.6	0.3	1.1	0.2	0.9	0.1	1.2	0.2	1.0	-	1.3	1780.8
LD7	0.2	-	2.7	2.9	0.2	0.3	0.3	0.8	0.2	0.6	0.1	0.9	0.2	0.7	-	0.9	1796.3
LD8	0.2	-	2.7	2.9	0.2	0.1	0.3	0.5	0.2	0.4	0.1	0.7	0.2	0.5	-	0.7	1817.5
LD9	0.2	-	2.7	2.9	0.2	-	0.3	0.5	0.2	0.3	0.1	0.6	0.2	0.4	-	0.6	1844.7
LD10	0.2	-	2.9	3.1	0.2	-	0.3	0.5	0.2	0.2	0.1	0.5	0.2	0.3	-	0.5	1879.2
LD11	0.2	-	3.1	3.3	0.2	-	0.3	0.5	0.2	0.1	0.1	0.4	0.2	0.2	-	0.4	1922.2
LD12	0.2	-	2.7	2.9	0.2	-	0.3	0.5	0.2	-	0.1	0.3	0.2	0.1	-	0.4	1974.8
LD13	0.2	-	2.8	3.0	0.2	-	0.3	0.5	0.2	-	0.1	0.3	0.2	0.1	-	0.3	2038.2
LD14	0.2	-	2.7	2.9	0.2	-	0.3	0.5	0.2	-	0.1	0.3	0.2	0.1	-	0.3	2113.4
LD15	0.2	-	2.8	3.0	0.2	-	0.3	0.5	0.2	-	0.1	0.3	0.2	-	-	0.3	2200.5
LS-1	0.2	0.1	2.9	3.1	0.2	0.1	0.3	0.6	0.3	0.1	0.1	0.4	0.3	0.1	-	0.4	2751.5
LS-2	0.2	0.1	2.8	3.1	0.2	0.1	0.3	0.6	0.3	0.1	0.1	0.4	0.3	0.1	-	0.4	2100.6
LS-4	0.2	0.1	2.9	3.2	0.2	0.1	0.3	0.6	0.3	0.2	0.1	0.5	0.3	0.2	-	0.4	1932.0
LS-6	0.2	0.1	3.0	3.3	0.2	0.2	0.3	0.7	0.3	0.2	0.1	0.5	0.3	0.2	-	0.5	1882.0
LS-8	0.2	0.1	2.7	3.1	0.2	0.2	0.3	0.7	0.3	0.2	0.1	0.5	0.3	0.2	-	0.5	1856.2
LS-10	0.2	0.2	2.8	3.2	0.2	0.2	0.3	0.7	0.2	0.2	0.1	0.5	0.3	0.2	-	0.5	1840.3

Table 16: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 1$.

max_{depth}. For example, in the USADDR dataset, $|\mathcal{P}| = 9$ and $\tau = 3$ (Table 11) the times varying from LD8 to LD15 was between 23.1 ms and 21.3 ms while the time in the LS-1 (best configuration for LS) was 53 ms, ie, LD was twice fast than the LS and the same is true for prefix queries sizes 13 and 17. In the MEDLINE and DBLP datasets (Tables 12 and 13), the same trend occurs and the LD advantage grows as we increase the value of τ (See Tables 18, 19 and 20). This great advantage for the LD surprised us and later we will try to understand the reason for this.

	$\tau = 3$															
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$			
	nodes	words	it 1º	it 2º	nodes	words	it 1º	it 2º	nodes	words	it 1º	it 2º	nodes	words	it 1º	it 2º
LD4	38.96	1412.35	105.07	1469.79	68.08	4835.46	150.51	5736.75	68.08	4848.83	150.51	5782.92	68.08	4850.18	150.51	5786.44
LD5	44.97	123.00	161.29	121.31	101.05	1165.10	317.25	1301.60	101.05	1183.38	317.25	1347.77	101.05	1184.72	317.25	1351.29
LD6	45.69	4.93	170.19	4.47	113.42	237.09	391.99	253.62	113.47	257.12	392.05	294.89	113.47	258.47	392.05	298.42
LD7	45.75	-	170.69	-	117.06	44.81	414.62	44.62	117.36	63.98	415.08	76.58	117.36	65.33	415.08	80.10
LD8	45.75	-	170.69	-	117.95	6.10	420.64	5.81	118.74	20.87	422.10	26.42	118.74	22.22	422.10	29.94
LD9	45.75	-	170.69	-	118.12	0.58	421.77	0.53	119.42	8.49	424.62	10.44	119.42	10.26	424.62	13.97
LD10	45.75	-	170.69	-	118.15	0.05	421.92	0.04	119.80	3.22	425.87	3.71	119.80	5.02	425.88	6.82
LD11	45.75	-	170.69	-	118.15	-	421.94	-	119.99	1.12	426.47	1.09	120.03	2.82	426.52	3.54
LD12	45.75	-	170.69	-	118.15	-	421.94	-	120.06	0.18	426.73	0.16	120.17	1.59	426.88	1.81
LD13	45.75	-	170.69	-	118.15	-	421.94	-	120.07	0.02	426.79	0.01	120.26	0.85	427.08	0.87
LD14	45.75	-	170.69	-	118.15	-	421.94	-	120.07	0.00	426.80	0.00	120.30	0.44	427.18	0.38
LD15	45.75	-	170.69	-	118.15	-	421.94	-	120.07	-	426.80	-	120.33	0.18	427.23	0.12
LS-1	44.33	9.77	160.44	10.25	110.97	35.15	382.72	39.22	112.44	36.09	386.25	40.54	112.62	36.21	386.56	40.71
LS-2	43.40	17.45	154.42	17.98	106.77	60.96	361.00	67.48	108.05	62.40	364.02	69.53	108.20	62.59	364.26	69.80
LS-4	42.05	30.21	146.23	31.10	101.27	100.61	333.55	111.65	102.35	102.68	336.02	114.66	102.47	102.95	336.21	115.05
LS-6	41.05	41.31	140.31	42.61	97.45	132.85	314.94	148.08	98.42	135.35	317.10	151.77	98.52	135.68	317.26	152.24
LS-8	40.14	52.12	135.21	54.01	94.31	161.85	300.16	181.31	95.20	164.68	302.10	185.53	95.29	165.06	302.24	186.07
LS-10	39.33	62.84	130.70	65.33	91.62	188.82	287.75	212.45	92.44	191.94	289.52	217.14	92.52	192.35	289.64	217.74

Table 17: USADDR - Average number of operations per query (on the scale of thousands) in query processing in the 1º and 2º level in the prefix query length 5, 9, 13 and 17 for $\tau = 3$ in the LD and LS methods.

In order to understand why the LD is much faster than the LS, we decided to compute the number of operations, such as: number of nodes created in the first level, number of word nodes created in the second level, number of iterations in the first level and number of iterations in the second level, with aim of verifying whether the difference in the number of operations might explain the differences in the time performance of the methods. For USADDR dataset, $\tau = 3$ and $|\mathcal{P}| = 17$, the query response time for LD7 was of 26.8 ms and the number of operations was 117.36, 65.33, 415.08 and 80.10 for the columns nodes, words, it 1º and it 2º, respectively, for LS-1 the query response time was of 53.0 ms and the number of operations was 112.62, 36.21, 386.56 and 40.71 for the columns nodes, words, it 1º and it 2º, respectively and this scenario is repeated for the other methods and prefix query lengths (see Tables 11 and 17). Surprisingly, we found a smaller number of operations for the LS-1 compared to the LD7, not justifying the advantage of the LD7 in this example. So this suggested another hypothesis for us to understand this wide advantage of LD in general, namely: the alternate processing between the levels affects the cache policies of the machine where we run the experiments.

The processing in the LD starts in the first level and after of a certain level in the TRIE the processing is entirely transferred to a new way of processing prefix queries, which are the lists in the second level. In the LS, the processing starts in the first level

	$\tau = 4$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	31.6	220.4	793.2	1045.2	60.9	570.0	3.3	634.2	60.9	572.8	0.1	633.9	61.0	573.0	-	634.0	2150.0
LD5	39.4	87.4	796.0	922.9	118.9	354.7	3.5	477.2	119.0	357.6	0.1	476.8	119.0	357.8	-	476.8	2169.1
LD6	25.6	17.6	794.0	837.3	109.4	146.7	3.4	259.5	109.6	149.6	0.1	259.3	109.6	149.7	-	259.4	2198.6
LD7	22.3	1.9	798.1	822.3	96.0	46.0	4.1	146.1	96.7	48.9	0.1	145.8	96.7	49.1	-	145.8	2253.4
LD8	21.8	0.2	798.2	820.1	90.4	12.3	4.2	106.9	91.8	15.2	0.1	107.1	91.8	15.4	-	107.2	2346.0
LD9	21.7	-	794.5	816.2	88.3	2.6	4.1	95.1	90.2	5.0	0.1	95.4	90.2	5.2	-	95.4	2476.1
LD10	21.6	-	784.7	806.3	87.3	0.5	3.5	91.3	89.4	1.9	0.1	91.4	89.4	2.1	-	91.5	2643.6
LD11	21.6	-	771.6	793.2	87.4	0.1	3.5	91.0	89.5	0.8	0.1	90.5	89.5	1.0	-	90.6	2845.9
LD12	21.6	-	792.4	814.0	87.4	-	4.2	91.6	89.5	0.3	0.1	90.0	89.6	0.5	-	90.1	3075.6
LD13	21.6	-	783.8	805.4	87.2	-	3.5	90.8	89.3	0.1	0.1	89.5	89.4	0.3	-	89.8	3323.6
LD14	21.9	-	767.7	789.6	87.9	-	3.5	91.4	89.9	-	0.1	90.1	90.1	0.2	-	90.3	3581.2
LD15	21.6	-	777.2	798.8	87.0	-	3.5	90.5	89.0	-	0.1	89.2	89.3	0.1	-	89.4	3841.2
LS-1	41.3	33.3	774.7	849.2	141.8	113.6	4.2	259.6	143.8	115.3	0.1	259.3	144.0	115.4	-	259.4	2983.1
LS-2	45.4	42.4	771.0	858.9	150.3	142.3	3.4	296.1	152.1	144.3	0.1	296.6	152.2	144.4	-	296.7	2601.9
LS-4	47.1	49.5	777.0	873.6	149.0	162.2	3.4	314.7	150.5	164.3	0.1	314.9	150.6	164.4	-	315.0	2405.9
LS-6	47.2	52.5	785.1	884.8	145.7	170.6	4.1	320.4	147.0	172.7	0.1	319.8	147.1	172.8	-	319.9	2334.5
LS-8	46.3	53.9	762.1	862.2	140.6	173.4	3.3	317.3	141.8	175.4	0.1	317.3	141.8	175.5	-	317.4	2296.3
LS-10	45.6	55.0	777.5	878.1	136.5	175.9	3.4	315.7	137.6	177.9	0.1	315.6	137.6	178.0	-	315.6	2272.2

Table 18: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 4$.

	$\tau = 4$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	26.4	345.2	1513.5	1885.1	49.0	880.2	13.9	943.1	49.3	890.3	1.9	941.6	49.4	892.3	0.3	942.0	6022.9
LD5	21.8	123.4	1462.0	1607.2	60.0	467.3	13.7	541.0	60.3	477.5	2.0	539.7	60.3	479.5	0.3	540.1	6030.0
LD6	14.1	19.3	1434.6	1468.1	49.0	152.6	13.7	215.3	49.2	162.2	1.9	213.4	49.3	164.2	0.3	213.8	6046.3
LD7	12.3	1.8	1436.0	1450.1	42.3	41.2	13.7	97.1	42.7	49.6	1.9	94.2	42.7	51.6	0.3	94.6	6067.1
LD8	11.9	0.1	1440.0	1452.0	38.9	11.0	13.6	63.4	39.6	17.9	1.9	59.4	39.6	19.9	0.3	59.9	6100.5
LD9	11.8	-	1449.9	1461.7	37.2	2.7	13.6	53.5	38.2	7.6	1.9	47.7	38.3	9.6	0.3	48.2	6147.8
LD10	11.9	-	1457.5	1469.3	37.1	0.6	13.7	51.5	38.1	3.3	1.9	43.3	38.1	5.1	0.3	43.6	6207.3
LD11	11.8	-	1449.6	1461.4	37.4	0.2	13.6	51.2	38.4	1.5	1.9	41.8	38.5	3.1	0.3	41.9	6291.9
LD12	11.8	-	1447.8	1459.6	36.7	0.1	13.6	50.4	37.7	0.7	1.9	40.2	37.8	2.0	0.3	40.1	6401.6
LD13	11.8	-	1454.1	1465.9	36.8	-	13.6	50.5	37.8	0.3	1.9	39.9	38.0	1.3	0.3	39.6	6539.2
LD14	11.8	-	1455.7	1467.5	36.8	-	13.6	50.4	37.8	0.1	1.9	39.8	38.0	0.8	0.3	39.2	6705.8
LD15	11.7	-	1451.7	1463.4	36.7	-	13.6	50.3	37.7	-	1.9	39.6	37.9	0.5	0.3	38.7	6904.7
LS-1	26.4	25.0	1329.7	1381.2	65.6	59.7	13.7	139.1	66.6	60.6	1.9	129.1	66.9	60.8	0.3	128.0	8298.2
LS-2	28.2	30.3	1352.7	1411.2	68.3	72.2	13.8	154.3	69.3	73.2	1.9	144.4	69.5	73.5	0.3	143.3	7104.4
LS-4	28.6	34.4	1356.6	1419.5	67.8	82.1	13.7	163.6	68.7	83.2	1.9	153.8	68.9	83.5	0.3	152.8	6591.1
LS-6	28.1	35.7	1285.0	1348.9	65.9	85.7	13.8	165.4	66.7	86.9	1.8	155.5	67.0	87.2	0.3	154.5	6419.4
LS-8	27.5	36.6	1287.4	1351.5	64.3	88.4	13.8	166.5	65.1	89.6	1.8	156.5	65.3	89.9	0.3	155.5	6330.4
LS-10	26.8	36.8	1288.7	1352.3	62.4	89.6	13.5	165.5	63.1	90.8	1.8	155.8	63.3	91.1	0.3	154.8	6275.1

Table 19: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 4$.

and at a certain level it can trigger processing in the second level and remain at both at the same time, which may negatively affect the cache policies of the machine. We thus decided to study the affects of the methods in the cache system. Preliminary data did not point a large difference in cache misses from one method to another as being the factor for the LD to perform better than the LS. However, we will try to better understand this cache factor in future works.

	$\tau = 4$																
method	$\|\mathcal{P}\| = 5$				$\|\mathcal{P}\| = 9$				$\|\mathcal{P}\| = 13$				$\|\mathcal{P}\| = 17$				Mem
	1º	2º	fetch	total	1º	2º	fetch	total	1º	2º	fetch	total	1º	2º	fetch	total	(MB)
LD4	21.2	111.6	384.4	517.2	39.6	283.3	3.5	326.3	39.6	285.9	0.3	325.9	39.7	286.3	0.1	326.1	1899.6
LD5	17.4	45.4	385.0	447.7	47.7	162.7	3.5	213.9	47.8	165.4	0.3	213.5	47.8	165.8	0.1	213.7	1895.9
LD6	10.9	7.5	387.0	405.4	36.9	54.1	3.5	94.5	37.0	56.7	0.3	94.0	37.0	57.1	0.1	94.2	1905.4
LD7	9.5	0.7	387.0	397.2	31.1	14.6	3.5	49.2	31.3	17.0	0.3	48.6	31.3	17.3	0.1	48.7	1917.4
LD8	9.1	0.1	378.0	387.3	28.7	4.0	3.5	36.2	29.1	6.1	0.3	35.5	29.1	6.4	0.1	35.6	1937.9
LD9	9.1	-	387.2	396.2	28.1	1.0	3.5	32.6	28.7	2.6	0.3	31.5	28.7	3.0	0.1	31.7	1963.2
LD10	9.0	-	384.6	393.6	27.9	0.2	3.5	31.7	28.5	1.1	0.3	30.0	28.5	1.5	0.1	30.1	1997.1
LD11	9.0	-	384.9	393.9	27.8	0.1	3.5	31.4	28.4	0.5	0.3	29.3	28.4	0.9	0.1	29.4	2040.2
LD12	9.1	-	384.3	393.4	28.0	-	3.5	31.5	28.6	0.2	0.3	29.2	28.7	0.6	0.1	29.3	2092.9
LD13	9.0	-	385.7	394.8	28.0	-	3.5	31.5	28.6	0.1	0.3	29.0	28.7	0.4	0.1	29.1	2156.5
LD14	9.1	-	387.7	396.8	28.0	-	3.5	31.5	28.5	-	0.3	28.9	28.7	0.2	0.1	29.0	2231.8
LD15	9.1	-	384.1	393.2	28.0	-	3.5	31.5	28.6	-	0.3	28.9	28.7	0.1	0.1	28.9	2319.0
LS-1	21.6	21.3	377.6	420.6	52.8	51.2	3.6	107.6	53.4	51.7	0.3	105.4	53.5	51.8	0.1	105.4	2870.2
LS-2	23.4	25.5	383.0	431.9	54.9	60.9	3.5	119.3	55.5	61.5	0.3	117.3	55.6	61.6	0.1	117.3	2219.2
LS-4	23.0	27.4	382.4	432.9	52.7	65.3	3.5	121.5	53.2	66.0	0.3	119.5	53.3	66.1	0.1	119.5	2050.8
LS-6	22.0	27.4	381.9	431.2	49.9	65.5	3.5	118.8	50.3	66.1	0.3	116.8	50.4	66.3	0.1	116.8	2000.9
LS-8	21.2	27.1	383.0	431.3	47.7	65.3	3.5	116.5	48.1	66.0	0.3	114.4	48.2	66.1	0.1	114.4	1975.1
LS-10	20.1	26.9	383.1	430.2	45.4	65.0	3.5	113.9	45.8	65.7	0.3	111.8	45.9	65.8	0.1	111.8	1959.3

Table 20: Query processing time (at 1º and 2º level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 4$.

As expected, the query processing times in second level for the LD becomes high when the processing in second level is triggered early for a very large list of words. But, as we increase the values of max_{depth} we notice a large difference in query processing time because the second level processing decreases considerably. The time difference is huge at $max_{depth} = 4$ and $max_{depth} = 15$. As an example, for $max_{depth} = 4$ the time spent is 119.2 ms and for $max_{depth} = 15$ the time spent is only 9.3 ms for the same query prefix length ($|\mathcal{P}| = 13$) in USADDR dataset and $\tau = 3$ (Table 11), a difference of more than 109 ms only by choosing the length of the prefixes to be indexed in the first level. While in LS, times do not vary so much as we increase the value of max_{slot}, in some scenarios even worse times. Thus, it is easy to see that it is necessary to study the ideal choice of max_{depth} to be indexed in the first level for the LD.

LD performs better when $|\mathcal{P}| + \tau \leq max_{depth}$, ie, the processing occurs only at the first level and the processing in second level is not required. Or when we have relatively large prefix sizes indexed in TRIE. At the end of this section we show such prefixes, which are not very large but which already provide a very competitive method. Thus, when occurs this scenarios we have a very competitive method in terms of processing queries. For example, in LD7, LD8, LD9, LD10 and LD11 methods for $\tau = 2$ and $|\mathcal{P}| = 13$ in Table

	$\tau = 2$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	3.1	17.9	8.4	29.4	3.3	22.8	0.2	26.3	3.3	23.0	-	26.4	3.3	23.1	-	26.4	2024.1
LD5	2.8	2.1	8.3	13.2	3.3	5.6	0.2	9.2	3.3	5.9	-	9.2	3.3	5.9	-	9.3	2036.8
LD6	2.6	0.1	8.2	11.0	3.2	1.7	0.2	5.1	3.2	1.9	-	5.2	3.2	2.0	-	5.2	2071.2
LD7	2.6	-	8.3	10.8	3.2	0.6	0.2	4.0	3.2	0.8	-	4.1	3.2	0.9	-	4.1	2133.8
LD8	2.6	-	8.3	10.9	3.2	0.2	0.2	3.6	3.2	0.4	-	3.7	3.2	0.4	-	3.7	2225.4
LD9	2.6	-	8.3	10.9	3.2	-	0.2	3.4	3.2	0.2	-	3.5	3.2	0.3	-	3.5	2355.2
LD10	2.6	-	8.3	10.9	3.2	-	0.2	3.4	3.3	0.1	-	3.4	3.3	0.2	-	3.4	2522.7
LD11	2.6	-	8.3	10.9	3.2	-	0.2	3.4	3.3	0.1	-	3.3	3.3	0.1	-	3.4	2725.0
LD12	2.6	-	8.3	10.9	3.2	-	0.2	3.4	3.3	-	-	3.3	3.3	0.1	-	3.4	2954.7
LD13	2.6	-	8.3	11.0	3.3	-	0.2	3.5	3.4	-	-	3.4	3.4	-	-	3.5	3202.7
LD14	2.6	-	8.4	10.9	3.2	-	0.2	3.4	3.2	-	-	3.3	3.3	-	-	3.3	3460.2
LD15	2.6	-	8.4	10.9	3.2	-	0.2	3.4	3.3	-	-	3.3	3.3	-	-	3.3	3720.2
LS-1	3.2	1.2	8.1	12.5	3.9	1.7	0.2	5.8	4.0	1.7	-	5.7	4.0	1.7	-	5.8	2862.9
LS-2	3.4	1.8	8.2	13.3	4.1	2.4	0.2	6.7	4.2	2.4	-	6.6	4.2	2.4	-	6.7	2481.8
LS-4	3.5	2.4	8.2	14.1	4.3	3.1	0.2	7.6	4.3	3.2	-	7.5	4.3	3.2	-	7.5	2285.0
LS-6	3.6	2.8	8.2	14.6	4.3	3.6	0.2	8.1	4.4	3.7	-	8.1	4.4	3.7	-	8.1	2213.5
LS-8	3.6	3.0	8.2	14.8	4.3	3.9	0.2	8.4	4.4	4.0	-	8.3	4.4	4.0	-	8.3	2175.1
LS-10	3.6	3.3	8.2	15.1	4.3	4.2	0.2	8.7	4.4	4.2	-	8.6	4.4	4.2	-	8.6	2150.5

Table 21: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in USADDR dataset and $\tau = 2$.

	$\tau = 2$																
method	$\lvert\mathcal{P}\rvert = 5$				$\lvert\mathcal{P}\rvert = 9$				$\lvert\mathcal{P}\rvert = 13$				$\lvert\mathcal{P}\rvert = 17$				Mem
	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	1^o	2^o	fetch	total	(MB)
LD4	2.6	29.9	34.6	67.0	2.9	44.4	2.0	49.3	3.0	46.6	0.6	50.2	3.0	47.2	0.1	50.3	5901.2
LD5	2.1	3.0	34.4	39.5	2.4	13.1	2.0	17.6	2.5	15.4	0.6	18.5	2.5	16.0	0.1	18.6	5904.9
LD6	1.9	0.2	33.3	35.4	2.3	5.2	2.0	9.4	2.3	7.4	0.6	10.3	2.4	8.0	0.1	10.5	5912.4
LD7	1.9	-	33.5	35.4	2.2	1.8	2.0	6.0	2.3	4.0	0.6	6.9	2.3	4.6	0.1	7.1	5936.5
LD8	1.9	-	33.3	35.2	2.2	0.6	1.9	4.8	2.3	2.6	0.6	5.5	2.3	3.2	0.1	5.6	5971.8
LD9	1.9	-	33.2	35.1	2.2	0.1	1.9	4.3	2.3	1.7	0.6	4.5	2.3	2.3	0.1	4.7	6020.8
LD10	1.9	-	32.9	34.8	2.2	-	1.9	4.2	2.3	1.0	0.6	3.9	2.3	1.6	0.1	4.1	6086.7
LD11	2.0	-	33.5	35.5	2.3	-	1.9	4.2	2.3	0.5	0.6	3.4	2.3	1.1	0.1	3.6	6172.9
LD12	1.9	-	32.8	34.7	2.2	-	1.9	4.1	2.3	0.2	0.6	3.0	2.3	0.8	0.1	3.2	6283.6
LD13	1.9	-	33.3	35.2	2.3	-	1.9	4.2	2.3	-	0.6	2.9	2.3	0.5	0.1	2.9	6420.6
LD14	1.9	-	34.4	36.3	2.2	-	2.0	4.3	2.3	-	0.6	2.9	2.3	0.3	0.1	2.7	6587.7
LD15	1.9	-	32.7	34.6	2.2	-	1.9	4.2	2.3	-	0.6	2.8	2.3	0.1	0.1	2.5	6786.2
LS-1	2.7	1.5	33.5	37.8	3.2	1.8	2.0	7.0	3.2	1.8	0.6	5.6	3.3	1.8	0.1	5.2	8179.5
LS-2	2.9	2.1	33.5	38.5	3.3	2.4	1.9	7.6	3.3	2.5	0.6	6.3	3.3	2.5	0.1	5.9	6985.8
LS-4	3.1	2.8	34.0	39.9	3.4	3.2	1.9	8.5	3.5	3.2	0.6	7.2	3.5	3.3	0.1	6.9	6472.4
LS-6	3.1	3.2	33.8	40.1	3.5	3.6	2.0	9.0	3.5	3.6	0.6	7.7	3.5	3.7	0.1	7.3	6300.6
LS-8	3.2	3.5	34.0	40.7	3.5	3.9	2.0	9.4	3.6	4.0	0.6	8.1	3.6	4.0	0.1	7.7	6211.5
LS-10	3.1	3.7	34.0	40.8	3.5	4.1	2.0	9.6	3.5	4.2	0.6	8.3	3.5	4.2	0.1	7.9	6156.1

Table 22: Query processing time (at 1^o and 2^o level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in MEDLINE dataset and $\tau = 2$.

21. There are also scenarios where processing in the second level is triggered frequently for LD, but does not negatively affect the searching time. This happens when processing queries of long query prefix length and larger τ. Such queries trigger the second level more often because they are often greater than the values of max_{depth}. However the time does not grow much because at deeper levels of TRIE, the set of results is small and, consequently, the list size to be processed in the second level are too small. For example, in DBLP dataset, $|\mathcal{P}| = 17$ and $\tau = 3$ the LD10, LD11, LD12, LD13, LD14 and LD15 (Table 13) methods trigger the second level, but the queries processing is fast.

	$\tau = 2$																
method	$\|\mathcal{P}\| = 5$				$\|\mathcal{P}\| = 9$				$\|\mathcal{P}\| = 13$				$\|\mathcal{P}\| = 17$				Mem
	1°	2°	fetch	total	1°	2°	fetch	total	1°	2°	fetch	total	1°	2°	fetch	total	(MB)
LD4	2.3	9.6	8.3	20.2	2.4	13.9	0.4	16.7	2.4	14.4	0.1	16.9	2.4	14.5	-	16.9	1764.2
LD5	1.8	1.1	8.3	11.1	2.0	4.1	0.5	6.5	2.0	4.6	0.1	6.7	2.0	4.7	-	6.8	1770.4
LD6	1.7	0.1	8.2	10.0	2.0	1.6	0.4	4.0	2.0	2.1	0.1	4.2	2.0	2.2	-	4.3	1781.3
LD7	1.7	-	8.3	10.0	2.0	0.6	0.4	3.0	2.0	1.1	0.1	3.2	2.0	1.2	-	3.3	1796.8
LD8	1.7	-	8.1	9.8	1.9	0.2	0.4	2.5	1.9	0.6	0.1	2.7	2.0	0.8	-	2.7	1817.7
LD9	1.7	-	8.3	10.0	1.9	-	0.4	2.4	2.0	0.4	0.1	2.5	2.0	0.5	-	2.5	1844.9
LD10	1.7	-	8.2	9.9	2.0	-	0.4	2.4	2.0	0.2	0.1	2.3	2.0	0.4	-	2.4	1879.4
LD11	1.7	-	8.1	9.8	1.9	-	0.4	2.4	2.0	0.1	0.1	2.2	2.0	0.3	-	2.3	1922.2
LD12	1.7	-	8.1	9.8	1.9	-	0.4	2.4	2.0	0.1	0.1	2.1	2.0	0.2	-	2.2	1974.8
LD13	1.7	-	8.3	10.0	2.0	-	0.4	2.4	2.0	-	0.1	2.1	2.0	0.1	-	2.2	2038.3
LD14	1.7	-	8.1	9.9	1.9	-	0.4	2.4	2.0	-	0.1	2.1	2.0	0.1	-	2.1	2113.5
LD15	1.7	-	8.3	10.0	1.9	-	0.4	2.4	2.0	-	0.1	2.1	2.0	-	-	2.0	2200.6
LS-1	2.4	1.6	8.8	12.8	2.7	1.8	0.5	5.0	2.8	1.8	0.1	4.8	2.8	1.8	-	4.7	2751.8
LS-2	2.7	2.2	9.6	14.4	3.0	2.5	0.4	5.9	3.0	2.5	0.1	5.6	3.0	2.5	-	5.6	2100.8
LS-4	2.8	2.8	8.8	14.4	3.1	3.1	0.4	6.7	3.1	3.2	0.1	6.4	3.2	3.2	-	6.4	1932.2
LS-6	2.9	3.2	8.4	14.4	3.2	3.5	0.4	7.1	3.2	3.6	0.1	6.9	3.2	3.6	-	6.8	1882.2
LS-8	2.9	3.4	9.4	15.6	3.1	3.8	0.4	7.4	3.2	3.8	0.1	7.1	3.2	3.9	-	7.1	1856.4
LS-10	2.9	3.6	8.4	14.8	3.2	4.0	0.4	7.5	3.2	4.0	0.1	7.3	3.2	4.0	-	7.3	1840.5

Table 23: Query processing time (at 1° and 2° level) and fetch time (ms) with prefix queries of sizes 5, 9, 13 and 17 and max_{depth} varying from 4 to 15 in LD, and max_{slot} with constants 1, 2, 4, 6, 8 and 10 in LS with memory usage value (MB) for each method in DBLP dataset and $\tau = 2$.

Therefore, for larger values of max_{depth}, usually starting at the prefix query length 7, the time spent in the second level decreases considerably, as can be seen in Tables 21 and 23 for $|\mathcal{P}| = 9$. In this way, as the value of max_{depth} grows the method will be faster. But the time difference between large values of max_{depth} is very small, for example $max_{depth} = 11$ and $max_{depth} = 15$ have an almost imperceptible difference, as can be seen in Table 21 for $|\mathcal{P}| = 13$ and $\tau = 2$, on the other hand, the same cannot be said for memory consumption. As we increase the values of max_{depth} there is considerable increase in memory consumption. Thus, it is necessary to choose the highest possible value for max_{depth} and that has a reasonable consumption of memory. We will analyze the memory consumption in more detail in the next section and try to suggest the best values of max_{depth} and max_{slot} for the LD and LS methods, respectively, in order to find a good balance between query processing time and memory consumption.

5.3.2 Memory Consumption

Query autocompletion systems keep the index and suggestions in memory to allow fast construction and access to the indexes, because it allows to show results to each user keystroke without a significant delay. However, it is necessary to balance the speed of query processing with the memory consumption because memory is a limited and costly resource. The amount of memory used is directly influenced by the amount of information that is added to the index. A practical example for this is the IncNGTRIE method proposed by Xiao et al. (2013), which is faster method in literature, however it consumes absurdly more memory than any algorithm in the literature, making its use impractical in several practical applications.

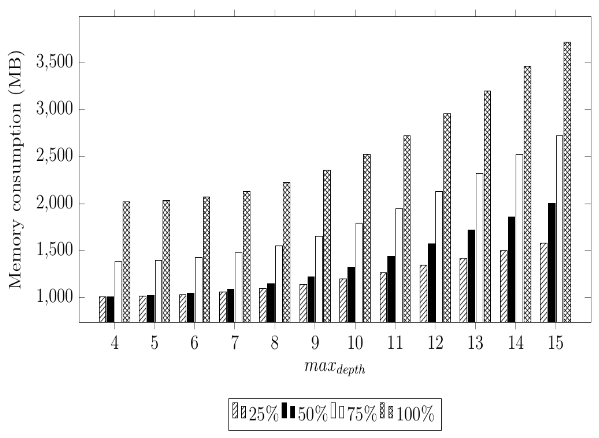
In contrast to query response time, the memory consumption in the LD is smaller when the value of max_{depth} is smaller because, in this scenario, there are few nodes indexed in the TRIE when compared to the number of nodes indexed in the TRIE when we use higher values for max_{depth}. For instance, the number of nodes is 150,513 and 34,466,454 and the memory consumption is 2017.2 MB and 3719.8 MB in USADDR dataset (Table 24), for LD4 and LD15 methods, respectively. For large values of max_{depth}, the memory consumption increases as the amount of information in the index increases, as we can see in Table 24 (column MEM. (MB) for all datasets). In LS, the memory consumption is small when the value of max_{slot} is high. For small values of max_{slot}, the memory consumption increases. There is a clear trade off between memory consumption in both methods, and we need to study the behaviour of the two methods as we vary theirs parameters.

From results presented in Tables 14, 15 and 16, we observe that when the query prefix lengths is small and $\tau = 1$ we see that the LD5 was quite efficient in both query processing time and memory consumption. We can notice that, although the LD method is fast for large values of max_{depth}, the memory consumption significantly grows as we increase the values of max_{depth}, as can be seen the curve growth in Figure 15a to 25%, 50%, 75% and 100% of the USADDR dataset portion. However, it is worth mentioning that the memory consumption even for large values of max_{depth} is still little when compared to the memory consumption of any method that does the complete indexing of strings in the TRIE.

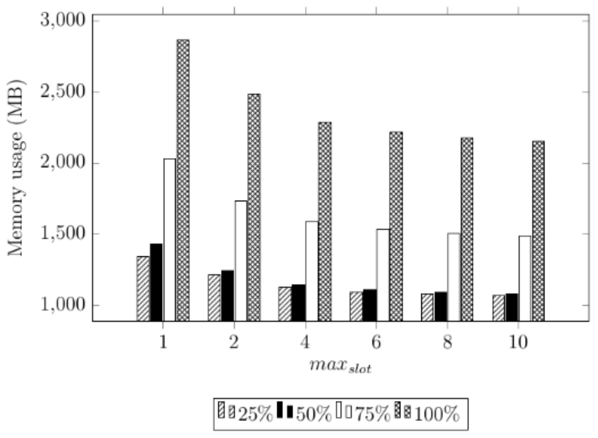
The context and scenario of a system can also influence in the choice of the ideal value for max_{depth} or max_{slot}. For example, a smaller value of max_{depth} can be chosen for less memory consumption, but the processing time can increase or a larger value of max_{depth} can be chosen to faster query response, but the memory consumption increases. Similarly, a larger value of max_{slot} can be chosen for less memory consumption, but the processing time can increase or a smaller value of max_{slot} can be chosen to faster query response, but the memory consumption increases. In addition, the ideal choice of max_{depth} or max_{slot} also depends on the average query prefix length that a given system receives,

	USADDR		MEDLINE		DBLP	
method	Number of nodes	Mem. (MB)	Number of nodes	Mem. (MB)	Number of nodes	Mem. (MB)
LD4	150,513	2017.2	121,865	5882.5	104,087	1761.6
LD5	589,965	2032.1	330,084	5890.6	266,186	1768.1
LD6	1,485,677	2067.6	685,473	5905.7	511,954	1779.1
LD7	2,938,372	2130.3	1,231,710	5930.0	852,792	1794.9
LD8	5,014,427	2224.6	2,013,567	5965.8	1,300,640	1816.3
LD9	7,795,064	2354.7	3,079,701	6015.7	1,870,284	1843.7
LD10	11,273,374	2522.3	4,488,767	6082.5	2,582,008	1878.4
LD11	15,352,963	2724.6	6,307,010	6169.7	3,460,691	1921.5
LD12	19,868,649	2954.2	8,596,566	6280.8	4,528,988	1974.3
LD13	24,655,213	3202.2	11,409,217	6418.9	5,806,163	2037.9
LD14	29,559,749	3459.9	14,785,962	6586.7	7,302,808	2113.1
LD15	34,466,454	3719.8	18,752,025	6785.7	9,019,275	2200.3
LS-1	19,360,374	2862.4	49,787,393	8179.2	20,550,602	2751.5
LS-2	11,767,608	2481.5	26,270,104	6985.4	8,147,970	2100.6
LS-4	7,432,404	2284.7	15,262,983	6472.2	4,560,809	1932.0
LS-6	5,713,691	2213.3	11,295,344	6300.4	3,390,379	1882.0
LS-8	4,740,798	2174.7	9,147,067	6211.2	2,755,905	1856.1
LS-10	4,097,443	2150.2	7,768,093	6155.8	2,352,604	1840.3
BEVA	77,542,385	5441.7	979,452,352	54802.2	250,356,405	14773.1

Table 24: Number of nodes and memory usage in TRIE index build in the USADDR, MEDLINE and DBLP datasets



(a) Amount of memory used (MB) for LD method



(b) Amount of memory used (MB) for LS method

Figure 15: Amount of memory used (MB) for LD and LS methods in the USADDR dataset with 25%, 50%, 75% and 100% of the dataset portion and $\tau = 1$

the amount of memory resource available and the dataset size.

Finally, in this dissertation we selected the LD8, LD10, LD12, LS-1 and LS-4 to be compared with baseline methods, based on the aspects that we exposed previously and also in the queries prefix lengths 5, 9, 13 and 17. Such methods showed a better balance in the average between query processing time and memory consumption among all datasets and edit distance thresholds experienced.

5.4 Baselines Comparison

We compared the LD and LS methods with META (code provided by the authors) and BEVA (Binary provided by the authors, however it contained a reading problem and so we used our BEVA implementation), currently the state-of-the-art algorithm in the literature. The works in the literature also compare with the ICAN, IPCAN and IncNGTRIE methods. However, for such algorithms we found problems of correctness in implementations available for them, so the comparison was not possible and we discarded them. For query response time, we experienced varying the τ from 1 to 4 and prefix queries of sizes 5, 9, 13 and 17. For memory consumption, we selected only the results of $\tau = 3$ (because memory consumption is close to all edit distance threshold) and portions of 25%, 50%, 75% and 100% of the sizes of the dataset to check the memory consumption between the baselines when we scale the size of the dataset.

5.4.1 Varying Edit Distance Threshold

In the Tables 25 and 26 we show the query response times for all algorithms with edit distance threshold τ between 1 and 4, at fixed prefix query lengths of 5 and 13, respectively. We show results only for prefix query lengths 5 and 13 because we want to analyze results with small and larger prefix query sizes. The prefix query size 13 is sufficient to make our proposed methods perform processing also in the second level, allowing us to analyze our methods with processing in both levels. From Table 25, $\tau = 1$ and USADDR dataset we observed that all compared methods are very close in time, with a slight advantage for the META and BEVA methods. When we increased the τ to 2, we noticed that the META method time become much worse, while the BEVA time remains smaller than the others, but with a very small difference. This worsening of the META method and the increase in difference between the BEVA and our proposed methods is greater when we increase the τ to 3 and 4. However, we observed that our methods maintain a smaller confidence interval when compared to the META and BEVA methods. This shows us that our methods is more stable than the BEVA despite having a slightly longer time. This trend is repeated also in the MEDLINE and DBLP datasets.

For $|\mathcal{P}| = 13$, USADDR dataset, varying τ from 1 to 4 (Table 26) we realized that the time difference between the BEVA and LD decrease even more, this is because the fetching time does not have much influence in the final results and the LD method triggers the processing in the second level for small lists, this becomes more frequently when we increase the prefix query size. As we can see we have 0.23ms and 0.27ms, 3.08ms and 3.33ms, 20.76ms and 21.43ms, 85.37ms and 89.95ms, for τ with sizes 1, 2, 3 and 4 in the BEVA and LD12 methods, respectively.

	USADDR				MEDLINE				DBLP			
	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$
BEVA	1.00 ± 0.096	9.51 ± 0.372	122.07 ± 3.257	706.66 ± 8.028	9.04 ± 1.417	30.91 ± 2.090	225.93 ± 6.939	1297.80 ± 29.329	2.69 ± 0.300	9.34 ± 0.516	70.64 ± 1.627	340.66 ± 4.290
META	0.79 ± 0.011	59.45 ± 0.985	4654.42 ± 136.661	78063.53 ± 1272.354	- -	- -	- -	- -	0.70 ± 0.010	50.60 ± 0.878	1918.91 ± 36.257	14145.36 ± 266.927
LD8	1.14 ± 0.003	10.89 ± 0.037	131.90 ± 0.192	820.13 ± 0.237	13.87 ± 0.002	35.22 ± 0.021	276.01 ± 0.072	1451.99 ± 0.120	2.87 ± 0.002	9.80 ± 0.020	97.04 ± 0.052	387.26 ± 0.078
LD10	1.15 ± 0.003	10.88 ± 0.038	131.83 ± 0.191	806.31 ± 0.224	13.63 ± 0.002	34.79 ± 0.021	275.04 ± 0.072	1469.33 ± 0.112	3.09 ± 0.002	9.94 ± 0.020	97.21 ± 0.052	393.60 ± 0.073
LD12	1.15 ± 0.003	10.88 ± 0.038	131.09 ± 0.193	813.99 ± 0.222	13.77 ± 0.002	34.72 ± 0.020	271.57 ± 0.072	1459.60 ± 0.110	2.88 ± 0.002	9.76 ± 0.020	98.85 ± 0.052	393.38 ± 0.074
LS-1	1.17 ± 0.002	12.54 ± 0.059	149.87 ± 0.493	849.22 ± 0.927	13.59 ± 0.003	37.79 ± 0.044	287.97 ± 0.255	1381.21 ± 0.416	3.12 ± 0.003	12.82 ± 0.045	115.87 ± 0.229	420.62 ± 0.337
LS-4	1.21 ± 0.003	14.11 ± 0.077	163.84 ± 0.661	873.64 ± 1.167	13.82 ± 0.003	39.87 ± 0.061	302.69 ± 0.320	1419.55 ± 0.482	3.22 ± 0.003	14.40 ± 0.062	125.61 ± 0.282	432.85 ± 0.389

Table 25: Query response time (ms) comparison with confidence interval between baselines methods varying τ from 1 to 4, $|\mathcal{P}| = 5$ in the USADDR, MEDLINE and DBLP datasets.

	USADDR				MEDLINE				DBLP			
	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$
BEVA	0.23 ± 0.005	3.08 ± 0.056	20.76 ± 0.388	85.37 ± 1.524	1.85 ± 0.621	2.60 ± 0.108	10.49 ± 0.267	36.66 ± 0.636	0.27 ± 0.013	2.00 ± 0.031	8.47 ± 0.103	26.53 ± 0.352
META	1.05 ± 0.019	63.01 ± 1.097	4712.01 ± 138.001	78878.52 ± 1285.878	- -	- -	- -	- -	0.84 ± 0.012	51.96 ± 0.899	1935.15 ± 36.679	14293.40 ± 270.603
LD8	0.42 ± 0.022	3.66 ± 0.090	23.11 ± 0.486	107.10 ± 2.371	2.86 ± 0.485	5.46 ± 0.475	16.85 ± 0.619	59.42 ± 1.764	0.71 ± 0.040	2.70 ± 0.059	10.78 ± 0.194	35.50 ± 0.687
LD10	0.33 ± 0.009	3.42 ± 0.065	21.67 ± 0.420	91.44 ± 1.704	1.64 ± 0.235	3.89 ± 0.202	13.16 ± 0.228	43.32 ± 0.771	0.48 ± 0.020	2.34 ± 0.035	9.53 ± 0.116	29.97 ± 0.426
LD12	0.27 ± 0.004	3.33 ± 0.060	21.43 ± 0.400	89.95 ± 1.606	0.89 ± 0.154	3.05 ± 0.064	12.06 ± 0.155	40.22 ± 0.574	0.34 ± 0.005	2.14 ± 0.027	9.26 ± 0.107	29.17 ± 0.381
LS-1	0.33 ± 0.005	5.75 ± 0.098	52.96 ± 0.944	259.27 ± 4.029	0.82 ± 0.146	5.61 ± 0.198	36.53 ± 0.433	129.06 ± 1.384	0.41 ± 0.004	4.75 ± 0.060	32.95 ± 0.362	105.43 ± 0.900
LS-4	0.39 ± 0.005	7.52 ± 0.121	69.95 ± 1.185	314.92 ± 4.385	0.80 ± 0.006	7.25 ± 0.089	47.87 ± 0.530	153.79 ± 1.510	0.48 ± 0.005	6.44 ± 0.079	42.13 ± 0.432	119.47 ± 0.904

Table 26: Query response time (ms) comparison with confidence interval between baselines methods varying τ from 1 to 4, $|\mathcal{P}| = 13$ in the USADDR, MEDLINE and DBLP datasets.

5.4.2 Varying Query Length

In Table 27, we experiment with our methods and baselines with fixed τ 3 and varying the prefix query with sizes 5, 9, 13 and 17. We use only $\tau = 3$ following the literature, it also allows us to analyze the query processing of the methods with a relevant amount of errors. Values greater than 3 are not common in practical scenarios. From Table 27 we can observe two things: (1) The LD method is the second fastest method, but with a lower confidence interval, in which it represents that this method is more stable. (2) When we increase the prefix query sizes to 9, 13 and 17, the LD method becomes very competitive when compared to the BEVA - which has a better time. For example, for $|\mathcal{P}| = 17$, the query response time is 20.79ms and 21.50ms for the BEVA and LD12 methods, respectively, a practically derisory difference, however the BEVA was slightly more stable with a confidence interval of 0.392 against 0.404 of the LD12 method.

For the DBLP and MEDLINE dataset, this trend continues, but the difference in query response times between the LD method (our fastest method) and the BEVA increases. This is because in the DBLP, despite being a smaller dataset than the USADDR dataset, with 4.5M words against 10.2M of words of the USADDR dataset, has an average

	USADDR				MEDLINE				DBLP			
	$\|\mathcal{P}\| = 5$	$\|\mathcal{P}\| = 9$	$\|\mathcal{P}\| = 13$	$\|\mathcal{P}\| = 17$	$\|\mathcal{P}\| = 5$	$\|\mathcal{P}\| = 9$	$\|\mathcal{P}\| = 13$	$\|\mathcal{P}\| = 17$	$\|\mathcal{P}\| = 5$	$\|\mathcal{P}\| = 9$	$\|\mathcal{P}\| = 13$	$\|\mathcal{P}\| = 17$
BEVA	122.07 ± 3.257	21.19 ± 0.420	20.76 ± 0.388	20.79 ± 0.392	225.93 ± 6.939	12.77 ± 0.481	10.49 ± 0.267	9.77 ± 0.139	70.64 ± 1.627	9.13 ± 0.134	8.47 ± 0.103	8.38 ± 0.100
META	4654.42 ± 136.661	4709.06 ± 137.903	4712.01 ± 138.001	4713.02 ± 138.020	- -	- -	- -	- -	1918.91 ± 36.257	1934.24 ± 36.660	1935.15 ± 36.679	1935.53 ± 36.685
LD8	131.90 ± 0.192	23.00 ± 0.447	23.11 ± 0.486	23.15 ± 0.489	276.01 ± 0.072	16.51 ± 0.373	16.85 ± 0.619	17.04 ± 0.771	97.04 ± 0.052	10.75 ± 0.165	10.78 ± 0.194	10.86 ± 0.204
LD10	131.83 ± 0.191	21.64 ± 0.378	21.67 ± 0.420	21.71 ± 0.423	275.04 ± 0.072	14.39 ± 0.141	13.16 ± 0.228	13.41 ± 0.382	97.21 ± 0.052	9.85 ± 0.103	9.53 ± 0.116	9.62 ± 0.127
LD12	131.09 ± 0.193	21.66 ± 0.377	21.43 ± 0.400	21.50 ± 0.404	271.57 ± 0.072	14.14 ± 0.136	12.06 ± 0.155	12.05 ± 0.233	98.85 ± 0.052	9.84 ± 0.101	9.26 ± 0.107	9.32 ± 0.113
LS-1	149.87 ± 0.493	53.02 ± 0.916	52.96 ± 0.944	53.00 ± 0.946	287.97 ± 0.255	38.77 ± 0.419	36.53 ± 0.433	35.81 ± 0.433	115.87 ± 0.229	33.54 ± 0.356	32.95 ± 0.362	32.89 ± 0.364
LS-4	163.84 ± 0.661	69.97 ± 1.155	69.95 ± 1.185	69.99 ± 1.186	302.69 ± 0.320	50.02 ± 0.515	47.87 ± 0.530	47.19 ± 0.530	125.61 ± 0.282	42.67 ± 0.425	42.13 ± 0.432	42.09 ± 0.435

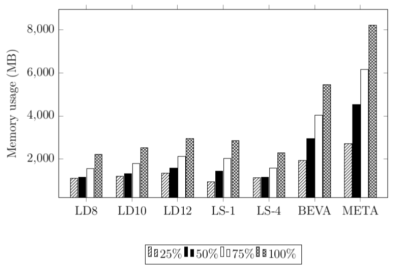
Table 27: Query response time (ms) comparison with confidence interval between baselines methods, varying the prefix queries sizes 5, 9, 13 and 17 for $\tau = 3$ in the USADDR, MEDLINE and DBLP datasets.

size of its items larger than the USADDR dataset, with an average size of 75.7 against only 20.7 of the USADDR dataset. This means that for DBLP, our proposed methods will have more data to process in the second level, on the other hand, we will have a more significant reduction in memory consumption because the TRIE does not have a very deep level. In MEDLINE we have an average item size of 90.1, that is, the time difference will be even greater, but the memory consumption will be much smaller for the proposed methods (See section 5.4.3).

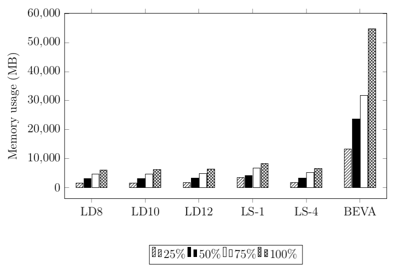
5.4.3 Varying Dataset Portions for Memory Consumption

When comparing memory consumption, we saw that our methods have a large advantage over baseline methods. In Figure 16 we show the memory consumption among the baselines methods in the USADDR, MEDLINE and DBLP dataset with portions of 25%, 50%, 75% and 100% of the dataset size. In the USADDR dataset (Figure 16a), the LD8 and LS-4 methods obtain the lowest memory consumption, a difference of three times and two times less compared to the memory consumption of the META and BEVA methods, respectively. In the DBLP dataset (Figure 16c), this difference in memory consumption of our methods is three times and five times less than the META and BEVA methods, respectively.

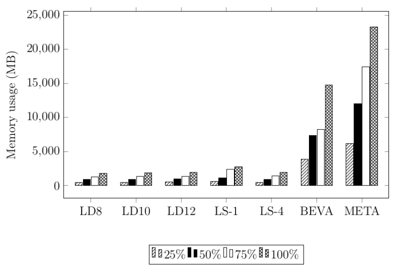
In the MEDLINE dataset - the largest dataset that we experienced, in which it has more than 14M words, the difference in memory consumption was very relevant. In the LD8 and LS-4 method we obtained 6GB and 6.5GB, respectively, while in the BEVA we obtained 56GB of memory - which represents a memory consumption of 800% more than our proposed methods, including LD8, LD10, LD12 , LS-1 and LS-4. In the META method we were unable to run 100% of the portion of this dataset with 64GB of available memory on our experiment server (Figure 16b). Therefore, this shows that it is essential to study and obtain the lowest memory consumption in error-tolerant query completion



(a) USADDR - Memory Consumption (MB) between LD8, LD10, LD12, LS-1, LS-4, BEVA and META for $\tau = 3$.



(b) MEDLINE - Memory Consumption (MB) between LD8, LD10, LD12, LS-1, LS-4 and BEVA for $\tau = 3$.



(c) DBLP - Memory Consumption (MB) between LD8, LD10, LD12, LS-1, LS-4, BEVA and META for $\tau = 3$.

Figure 16: Memory Consumption (MB) with portion of 25%, 50%, 75% and 100% of the dataset sizes in USADDR, MEDLINE and DBLP.

methods, because memory is a limited and costly resource, in which we not always have enough memory to run state-of-the-art algorithms from literature.

The use of the two-level approach in the context of error-tolerant query autocompletion algorithms proved to be quite effective. It is a good alternative to compress indexes in such algorithms and thus achieve low memory consumption. Given these results, we can finally answer the question raised at the beginning of this chapter. The two-level approach with the strategy of limiting the depth of the TRIE proved to be quite competitive when compared to the main algorithms available in the literature, especially when the size of the prefix query is smaller than the parameter max_{depth} or when the value of max_{depth} is reasonably higher. In addition, we achieved a much lower memory consumption, with 90% less memory used in the MEDLINE dataset, for example. The competitiveness of our method is due to the following: (1) In scenarios where we have a prefix query size smaller than the configured parameters or where the size of the set parameter is reasonably large, we have almost the same query processing time, especially when the prefix query size is small, which often occurs in real application scenarios. (2) given an approximate processing time, the low memory consumption reinforces that the method is very competitive.

6 Conclusion

Autocomplete has become an important and popular feature for search applications. Its use brings an important gain in the usability of search applications helping users to complete or correct misspelled queries. However, the main algorithms in the literature consume a high amount of memory, which is a limited resource with high cost.

This dissertation proposes implementation improvements in the BEVA algorithm, currently the state-of-the-art in the literature in order to achieve a reduction in its memory consumption and remain efficient in the query processing time. For this, we index only small prefixes of the queries in the TRIE and use such index to select subset of queries that are candidate for matching the prefix query already typed by the user. We then perform a sequential search in this subset to find the current list of suggestions that match the prefix typed by the user. We name this approach as two-level, since the query processing is performed partially using the TRIE, the first level, and partially performing sequential search in the query results, the second level. The results for such approach were quite competitive when compared to the main algorithms in the literature, being more effective in query processing times when the prefix query size is less than the depth of the TRIE or when the depth of the TRIE is reasonably deep. For memory consumption, we achieved a reduction of approximately 90% compared to other methods tested in MEDLINE dataset.

In addition, we studied two ways of building the TRIE, using the key-at-a-time (kaat) and level-at-a-time (laat) strategies. As far as we know, no previous works about error-tolerant query autocompletion has addressed the question about alternative strategies to build the index and authors use always the standard kaat strategy (Chaudhuri and Kaushik (2009), Ji et al. (2009), Zhou et al. (2016), Deng et al. (2016)). Our experiments show that we can improve the performance of BEVA when using laat strategy to build the index. We achieve reductions in processing time varying from 50% for $\tau = 1$ to 75% when $\tau = 3$ when using laat compared to the use of kaat.

As future work, it is necessary to further study and understand the the reasons why the LS method was not efficient in processing queries. Our initial hypothesis was that the cache policies of the machine would be affected by intermittent processing between the first and second level, unlike processing in the LD method which is completely transferred to the second level. However, initial data showed us that there is not such a large difference in cache misses between the LD and LS methods, thus it is still necessary to fully understand its inefficiency. Other works that can be studied are the use of the patricia tree as an alternative to the two-level approach, since the patricia tree also brings a significant reduction in memory consumption and it could not have the efficiency problems that the methods with two-level approach have when they process a large mass of data at the second level. In addition, testing patricia trees with the two-level approach could bring an even greater reduction in memory consumption.

Bibliography

Baeza-Yates, R. and G. Navarro
2000. Block addressing indices for approximate text retrieval. *Journal of the American Society for Information Science*, 51(1):69–82. Cited on page 30.

Baeza-Yates, R. A. and B. Ribeiro-Neto
1999. *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Cited on page 35.

Bast, H. and I. Weber
2006. Type less, find more: Fast autocompletion search with a succinct index. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, P. 364–371, New York, NY, USA. Association for Computing Machinery. Cited on page 29.

Broder, A., D. Carmel, M. Herscovici, A. Soffer, and J. Zien
2003. Efficient query evaluation using a two-level retrieval process. *International Conference on Information and Knowledge Management, Proceedings*, Pp. 426–434. Cited on page 38.

Chaudhuri, S. and R. Kaushik
2009. Extending autocompletion to tolerate errors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, Pp. 707–718, New York, NY, USA. ACM. Cited 8 times on pages 23, 29, 33, 39, 40, 47, 64, and 85.

da Costa Xavier, D.
2019. Um método em dois níveis para completação automática de sentenças. '. Cited 2 times on pages 35 and 38.

Deng, D., G. Li, H. Wen, H. Jagadish, and J. Feng
2016. Meta: an efficient matching-based method for error-tolerant autocompletion. *Proceedings of the VLDB Endowment*, 9(10):828–839. Cited 8 times on pages 23, 24, 26, 29, 35, 47, 63, and 85.

Fredkin, E.
1960. Trie memory. *Commun. ACM*, 3(9):490–499. Cited 2 times on pages 24 and 29.

Grabski, K. and T. Scheffer
2004. Sentence completion. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, Pp. 433–439. ACM. Cited on page 29.

Heinz, S., J. Zobel, and H. E. Williams

2002. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, 20(2):192–223. Cited on page 30.

Ji, S., G. Li, C. Li, and J. Feng

2009. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th international conference on World wide web*, Pp. 371–380. ACM. Cited 10 times on pages 23, 24, 26, 29, 30, 38, 39, 40, 47, and 85.

Levenshtein, V.

1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707. Cited on page 33.

Li, G., S. Ji, C. Li, and J. Feng

2011. Efficient fuzzy full-text type-ahead search. *The VLDB Journal—The International Journal on Very Large Data Bases*, 20(4):617–640. Cited 6 times on pages 23, 24, 26, 29, 30, and 39.

Manber, U., S. Wu, et al.

1994. Glimpse: A tool to search through entire file systems. In *Usenix Winter*, Pp. 23–32. Cited on page 30.

Miller, R. B.

1968. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, Pp. 267–277. ACM. Cited on page 25.

Nandi, A. and H. V. Jagadish

2007. Effective phrase prediction. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, P. 219–230. VLDB Endowment. Cited on page 29.

Navarro, G., E. S. De Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates

2000. Adding compression to block addressing inverted indexes. *Information retrieval*, 3(1):49–77. Cited on page 30.

Qin, J., C. Xiao, S. Hu, J. Zhang, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane

2019. Efficient query autocompletion with edit distance-based error tolerance. *The VLDB Journal*, Pp. 1–25. Cited on page 30.

Sussenguth Jr, E. H.

1963. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279. Cited on page 30.

Ukkonen, E.
1985. Algorithms for approximate string matching. *Inf. Control*, 64(1–3):100–118. Cited 3 times on pages 25, 34, and 35.

Xiao, C., J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane
2013. Efficient error-tolerant query autocompletion. *Proceedings of the VLDB Endowment*, 6(6):373–384. Cited 5 times on pages 23, 24, 26, 30, and 78.

Zhou, X., J. Qin, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa
2016. Beva: An efficient query processing algorithm for error-tolerant autocompletion. *ACM Transactions on Database Systems (TODS)*, 41(1):5. Cited 17 times on pages 23, 24, 26, 30, 35, 39, 41, 42, 43, 44, 45, 47, 56, 58, 59, 63, and 85.