Indexes and Algorithms for Scalable and Flexible Instant Overview Search

Pavlos Fafalios

Thesis submitted in partial fulfillment of the requirements for the

Masters’ of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisor: Assistant Prof. Yannis Tzitzikas

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.
The work is partially supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).
Indexes and Algorithms for Scalable and Flexible Instant Overview Search

Thesis submitted by

Pavlos Fafalios

in partial fulfillment of the requirements for the Masters’ of Science degree in Computer Science

THESIS APPROVAL

Author: Pavlos Fafalios

Committee approvals:

Yannis Tzitzikas
Assistant Professor, Thesis Supervisor

Dimitris Plexousakis
Professor, Committee Member

Constantine Stephanidis
Professor, Committee Member

Departmental approval:

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, September 2012
Indexes and Algorithms for Scalable and Flexible Instant Overview Search

Abstract

There is an increasing interest on recommending to the user instantly (during typing characters) queries and query results. This is evidenced by the emergence of several systems that offer such functionalities (e.g. Google Instant Search, Facebook for social searching, IMDB for movie searching, etc). In this thesis we consider more informative recommendations based on various precomputed aggregated information. Such recommendations can accommodate the products of various services like autocompletion, search-as-you-type, results clustering, faceted search, entity mining, etc. The instant presentation of these recommendations helps the user (a) to discover fast what is popular among other users, (b) to decide fast which (of the suggested) query completions to use, and (c) to decide what hits of the returned answer to inspect. In this thesis we focus on making this feasible (scalable) and flexible. Regarding scalability we elaborate on an approach based on precomputed information and we comparatively evaluate various trie-based index structures for making real-time interaction feasible, even if the size of the available memory space is limited. For improving the throughput that can be served we analyze and experimentally evaluate various caching policies. We report performance results over a server running on a modest personal computer (with 3 GB main memory) that provides instant services (in less than 140 ms) for millions of distinct queries and terabytes of precomputed information. As regards flexibility, in order to reduce user’s effort and to increase the exploitation of the precomputed information, we elaborate on how the recommendations can tolerate different word orders and spelling errors, assuming the proposed trie-based index structures. The experimental results revealed that such functionality significantly increases the number of recommendations especially for queries that contain several words.
Ευρετήρια και Αλγόριθμοι για Κλιμακώση και Ευέλικτη Στιγμιαία Επισκοπική Αναζήτηση

Περίληψη

Τα τελευταία χρόνια υπάρχει αυξανόμενο ενδιαφέρον για υπηρεσίες που συστήνουν στιγμιαία ερωτήματα και αποτελέσματα ερωτήματων καθώς ο χρήστης πληροφορεί το ερώτημά του χαρακτήρα-χαρακτήρα. Η τάση αυτή αποδεικνύεται από την συνεχή εμφάνιση διάφορων συστημάτων που προσφέρουν αυτή τη λειτουργικότητα. Ενδεικτικά παράδειγμα είναι το Google Instant Search, η χονδρική αναζήτηση στο Facebook, η αναζήτηση ταυτών στο IMDB, κα. Μεγάλη εργασία αυτή μελετά τη στιγμιαία παροχή πλούσιων συστάσεων βασισμένων σε προοπόλογησμένη πληροφορία συγκεκριμένων δίσκων (εξ αυτού ο όρος «στιγμιαία επισκοπική αναζήτηση»).

Η λειτουργικότητα αυτή είναι πιο βοηθητική από τις υπάρχουσες χρονικότερη, όπως την αυτόματη υπολογισμός και αναζήτηση των υπηρεσιών της εταιρείας της Google (query auto completion), της ομαδοποίησης αποτελεσμάτων (results clustering), της πολυδιάστατης πλοήγησης (faceted search navigation), της εξόρυξης οντοτήτων (entity mining), κ.α. Η στιγμιαία παροχή τέτοιων συστάσεων βοηθάει το χρήστη α) να ανακαλύψει χρήστη στοιχεία ποιες ερωτήσεις έχουν δημιουργεί μεταξύ των άλλων χρηστών, β) να αποφασίσει γρήγορα ποιο (προτεινόμενο) ερώτημα να επιλέξει και γ) να αποφασίσει ποια από τα εμφανιζόμενα αποτελέσματα να διερευνήσει. Επικεντρωμένη στο να προσφέρουμε αυτή τη λειτουργικότητα αποδυτικά και ευάλωτα, για να πετύχουμε υψηλή απόδοση, προσπαθούμε μια προσέγγιση που βασίζεται σε προοπόλογησμένη πληροφορία και αξιολογομένη συχρονικά διάφορες τεχνικές ευεργέτησης που βασίζονται σε ευεργέτες προθεμάτων (Tries) τα οποία αξιοποιούν τη διαδικασία χώρα μηνυής κόμμα και αν ευνοεί έγινε περιορισμένο μέγεθος. Συνάψιμα, για τη βελτίωση της διεπιχειρησιακής κατανόησης του πλασκομετα, αναλύουμε και αξιολογούμε πειραματικά διάφορες τεχνικές προσωρινής αποθήκευσης (caching). Να αναφέρουμε τις επιδόσεις της μεθόδου επί ένας συμβατικού υπολογιστή (με 3 GigaBytes χώρα μηνυής) ο οποίος δύναται να προσφέρει υπηρεσίες στιγμιαία επισκοπικής αναζήτησης (με απόκριση σε λιγότερο από 140 ms) για εκατομμύρια διαφορετικές ερωτήσεις και αξιοποιεί προοπόλογησμένη πληροφορία χάλματας terabyte. Σχετικά με την ευελπίζω να μειώσουμε την προσπάθεια του χρήστη και συνάψιμα να αυξήσουμε την εξόρυξη της προοπόλογησμένης πληροφορίας, μελετάμε πώς τα προτεινόμενα ευεργέτες μπορούν να προσφέρουν υπηρεσίες ανωτερος σε ορθογραφικά λάθη και στη σειρά των πληροφοριογραμμικών λέξεων. Τα πειραματικά αποτελέσματα χαρακτερίζουν ότι ο αριθμός των προτεινόμενων ερωτήσεων αυξάνεται σημαντικά, ειδικά όταν αυτά αποτελούνται από πολλές λέξεις.
Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη καθηγητή μου Χ. Γιάννη Τζίτζικα για την άψογη συνεργασία μας τα δύο τελευταία χρόνια, καθώς και για την ουσιαστική του καθοδήγηση και συμβολή στην όλοκλήρωση της παρούσας εργασίας. Θα ήθελα να εκφράσω επίσης τις ευχαριστίες μου στους καθηγητές Χ. Δημήτρη Πλεξουσάκη και Χ. Κωνσταντίνου Στεφανίδη για τη μεγάλη προμήθεια τους να συμμετέχουν στην τριμελή επιτροπή.

Ακόμα θα ήθελα να ευχαριστήσω το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας για την υποτροφία που μου προσέφερε και άλλη τη διάκειμα της μεταπτυχιακής μου εργασίας, καθώς και για την πολύτιμη υποστήριξη σε υλικο-γραφική υποδομή και τεχνογνωσία.

Πολλές ευχαριστίες θα ήθελα να εκφράσω επίσης στους φίλους μου για τη στήριξή τους και για όλες τις στηνες μου οιοραστήρια όλο αυτό τον χαιρό.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαίτερα τους γονείς μου Μάρκο και Αναστασία, και την αδελφή μου Νάντια, για την υποστήριξη και την αγάπη που με περιβάλλουν.
στούς γονείς μου
# Contents

Table of Contents i  
List of Tables iii  
List of Figures v  

1 Introduction 1  

2 Motivation and Related Work 5  
2.1 Human Decision Making during Web Searching . . . . . . . . . . . . . . . . . 5  
2.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  
2.2.1 Instant Search on the Web . . . . . . . . . . . . . . . . . . . . . . 8  
2.2.2 Instant Search in Documents . . . . . . . . . . . . . . . . . . . . 8  
2.2.3 Instant Search in Relational Systems . . . . . . . . . . . . . . . . 9  
2.2.4 The Distinctive Difference of our Approach . . . . . . . . . . . . . 10  

3 Trie-based Index Structures for Instant Overview Search 13  
3.1 Trie Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14  
3.2 Indexes to External Random Access Files . . . . . . . . . . . . . . . . . . . 14  
3.3 Combining Trie Partitioning and Indexes to External Files . . . . . . . . . 15  
3.4 Index Construction and Update . . . . . . . . . . . . . . . . . . . . . . 16  

4 Throughput and Caching 19  
4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
4.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
4.3 Caching Schemes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20  

5 On “Flexible” Recommendations 23  
5.1 On Relaxing the Word Order . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
5.1.1 Two-Word Queries . . . . . . . . . . . . . . . . . . . . . . . . . . 24  
5.1.2 Queries with Many Words . . . . . . . . . . . . . . . . . . . . . . 24  
5.1.3 Incremental Suggestions . . . . . . . . . . . . . . . . . . . . . . . 26  
5.1.4 Synopsis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27  
5.2 Typo-Tolerant Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28  

i
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Motivation and Related Work</td>
<td>28</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Implementation Approaches</td>
<td>28</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Synopsis</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>Combining Typos and Different Word Orders</td>
<td>30</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Motivation</td>
<td>31</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Implementation Approaches</td>
<td>31</td>
</tr>
<tr>
<td>5.4</td>
<td>On Recommendation Ranking</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>Experimental Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>6.1</td>
<td>Evaluation of Index Structures</td>
<td>33</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Evaluation Aspects</td>
<td>33</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Data Sets and Setup</td>
<td>34</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Results on Trie’s sizes</td>
<td>35</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Average Retrieval Time</td>
<td>37</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Large Scale Evaluation of PTIE</td>
<td>39</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Construction and Update Time</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>Selecting the Right Index</td>
<td>40</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Autonomic Selection of the Right Index</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>Evaluation of Caching Schemes</td>
<td>43</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Served Queries</td>
<td>44</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Average Response Time</td>
<td>44</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Factors that affect the Experimental Results</td>
<td>45</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Synopsis</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>Evaluation of “Flexible” Recommendations</td>
<td>47</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Retrieval Time</td>
<td>47</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Number of Additional Suggestions</td>
<td>48</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>7</td>
<td>Server’s Benefits</td>
<td>51</td>
</tr>
<tr>
<td>8</td>
<td>Applications of IOS</td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td>IOS and Results Clustering</td>
<td>55</td>
</tr>
<tr>
<td>8.2</td>
<td>IOS and Metadata-based Groupings</td>
<td>56</td>
</tr>
<tr>
<td>8.3</td>
<td>IOS and Entity Mining</td>
<td>57</td>
</tr>
<tr>
<td>8.4</td>
<td>IOS and Linked Open Data</td>
<td>59</td>
</tr>
<tr>
<td>8.5</td>
<td>IOS and Vertical Search</td>
<td>59</td>
</tr>
<tr>
<td>8.6</td>
<td>Synopsis</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>Concluding Remarks</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>63</td>
</tr>
</tbody>
</table>
# List of Tables

5.1 Synopsis of the implementation approaches of word-order independent search. .......................................................... 26
5.2 Synopsis of the implementation approaches of typo-tolerant search. ................................................................. 30
6.1 The query logs used on experiments. ................................................................. 34
6.2 Trie’s construction time. ................................................................. 40
6.3 Estimated size of the trie’s required data. ................................................................. 42
6.4 Average retrieval time of word-order independent and typo-tolerant suggestions. ................................................................. 47
6.5 Average number of additional suggestions per query when tolerating different word orders. ................................................................. 49
6.6 Average number of additional suggestions per query when tolerating spelling errors. ................................................................. 49
6.7 Efficient support of word-order independent search, typo-tolerant search and their combination over the proposed index structures. ................................................................. 50
## List of Figures

1.1 IOS indicative screen dump. ........................................ 3  
2.1 Human decision making and the typical web search process versus the proposed instant overview search process. ................. 6  
2.2 A search-as-you-type system. ..................................... 7  
3.1 Extending the query trie by two additional strings. ............ 13  
3.2 Distributing queries to partitions .................................. 15  
3.3 Trie-based index structures for Instant Overview Search ...... 16  
3.4 Incremental trie update. ............................................ 17  
4.1 Distributing queries by the first two characters, having used a query log from the Excite web search engine (40,000 queries) ............. 20  
5.1 Trie’s active nodes for input meri and edit distance = 1. ......... 29  
6.1 Size of the trie - experimental results for SET and PET .......... 35  
6.2 Size of the trie - experimental results for STIE and PTIE ........ 36  
6.3 Comparison of the trie’s size. ...................................... 36  
6.4 Average retrieval time - experimental results for SET and PET .. 37  
6.5 Average retrieval time - experimental results for STIE and PTIE .. 38  
6.6 Comparison of the average retrieval time. ......................... 38  
6.7 Retrieval time analysis of PTIE. ...................................... 41  
6.8 Trade-off for the four indexing approaches. ....................... 41  
6.9 Percentage of queries that were served from the cache fast, with delay and that were not served, for various memory partition policies. 44  
6.10 Average response time of various memory partition policies. ... 45  
6.11 Evaluation of the main factors that affect the caching experiments. 46  
8.1 IOS and clustering of the top results. ............................ 56  
8.2 IOS and metadata-based groupings. ................................ 57  
8.3 IOS and entity mining. .............................................. 58  
8.4 IOS and entity mining - Limiting the search space. ............. 58  
8.5 The steps of a standalone and meta search process and IOS’ benefits. 59
Chapter 1

Introduction

There is an increasing interest on recommending to the user instantly (after each keystroke) queries and query results. This is evidenced by the emergence of several systems that offer such functionalities. Indicative examples include Google Instant for plain web searching, Facebook\(^1\) which shows the more relevant friends (pages, etc), IMDB\(^2\) which shows the more relevant movies (actors, etc) together with a photo, and so on.

These instant recommendations apart from enabling the user to see what is popular among other users, they allow the user to stop typing if the desired results have already been recommended. If on the other hand the sought results are not there, the user can continue typing or change what he has typed so far. In general, we can say that the user adapts his query on-the-fly until the results match what he wants. Such services save user’s time and effort also because people type slowly but read quickly, implying that the user can scan a results page while he types. For example, according to Google\(^3\), Instant Search can save 2-5 seconds per search.

In this thesis we generalize and propose a more powerful search-as-you-type functionality which apart from showing on-the-fly only the first page of results of the guessed query, it can show several other kinds of supplementary information that provide the user with a better overview of the search space. We call this paradigm of searching Instant Overview Searching, for short IOS. The supplementary information can be the result of various tasks like results clustering, metadata-based groupings, entity mining, etc., and is visualized and exploited according to the faceted exploration interaction paradigm \([46]\): when the user clicks on an cluster (or group of results in general), the hits are restricted to those that contain that entity, and so on.

However, the instant provision of such services for large number of queries, big amounts of precomputed information, and large number of concurrent users is challenging. To tackle this challenge, we propose enriching the trie structure \([50]\)

\(^{1}\)http://www.facebook.com/
\(^{2}\)http://www.imdb.com/
\(^{3}\)http://www.google.com/insidesearch/instant-about.html
which is used for query autocompletion with the various precomputed supplementary information. For example, and for the case of clustering, for each query in the trie we keep the outcome of the results clustering algorithm, specifically the cluster label tree (the cluster label tree is a tree of strings, each being the label, readable name, for a cluster). This choice greatly reduces the required computation at interaction time, however it greatly increases the space requirements of the trie. For this reason in this thesis we describe and comparatively evaluate various index structures. Essentially each index structure is actually a method for partitioning the information between the main and the secondary memory. We have conducted a detailed performance evaluation according to various aspects such as system’s response time and main memory cost, which reveals the main performance trade-off and assists deciding which index structure to use according to the available main memory and the desired performance. Specifically, we show that with partitioned trie-based indexes we can achieve instant responses even if the precomputed information is too large to fit in main memory. Apart from these, we detail the selection of the trie-based index structures according to several parameters of the precomputed information (e.g. size of cluster label tree, size of top-K results), and we describe an incremental method for updating the indexes.

Furthermore, we propose and evaluate various caching schemes that apart from speeding up these services they increase the throughput that can be served, and we show how the recommendation service can tolerate different word orders, spelling errors, or both, for reducing user’s effort and increasing the exploitation of the precomputed information. Finally, we discuss the benefits of this functionality for the server’s side and we demonstrate various novel applications of IOS that exploit and recommend different kinds of supplementary information.

The motivation for focusing on scalability is obvious (without tackling this issue it is not feasible to provide such services in real applications), while the motivation for enhancing flexibility, and how it assists decision making, can be made evident also from the following example.

**Examples 1.1.** Consider a user that wants to write an article about the consequences (mainly the economic) of the earthquake that hit Haiti in 2010. For this reason he starts typing letter by letter the query Haiti earthquake economic consequences. In each keystroke, a list of suggestions appears instantly, which represents what is popular for the current input. Furthermore, for the top suggestion, user is able to view the top hits and a clustering of the search space. After having typed the string Haeti e (notice that the word Haiti was misspelled), the user gets the suggestions Haiti education, Haiti economy, Haiti earthquake, Haiti earthquake effects, Embassy of Haiti, and environment of Haiti. Note that all the suggestions do not contain the user’s input word Haeti, but the “similar” word Haiti. Furthermore the last two suggestions contain the word Haiti and a word that starts with “e” (Embassy and environment) in different word order than that of user’s input. User notices that the suggestion Haiti earthquake effects satisfies his search need. Thus, by pressing the down arrow key he navigates and selects the particular suggestion. At that time, the
top hits of the selected suggestion and a clustering of the results appear \textit{instantly} (note that user has not yet submitted the query). Afterwards, the user notices that there is a cluster with label \texttt{economic(9)}, which actually contains 9 web pages that describe the economic damage of the earthquake in Haiti. The user clicks that label and instantly the results of this cluster appear in his screen. All these results are very likely relevant to his information need. Furthermore, the user notices that there are many other interesting cluster labels, such us \texttt{health(8)}, \texttt{nature(12)} and \texttt{pictures(6)}. So, he decides to include some pictures in his article and to write one more paragraph for the \texttt{health} problems that Haitian people faced because of the earthquake. Figure 1.1 depicts a screen dump of the aforementioned example.  

\begin{figure}[h]
    \centering
    \includegraphics[width=\textwidth]{figure1.png}
    \caption{\textsc{ios} indicative screen dump.}
\end{figure}

In general we can say that the instant presentation of such recommendations helps the user to discover fast what is popular among other users (since the suggestions are based on log analysis), to decide fast which (of the suggested) query completions to use, and what hits of the returned answer to inspect. It is widely accepted that human decision making is a complex process \cite{53} and this is true also in the context of information searching. In this context, i.e. during the search process, the user has to make two main kinds of decisions: the first concerns what query or queries to submit, the second is about what hits of the answers to inspect. We further analyze this process and discuss the impact of our work on decision making at Section 2.1.

The rest of this thesis is organized as follows. Chapter 2 discusses decision making in the context of searching and related work. Chapter 3 introduces the
CHAPTER 1. INTRODUCTION

partitioned trie-based index structures and details the main steps of trie construction and of an incremental method for refreshing them. Chapter 4 introduces the caching mechanism. Chapter 5 focuses on making word-order independent and typo-tolerant search feasible also over the partitioned trie-based index structures. Chapter 6 reports extensive comparative experimental results for evaluating the proposed index structures and the caching schemes, demonstrating the feasibility of the flexibility features and for quantifying their benefits in decision making. Chapter 7 discusses the benefits of IOS for the server’s side, while Chapter 8 presents possible applications including prototypes that we have designed and developed. Finally, Chapter 9 concludes and identifies issues that are worth further research.

Publications Derived by this Thesis

The results of this thesis were published in the international conferences WISE [25], WWW [23] and IRFC [22], and in the national conference HDMS [24]. Specifically, in [25] we elaborate and experimentally evaluate the proposed index structures, in [23] and [24] we demonstrate a family of IOS prototypes, we perform a large-scale experimental evaluation of the most scalable index structure, and we evaluate various caching policies, while at [22] we focus on enriching the classical web searching with entity mining that is performed at query time.
Chapter 2

Motivation and Related Work

At Section 2.1 we discuss the motivation from a decision making point of view, while at Section 2.2 we discuss related works from a technical point of view.

2.1 Human Decision Making during Web Searching

Human decision-making is defined as a cognitive process in which a preferred option or a course of action is chosen from among a set of alternatives, based on certain information or considerations. In our context, and during the search process we can say that the user has to make two main kinds of decisions: what query or queries to submit, and what hit(s) of the returned answers to inspect.

As regards what query to use, we should stress that there are several possible queries for expressing the same information need. This means that the user has to decide which one to type. The recommended query completions assist the user to complete what he has typed based on the popularity of the queries that other users have submitted in the past, and also enable him to identify possible misspellings. Furthermore, the supplementary information that is delivered instantly at each keystroke (corresponding either to a character or to an arrow button for shifting up and down in the list of suggested query completions) allows the user to predict the consequences of each choice, and thus help him to make a more “justified” decision.

Concerning the second kind of decisions, i.e. what hits to inspect, the supplementary information can greatly assist him in locating hits that would be very difficult and laborious to find (i.e. those that are not in the first page of results). Note that users tend to look only the first page of results not only because it is more time consuming and laborious to click, get and inspect the subsequent pages, but also because they feel that the hits in the first page are more reliable. As a consequence, users can be easily biased, and they are vulnerable to web spamming, as well as to the various search optimization techniques. We should also note that although search engines are programmed to rank web sites based on their popularity and relevancy, empirical studies indicate various political, economic, and social
biases in the information they provide [49, 56]. Our thesis is that the provision of overview information alleviates also this problem since the users have the ability to see an interesting cluster label and by clicking on that label to end up to hits that could be low ranked (i.e. not in the first pages of results), practically to hits they would never inspect. Furthermore, the instant provision of supplementary information can assist the user in refining, or just differentiating, his information need. This can be realized either by focusing at a particular set of results (e.g. by clicking on a cluster name), or by starting the formulation of a new query. For instance, in our previous example, although the user initially wanted to search for economic effects, the appearance of the supplementary information made him realize that there are also health effects, and to decide that he would like to get information about them.

An analysis of the process is given at Figure 2.1. At the core of the left diagram we can see a general model of the human decision making process. Notice that it is cyclic in the sense that each action and its evaluation leads to a different situation that may trigger a new information need. The outer process shows how this model maps to the steps of the classical (“search in a box”) information retrieval process of typical search systems. We can say that the user gets no information about the step Options, nor any assistance on the steps Choose or Act, nor any aid in Evaluate, i.e. on the inspection of results, apart from the provision of snippets.

At the right diagram we can see that the process we suggest includes a step named “see suggestions and inspect the expected results during typing letter by letter” which is related to the phases Options, Choose, Act, and Evaluate. In
2.2. RELATED WORK

A search-as-you-type system computes answers to keyword queries as the user types in keywords letter by letter. Figure 2.2 illustrates the architecture of such a system. At each keystroke of the user the browser sends (in AJAX style) the current string to the server, which in turn computes and returns to the browser a list of suggestions and the top hits of the top-suggested query.

There are several works describing letter-by-letter searching for various kinds of sources, e.g. for relational databases [31, 57, 38, 39], or documents [6, 37]. To cope with the efficiency problem, they index the entire collection and most use in-memory structures (i.e. the entire index is loaded in main memory). In the next sections, we describe in more detail the works that have been done in that area and the differences of our work.

---

other words this step assists all these phases. In particular, in a keystroke basis, the user is getting information about **Options** (query completions), aid for **Choose** (appearance of supplementary information), aid for **Actions** (no need to type the entire query so the actions are less laborious), aid for **Evaluation** (the user can see the number of results, clusters, etc.), and aid for the transition to a new **Situation**.

Finally, we could say that in contrast to the typical recommender systems in which the available choices are usually less than millions (e.g. in the domain of movies, hotels, books, products), in the context of web searching the choices are billions. Moreover there are no fixed features (e.g. as in products), their number can be large and they are not homogeneous in an open environment like the Web (meaning that approaches like [11] cannot be applied).
2.2.1 Instant Search on the Web

Recently, Google adopted the search-as-you-type functionality on its web search\(^1\). There are many other web search systems that support similar functionality like EasySearch\(^2\), which moreover groups on-the-fly the results and Keyboardr\(^3\). Moreover, YouTube Instant\(^4\) searches and plays the top suggested video as the user types a query, while Wiki Instant\(^5\) shows instantly the Wikipedia page of the top suggestion.

However, none of these systems offers instant recall-oriented search services, e.g. clustering of the results or ability for faceted browsing as the user types a query letter by letter. The idea of enriching the classical query-and-response process of current web search engines, with static and dynamic metadata for supporting exploratory search was proposed in [44] and it is described in more detail (enriched with the results of a user-based evaluation) in [43]. However, no instant behavior had been proposed or provided by that work.

2.2.2 Instant Search in Documents

A closely related study is the work by Bast et al. [5, 6, 7, 4]. This is the only work we managed to found that does not load everything in main memory. They introduce a semantic search engine (so-called CompleteSearch\(^6\)), in which a user types in keywords letter by letter and the system finds on-the-fly and presents precomputed records and semantic classes that match these keywords. To tackle the efficiency challenge, the authors introduce a indexing data structure, called “HYB”. The basic idea behind that structure is the usage of compressed precomputed inverted lists for union of words (the union of all lists for an arbitrary word range). This compressed index is not loaded in main memory but is stored in a single file with the individual lists concatenated and an array of list offsets at the end. The vocabulary is stored in a separate file. The main unit of processing is a block (a range of words). A block consists of all pairs (word,document) and is sorted by document id. HYB consists of a collection of such blocks and supports two operations on the block: (1) intersection with a sorted list of documents ids and (2) intersection with a list of word ids. The evaluation of HYB was performed according to the required index space and the query processing time. The results illustrate that HYB uses no more space than a state-of-the-art compressed inverted index and yields fast average response times of one tenth of a second (7.4 GB documents size, 500 MB index size, 200 ms average processing time). Details of the experimental evaluation can be found in [6]. That work differs from ours in the following aspects: a) CompleteSearch is not a generic search engine, but

\(^1\)http://www.google.com/instant/
\(^2\)http://easysearch.webs.com/home.htm
\(^3\)http://keyboardr.com/
\(^4\)http://ytinstant.com/
\(^5\)http://wikinstant.com/
\(^6\)http://search.mpi-inf.mpg.de
2.2. RELATED WORK

is based on specific known datasets (like Wikipedia and DBLP) with predefined semantic categories, b) it’s suggestion system is word based, not query based, i.e. it suggests only words that match user’s current input, not whole queries, c) it focuses on compression of index structures, especially in disk-based settings, while IOS uses partial in-memory indexes, d) in general CompleteSearch helps users to formulate a good semantic query while IOS helps users to locate directly and fast what he is looking for.

In [37] the authors study an information-access paradigm for XML data in which the system searches for precomputed results, as the user types in query keywords. This system keeps in memory a trie data structure which contains the tokenized words in XML data. A trie’s leaf node contains an inverted list of the XML elements that match the predicted word. The experimental evaluation illustrates that this system can achieve very low server running time (about 30 ms).

2.2.3 Instant Search in Relational Systems

The work in [31] formalizes the problem of interactive search on a relational table, finding relevant records that have keywords matching query keywords approximately. In order to achieve an interactive speed, they use precomputed and cached results. In particular, they use a trie data structure for indexing the words of a relational table. Each trie’s leaf node has an inverted list of ids of records that contain the corresponding word with additional information such as the attribute in which the keyword appears and its position. For queries with multiple keywords, they use forward lists of keyword ids for checking whether a record matches query keyword conditions. The evaluation of this work illustrates that the proposed method achieves efficient retrieval time even for large relational tables (1.25 GB data size, 164 MB trie size, 445 MB inverted-list size, 454 MB forward-list size, 20 ms average response time).

In [57] a cache-based search-as-you-type system for relational data is described. It allows a user to specify his keywords in multiple input boxes on a form and get instantly the results and all attribute values of the edited input box that match user’s input. To cope with the efficiency problem, the authors suggest to keep in memory a so-called “dual-list trie” for each local table (an original “global” table is partitioned into several “local” tables, each for a specific attribute). A dual-list trie has two inverted lists in each leaf node, one for the local ids and one for the global ids of a word. The experimental evaluation of this system illustrates that the proposed method achieves efficient average query time (800,000 records, 300 MB index size, 30 ms average query time).

The work in [38] formulates the problem of search-as-you-type in which the relational data are modeled as a database graph. They use a trie to index the tokenized words in the database. For each leaf node, they assign a unique id to the corresponding word in the alphabetical order. Each leaf node has an inverted list of ids of vertices in the database graph that contain the word. Each trie node
has a keyword range of $[L, U]$, where $L$ and $U$ are the minimum and maximal keyword id in the subtrie of the node respectively. It also maintains the size of the union of inverted lists of its descendant leaf nodes, which is exactly the number of vertices in the database graph that have the node string as a prefix. Moreover, to efficiently predict complete queries, they constructed a two-tier trie (each leaf node of the initial trie contains a trie). The experimental evaluation illustrates that the proposed system is efficient (470 MB data set size, 10,000 subgraphs, 130 MB index size, 20 ms average search time).

In [39] the authors introduce a query model where the user does not have to submit a complete query. It provides instantly predicted keywords when a user submits letter-by-letter a few characters of the underlying data in order (e.g. for the input “mrd” the system proposes the words “maradei” and “maradona”). This model uses precomputed results and in memory indexes to achieve high interactive speed. In particular, they use a trie to index the words in the relational table. Each leaf node of the trie has an inverted list of ids of records that contain the corresponding word, with additional information such as the attribute in which the keyword appears and its position. Moreover, in order to get the position of characters in the trie, they build an inverted list for each character, with the ids of the node that has a label of the character. The experimental evaluation illustrates the efficiency of this system (1,062,361 records, response time lower than 10 ms).

In [34], the authors present a system that provides on-the-go, context-aware assistance in the SQL composition process. In particular, this system aims to help the non-expert database users, who need to perform complex analysis on their large-scale datasets, but have difficulty writing SQL queries. As a user types a query, the system recommends possible additions to various clauses in the query, using relevant snippets collected from a log of past queries. The system was evaluated on two real datasets regarding the quality of the recommendations and its efficiency (response time).

2.2.4 The Distinctive Difference of our Approach

The aforementioned systems either need a high amount of main memory in order to load the indexing structures or use disk-based compressed formats like [6]. We have not managed to find any work that partitions the index and can adapted to a system’s main memory capacity. Our approach requires loading only the part of the index that is needed, by dividing the index and exploiting the secondary memory. In this way, we can reduce the amount of the required main memory and thus we are able to exploit large amounts of precomputed information (e.g. also images for visualization). In addition, for each suggestion our system saves an overview (e.g. cluster label tree) and a part (e.g. top-10) of the results. Note that the space required is independent of the size of the collection; it is affected only by the size of the query log that we want to consider.

Concerning caching, we have not managed to find any work on instant search that caches a part of the index. Nevertheless, this problem resembles caching in
2.2. RELATED WORK

web search engines and for this reason at Section 4.2 we review in brief the work that has been done in that area and the differences of our work.

As regards typo-tolerant and word order independent query suggestions, at Section 5.2.1 we discuss the main works about spelling correction for query completions for in-memory indexes. None of the related works elaborate on how to offer such functionality in a partitioned index structure.
Chapter 3

Trie-based Index Structures for Instant Overview Search

To tackle the requirements of IOS, we propose the adoption of trie-based index structures because they allow fast lookup with cost analogous to the length of the string that we search and not the number of the stored queries. Specifically, for each query string stored in the query log, we extend its node in the trie with two additional strings (as Figure 3.1 depicts). The first string contains the first page of results of the query (i.e. an HTML string containing for each hit its title, snippet and URL). The second string contains the supplementary information that we want to offer, for example the cluster label tree, the metadata-based groupings of the results, etc. Note that we prefer to store both strings in HTML format in order to save time while presenting the results to the user (no need for any post-processing).

Obviously, such enrichment significantly increases the size of the trie, since for each query we have to save two additional long strings. Specifically, for each logged query of \(n\) characters, the trie keeps a node of about \(n = 16\) bytes. However, the string that represents the first page of results can be about 60,000 bytes and the corresponding string of the cluster label tree can be about 30,000 bytes in common.

![Figure 3.1: Extending the query trie by two additional strings.](image)

13
real settings (including the characters of the HTML code for both strings). Below we propose methods and index structures for managing this increased amount of data.

### 3.1 Trie Partitioning

The first idea is to adopt the *trie partitioning* method proposed in [33], i.e. to partition the enriched trie to a number of subtries based on the first character(s) of the queries. According to that method, only one “subtrie” of much smaller size is loaded in main memory for serving one request; the subtrie containing the queries whose starting characters are those that the user has typed so far.

In particular, if \( Q \) is the set of all distinct queries in the log, let \( QT \) be the full trie over \( Q \) and \( Q_{\text{Subtries}} = \{ qst_1, \ldots, qst_N \} \) be the partition of \( QT \) to \( N \) subtries according to [33] which is based on the first \( k \) characters. Each subtrie \( qst_i \) contains a set of queries \( \text{queries}(qst_i) \) whose starting characters belong to a specified set of characters. For example, if we assume that the query log contains queries starting with latin characters only and we decide to partition the trie based on the first character \((k = 1)\), then we can divide a trie into two subtries: one containing all query strings \( q \) where \( q[0] \in \{a, b, \ldots, m\} \) and another containing all \( q \) where \( q[0] \in \{n, o, \ldots, z\} \).

Figure 3.2a (from [33]) shows how queries are distributed to subtries when trie partitioning is based on the first character only \((k = 1)\), the first two characters \((k = 2)\), and the first three characters \((k = 3)\)\(^1\). Figure 3.2b demonstrates how smoothly the queries are distributed to the subtries, by showing the standard deviation for each distribution, \((\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2} )\), where \( N \) is the number of subtries, each \( x_i = |\text{queries}(qst_i)| \) represents the number of queries in subtrie \( i \), and \( \mu \) is the mean value of these numbers. It is evident that for \( k \geq 2 \), the distribution is dramatically smoother (closer to the ideal) than in the case of \( k = 1 \).

Since users seldom change their initial queries [51], dividing the trie in this way implies that once the appropriate subtrie has been loaded (during the initial user’s keystrokes), the system can compute the completions of the subsequent requests using the same subtrie.

### 3.2 Indexes to External Random Access Files

Now suppose the case where we do not have enough main memory to load the enriched trie (or subtrie). In such case we can save the results of the preprocessing steps (e.g. first hits, cluster label tree, facets, etc.), in one or more different files. Consequently, the trie for each query entry has to keep only three numbers: (a) one for the file, (b) one for the bytes to skip and (c) one for the bytes to

\(^1\)It was used a query log from the Excite search engine (http://www.excite.com) which contained about 25,500 distinct queries.
Combining Trie Partitioning and Indexes to External Files

We can further reduce the size of the trie that is loaded in the main memory by combining the last two approaches (trie partitioning and trie with indexes to external files). Such approach requires very small amount of main memory, however it requires more time for loading the appropriate subtrie in the main memory and reading the data from the pointing file.

To clarify the pros and cons of the above choices, at Section 6.1 we comparatively evaluate the following approaches (depicted at Figure 3.3):

(a) (SET) **Single Enriched Trie.** Here each node of the query trie is extended with two strings: one for the first page of results and one for the supplementary information (e.g. results clustering). The entire enriched trie is kept in main memory (Figure 3.3a).

(b) (STIE) **Single Trie with Indexes to External files.** Here each node of the query trie is enriched with pointers to external random access files, where a pointer consists of 3 numbers: file number, bytes to skip, and bytes to read (Figure 3.3b). The single (query) trie is kept in main memory.

(c) (PET) **Partitioned Enriched Trie.** Here the enriched trie, i.e. the one
described in (a), is partitioned to several subtries, each stored in a separate file (Figure 3.3c). The subtrie that is needed in order to serve a request is loaded in main memory at request time.

(d) (PTIE) Partitioned Trie with Indexes to External files. Here the (query) trie is partitioned but each subtrie is enriched with pointers to external random access files (Figure 3.3d). Here also the subtrie is loaded in main memory at request time.

3.4 Index Construction and Update

For the construction of the trie (or subtries), the main tasks are a) the analysis of the query log, b) the execution of each distinct query in order to get the first page of results and the supplementary information, c) the creation of the trie’s file. We precompute and store also the score of each distinct query, which is used by the autocompletion algorithm. Specifically, we use the method proposed in [33] that takes into account the frequency of the query and the frequencies of the queries that have this as prefix. One can easily see that the time required by these tasks depends on the particular algorithms that are employed (for query evaluation, clustering, etc).

Note however that the index should be updated periodically, based on the contents of the constantly changing query log and the new fresh results of the underlying web search engine. One policy is to update the index daily. Since the construction of the trie from scratch is a time consuming task, it is worth providing
an incremental method. We propose the following incremental update procedure:
create the trie of the new query log and then merge the old “big” trie with the
new one which is much smaller. Figure 3.4 illustrates how the incremental update
is performed in the SET index. If an entry of the new trie does not exist in the
initial trie then we just add the new query with all its data to the initial trie. If
however an entry of the new trie exists in the initial trie then we have to update
its results, its clusters, its score and its date. The procedure is almost the same in
case of STIE. During the creation of the new trie, we could either store the results
and the supplementary information in new random access files, or append them in
the existing ones. However, in case of PET and PTIE the update is a bit more
complex because multiple subttries have to be merged.

In case we want to keep stable the size of the trie (or subttries), we can remove
the “old” queries according to a date threshold. In this case we have to perform
a trie traversal and remove those queries that are chronologically before the date
threshold. This procedure can be the first step in the incremental update of the
trie (remove the “old” queries from the initial trie). Furthermore, the incremental
update can be performed faster if in each query submission, the system stores the
results and the supplementary information. This allows the system to use these results (although they are not the most "fresh") during the update of the trie, saving in this way considerable time.
Chapter 4

Throughput and Caching

4.1 Motivation

Consider a scenario where a large number of users start typing queries at the same time. A rising question is how each index structure (either SET, STIE, PET or PTIE) reacts and in what cases the system will get overloaded. In the SET and STIE approaches, the trie is loaded only once at system’s start-up and therefore, the number of the requests that the system can serve depends only on the server’s request/session capacity. Note that in the STIE approach, the usage of random access files does not cause any additional load to main memory. However, PET and PTIE require loading multiple subtries, i.e. the appropriate subtrie for each user’s keystroke, and this could overload the system since multiple subtries will have to be hosted at main memory. This is true especially for PET whose average subtrie size is bigger than that of PTIE (because the subtries of PET contain the supplementary information, while those of PTIE do not). The small size of PTIE implies that many different subtries will be loaded if many users concurrently type, however these subtries have low loading time because they are small.

The main question is how to exploit the available main memory in order to serve concurrently several users as fast as possible without overloading the system. Is it better to load in memory the appropriate subtrie at request time and remove it afterwards, or is it better to keep it in memory for a period of time? One general method to alleviate the throughput problem is to adopt a caching mechanism. In this way, if the requests of two or more users require loading the same subtrie, the loading will be done only once. Obviously, the system has to look the cache before loading a subtrie.

4.2 Related Work

This problem resembles caching in web search engines. Below we review in brief, the work that has been done in that area. A large body of work has been devoted to results caching and posting lists caching. [41] introduced caching query results as a
means to reduce the response time of a search engine. Saraiva et al. [48] propose a
dynamic caching system for caching query results and posting lists. Long and Suel
[40] recommend a caching architecture that includes an intermediate level with on-
disk caching of the intersections of the posting lists of frequently occurring pairs of
terms. Fagni et al. [26] employ a hybrid results caching scheme where the available
cache space is split into a static and a dynamic segment in order to capture both
recent and frequent queries. Baeza-Yates et al. [2] explore the impact of results
caching and posting lists caching in web search engines and show that posting
lists caching offers higher hit ratios than results caching. Lembel and Moran [36]
propose a cache replacement policy based on the probability of a result page being
requested. Skobeltsyn et al. [52] propose an architecture that combines results
caching with index pruning to reduce query processing load while Gan and Suel
[27] studied weighted results caching techniques which consider both the frequency
of the queries and their estimated execution costs. Finally, more recent works focus
on the freshness of the cached results [13, 10].

Our case differs from the classic caching problem in the sense that we must
offer much faster response time (of a few milliseconds) in order to present instantly
the results and the supplementary information as the user types a query letter-by-
letter. Furthermore, in our case we have to manage bigger amounts of information
(cluster label tree, facets, etc).

### 4.3 Caching Schemes

One approach is to adopt a dynamic caching scheme, i.e. to start from an empty
cache and put in it each requested subtrie. If a new subtrie has to be loaded
and the cache is full, the system replaces an existing cached subtrie (e.g. the less
frequent one that is not in use by a session) with the new one. It could also refresh
the cache by removing the old subtries, i.e. the subtries that are not in use for
a specific time threshold. In such a dynamic caching mechanism, the cache size
can be set to be equal to the maximum number of loadable subtries that can fit
in main memory at a given time. In a static approach, we could keep in cache the more frequently used subtries based on a past log analysis. Figure 4.1 shows the distribution of the top-20 first two characters in queries stored in a log of 40,000 queries of Excite\(^1\) web search engine. We observe that the prefix “co” is the most frequent prefix appearing in 2.39% of all queries. Moreover, the top-20 prefixes of size 2 appear in the 28% of all queries and this means that if we keep cached these subtries we expect a cache hit ratio of 28%. Taking into account the above facts, we could keep always in memory the subtries of the most frequent first two characters.

In an hybrid caching mechanism, we combine dynamic and static approach. In this way, we keep always in memory the subtries of the most frequent queries (static part) and keep an amount of memory for loading the subtries that are not in the cache (dynamic part). For example, if the available memory capacity can host \(MC\) subtries, we can split it into two parts: a static (with size \(S\)) and a dynamic (with size \(D = MC - S\)).

Algorithm 1 describes the above hybrid caching mechanism. \(StaticMap\) is a map for the static part of the cache and keeps information for each cached subtrie (e.g. frequency, last used time, subtrie’s file, etc), while \(DynMap\) is the corresponding map for the dynamic part. We observe that in case we have a request for a subtrie that lies in the static part (line 3), the system can instantly retrieve the required data, otherwise it must check the dynamic part (line 5). In the dynamic part, if we have a request for a subtrie that is not in the cache (line 10) and the cache is full (line 15), we find (line 16) and remove (lines 18, 19) the less frequent subtrie that is not in use according to a time threshold \(TT\), and then load the new subtrie (lines 20, 21). The function \(\text{refreshLastUsedTime}(\text{Trie subtrie})\) sets the last used time of the corresponding subtrie to the current requested time, the function \(\text{refreshFrequency}(\text{Trie subtrie})\) increases the frequency of the specific subtrie by one unit, and the function \(\text{getWorst}(TT)\) finds the less frequent subtrie that is not in use according to the time threshold \(TT\). Finally, in case the dynamic cache is full and there is not a subtrie that is not in use (line 23), the system cannot serve instantly the user and thus the user has to submit the query.

The problem of deciding how to partition the available memory in a static and a dynamic part depends on the characteristics of the expected workload and the intended behavior of the system. In general we could say that it is a good choice to have a dynamic part in order to offer fast responses for recent and frequent queries and adapt to emerging temporal trends (e.g. breaking news about an earthquake will cause a significant number of new queries on the subject). In [41, 58, 54], the authors illustrate that the majority of the repeated queries are referenced again within short time intervals, i.e. queries have significant temporal locality. Nevertheless, there remains an important portion of queries that are repeated within relatively longer time intervals. The authors in [36] observed that query popularity follows an inverse power-law distribution, and that the most popular

\(^1\)http://www.excite.com
queries submitted to search engines do not change very frequently. This is a strong rationale for introducing a static part. But which is the better way to partition the memory? In [41], the author mentions that although dynamic caching (most recently accessed queries) can use large caches more effectively, static caching (caching of the most popular queries) can perform better for (very) small caches. Furthermore, the experiments in [26] illustrate that for all the replacement policies, the best way to partition the memory is to give more space to the static part (specifically, giving 60% to 80% of the memory to the static part, we can achieve the highest hit ratio).

At Section 6.3, we experimentally evaluate various caching policies in order to reveal the most appropriate for our case.

Algorithm 1 A hybrid caching mechanism

**Input:** Current query $q$  
**Output:** A subtrie’s node which contains the required data

1: $st \leftarrow \text{subtrie}(q)$  //for the current query $q$, find the corresponding subtrie  
2: $S = \text{StaticMap.size()}$  
3: if $\text{StaticMap.contains}(st)$ then  
4: return $\text{StaticMap.getNode}(st,q)$  
5: else  //the requested subtrie is not in the static part  
6: if $\text{DynMap.contains}(st)$ then  
7: \text{refreshLastUsedTime}(st)  
8: \text{refreshFrequency}(st)  
9: return $\text{DynMap.getNode}(st,q)$  
10: else  //the requested subtrie is not in the dynamic part  
11: if $\text{DynMap.size()} < MC - S$ then  
12: load($st$, frequency=1)  
13: $\text{DynMap.add}(st)$  
14: return $\text{DynMap.getNode}(st,q)$  
15: else  //dynamic cache is full, remove the less frequent subtrie that is not in use  
16: $s \leftarrow \text{DynMap.getWorst}(TT)$  
17: if $s$ then  
18: unload($s$)  
19: $\text{DynMap.remove}(s)$  
20: load($st$, frequency=1)  
21: $\text{DynMap.add}(st)$  
22: return $\text{DynMap.getNode}(st,q)$  
23: else  //the system cannot serve instantly the user, user has to submit a full query  
24: print Memory is full and in use! Please submit the query!  
25: return NULL
Chapter 5

On “Flexible” Recommendations

Even if we use a very large query log, our index may not contain queries that start with a particular user’s input string. In that case, the system cannot suggest any queries. However, our index may contain a query that is very close to what the user has typed. For instance, a query in the index may contain all the words of user’s input in different order, or the user may have typed a word with typo(s).

It would be nice if in such cases, the system could avoid returning an empty list of suggestions. Below at Section 5.1 we discuss how we can tolerate different word orders, at Section 5.2 we elaborate on typo-tolerant search, while at Section 5.3 we detail how we can efficiently exploit both of the above functionalities.

5.1 On Relaxing the Word Order

Consider a user that starts typing avensis toyota. Suppose that the query trie (or subtrie) contains the query toyota avensis but not the query avensis toyota. After having typed avensis, no toyota is suggested however the user types “t” hoping that toyota will be suggested. Suppose that the trie does not contain any completion for avensis t, so the system cannot offer any suggestion.

To tackle this problem the system can also load the suggestions starting from “t”. Moreover, the system can exploit the fact that the user has already typed avensis. For this reason, instead of suggesting the top rating completions for the prefix “t”, it should search the trie for finding (and ranking) all completions that start from “t” and include the word avensis as second or third (and so on) word. This requires traversing the (sub)trie after “t” (which in any case is traversed for ranking the possible query suggestions), and keeping (scoring with a non zero value) only those completions that contain avensis (i.e. performing some string comparisons). This may yield suggestions like toyota avensis, technical characteristics avensis, test drive avensis, etc. If the user does not select any of the above and instead types “i” this process will be repeated, leading to
suggestions like *tires avensis*, etc.

Now consider the same scenario as before, up to the point where the user has typed *avensis t*. Now suppose that the “a”-starting trie of the system contains a completion for *avensis t*, e.g. *avensis test*. The system can behave as we described earlier, however now it has to decide which completion to place higher, *avensis test* or *toyota avensis*. One approach is to consider all suggestions that are collected by the possible word permutations and rank them based on their frequency (we analyze more ranking approaches at Section 5.4). Below we discuss the most common case of two-word queries and then various approaches for offering this functionality in case of queries with more than two words.

### 5.1.1 Two-Word Queries

For two words queries (which is the majority of queries in web search engines) the system can check both orders (at the time the user starts typing the second word). During looking up the permuted query, the system exploits the knowledge of the first word to restrict the possible suggestions. Finally the system ranks the suggestions yielded by each word order and derives a single list of recommended query suggestions. Regarding efficiency, if the two permutations belong to the same trie, then only one more subtrie traversal is needed (in order to collect the suggestions that start with the second word). Furthermore, the system has to perform $|Q_w|$ string comparisons (where $Q_w$ is the set of queries that start with the character sequence $w$), in order to check if the queries that start with the second word (or part of word if user has not finished typing) contain the first word. The cost for checking if a string is contained in an other string has complexity $O(n)$, where $n$ is the length of the biggest string (Boyer-Moore string search algorithm\(^1\)). However, if the two permutations do not belong to the same subtrie (in case of PET and PTIE), the system has to load the new required subtrie, which costs a bit more if the particular subtrie is not in the cache (in case we adopt a caching mechanism).

### 5.1.2 Queries with Many Words

The case of queries with more than two words is more expensive since a $m$-words query has $m!$ possible permutations. However we should note that long queries are not frequent. Nevertheless, a system can adopt one of the following approaches:

(A) **Check all possible $m!$ permutations** (where $m$ is the number of words of user’s input string). This approach is prohibitively expensive (especially for queries with many words), as it requires $m!$ trie traversals.

(B) **Check for queries that start from a word of user’s input string and contain at least one of the remaining words**. The more of the remaining words the query contains, the higher rank it receives. In this approach, a query in the index does not have to contain all the words of user’s input in order to be added in the list

of suggestions, limiting in this way the probability of returning an empty list. For a query with \( m \) words, this approach has to perform \( m \) trie traversals and \( \sum_{i=1}^{m} |Q_i|(m - 1) \) string comparisons, where \( Q_i \) is the set of queries that start with the \( i \)-th word of the query. Algorithm 2 describes this approach. For each word of user’s input string (line 3), we find all the queries of the log that start with this word (line 5). Then for each query (line 6), we check if it contains one or more of input’s words (lines 8, 9). The more of the remaining words the query contains, the higher rank it receives (line 10). Note that this rank is not the final rank of the suggestion. It actually represents the number of words in query that are contained in each suggestion and it will be taken into account in the final ranking (of both normal and different word-order suggestions). Finally, if it contains at least one word (line 11), we add this query to the list of suggestions (line 12).

(C) Check for queries that start with the most frequent (in the query log) words of user’s input string and contain at least one of the remaining words (the more of the remaining words the query contains, the higher rank it receives). In this approach, the system has to keep one more index for each word that represents its frequency in the query log (the index must be in descending frequency order). The main idea is that the system first checks the permutations that have the higher probability, then those with lower and so on. This maximizes the probability of yielding non empty suggestions and allows us to control the required number of (sub)trie traversals since it is not required to check for queries starting with each word of user’s input. Specifically, the system has to perform \( m_f \) trie traversals, where \( m_f \) is the desired number of the most frequent words of user’s input that we want to check (\( m_f < m \)), and \( \sum_{i=1}^{m_f} |Q_{f_i}|(m - 1) \) string comparisons, where \( Q_{f_i} \) is the set of queries that start with the \( i \)-th more frequent word of the query.

---

**Algorithm 2** Finding suggestions without considering the word order

**Input:** Current user’s input \( q \), the query log trie

**Output:** A list of suggestions each accompanied by its score which is actually the number of words in \( q \) that are contained in each suggestion.

1: \( \text{words} \leftarrow \text{getWords}(q) \) //for the current input \( q \), find and return its distinct words
2: \( \text{suggestions} \leftarrow \text{new list}() \)
3: for all \( \text{word} \) in \( \text{words} \) do
4: \( \text{trie.descendTo(word)} \) //trie traversal
5: \( \text{queries} \leftarrow \text{trie.findDescendantQueries()} \) //get queries starting with \( \text{word} \)
6: for all \( \text{query} \) in \( \text{queries} \) do
7: \( \text{score} = 0; \) //initialization of suggestion’s score
8: for all \( \text{cur_word} \) in \( \text{words} - \{\text{word}\} \) do
9: if \( \text{query.contains(cur_word)} \) then
10: \( \text{score}++ \)
11: if \( \text{score} > 0 \) then //the query contains at least one of input’s words
12: \( \text{suggestions.add(query, score)} \)
13: return \( \text{suggestions} \)
CHAPTER 5. ON “FLEXIBLE” RECOMMENDATIONS

<table>
<thead>
<tr>
<th>Cost and Functionality</th>
<th>Approach A</th>
<th>Approach B</th>
<th>Approach C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of (sub)trie traversals</td>
<td>(m)</td>
<td>(m)</td>
<td>(m_f)</td>
</tr>
<tr>
<td>Num of string comparisons</td>
<td>(\emptyset)</td>
<td>(\sum_{i=1}^{m}</td>
<td>Q_i</td>
</tr>
<tr>
<td>Max number of subtrie loadings (for PET and PTIE)</td>
<td>(m)</td>
<td>(m)</td>
<td>(m_f)</td>
</tr>
<tr>
<td>Suggestion must contains all the words of user’s input</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Need of additional index</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.1: Synopsis of the implementation approaches of word-order independent search (\(m\) is the number of words of user’s input, \(m_f\) is the desired number of the most frequent (in the log) words of user’s input that we want to check, \(Q_i\) is the set of queries that start with the \(i\)-th word of the query, and \(Q_{f_i}\) is the set of queries that start with the \(i\)-th most frequent word of the query).

For the first two approaches, and in case we adopt PET or PTIE, the system has to load up to \(m\) distinct subtries (one subtrie for each distinct word of user’s input). For the third approach, the system has to load up to \(m_f\) distinct subtries. Furthermore, the user must type at least \(k\) letters of the last word in order to check also for queries that start with this word (where \(k\) is the number of the first characters that determine trie partitioning).

Table 5.1 summarizes the aforementioned approaches. For queries with few words, the second approach is beneficial, since:

i) it requires few trie traversals (proportional to the number of user’s distinct words),

ii) it limits the probability of returning an empty list of suggestions (a suggestion does not have to contain all the words of user’s input), and

iii) it does not need an additional index.

For queries with many words, it is better to adopt the third approach, which limits the (sub-)trie traversals, the string comparisons and the subtrie loadings (for PET and PTIE), and moreover maximizes the probability of yielding non empty suggestions since we first check for queries that start with the most frequent (in the query log) words. Furthermore, in PET and PTIE, the subtries that contain the most frequent words is more possible to lie in the cache (since the static part of the cache contains the subtries of the most frequent queries).

### 5.1.3 Incremental Suggestions

Note that while the user is typing a word, if the new input is part of the old input, i.e. user has not changed what he has already typed (which is the common case), then the new suggestions are subset of the previous suggestions. For example, if the user has typed “toy” and for this input he gets the suggestions “toyota”, “toyota avensis”, “toyota corolla”, “toys” and “toy story”, then by pressing “o” he gets the
suggestions *toyota*, *toyota avensis* and *toyota corolla* which are a subset of the previous set of suggestions. Thereby, once the system has retrieved a list of suggestions for a particular word and user has not changed what he has already typed, we can avoid trie traversals and consequently subtrie loadings in case of PET and PTIE. This is also important in word-order independent search for queries with more than one word, in particular for the second approach (B). For example, if user has typed *toyota* and for this input he gets the suggestions *toyota*, *toyota avensis* and *toyota corolla* then by continue typing *toyota av*, we can first filter the existing suggestions by removing those that do not contain *av*, and then traverse the trie for getting the new suggestions that start with *av* and contain *toyota*. Once we have retrieved the new suggestions and user continues typing the second word, we just filter the new suggestions according to what user is typing (i.e. there is no need for new trie traversals). Consequently, in this case the costs in Table 5.1 do not represent the costs of one user’s keystroke, but the total costs of the whole query (e.g. user copies and pastes his query). However, if user types his query too fast or changes his initial input (by pressing the backspace button or using the mouse), we cannot exploit the incremental suggestions.

Algorithm 3 describes the incremental algorithm. We first check if the new input starts with the old one (line 2). If so, we filter the last suggestions according to the new input (line 3) and if it contains new words (line 4), we retrieve them (line 5). Otherwise, if the user has changed his previous query (line 6), we get the words of the entire new input (line 7). Afterwards we proceed as in Algorithm 2, i.e. we check for queries that start with each word and contain at least one of the remaining words.

**Algorithm 3** Incrementally find suggestions without considering the word order

**Input:** Current input *q*, previous input *qprev*, last suggestions, the query log trie

**Output:** A list of suggestions each accompanied by its score which is actually the number of words in *q* that are contained in each suggestion.

1: \( \text{words} \leftarrow \text{new list()} \)
2: \( \text{if } q.\text{startsWith}(qprev) \text{ then} \)
3: \( \text{suggestions} \leftarrow \text{filter}(q) \)  //filter the suggestions according to the new input
4: \( \text{if } \text{containsNewWords}(q,qprev) \text{ then} \)
5: \( \text{words} \leftarrow \text{getNewWords}(q,qprev) \)  //return only the new words
6: \( \text{else} \)
7: \( \text{words} \leftarrow \text{getWords}(q) \)  //find and return all the words of *q*
8: \( \text{<<LINES 3-13 OF ALGORITHM 2>>} \)

### 5.1.4 Synopsis

Relaxing the word order of the suggestion system offers two benefits, one for the user’s side and one for the server’s side. Obviously, user’s satisfaction grows since the system returns suggestions that it would not find without this functionality.
On the other hand, the system does not need to index queries that contain exactly the same words, reducing in this way the size of the trie. However, if we cannot exploit the incremental approach, it increases system’s response time since the system has to perform more trie traversals, compute many string comparisons and maybe to load and access other subtries (in case of PET and PTIE). At Section 6.4 we report experimental results that demonstrate the feasibility of this functionality in the proposed index structures.

5.2 Typo-Tolerant Search

In this Section we describe approaches for offering typo-tolerant query suggestions, and the challenges that arise by adopting them in our partitioned trie-based indexes.

5.2.1 Motivation and Related Work

Consider a user typing a query who is not sure for the spelling of a word, e.g. he types merilyn, but actually he would like to type marilyn. Suppose that the query trie (or subtrie) contains the queries marilyn, marilyn monroe and marilyn manson, but not the query merilyn. The user (after having typed the first 2 characters) will never get the suggestion marilyn and consequently the other two suggestions marilyn monroe and marilyn manson.

[21] and [14] study the problem of online spelling correction for query completions and both propose a trie-based approach and capture typos either via Edit distance [14] or a n-gram transformation model [21]. However, instant behavior is more challenging in our partitioned trie-based indexes because we have to load and access many subtries, which is time consuming and may overload the system.

5.2.2 Implementation Approaches

(A) One approach is to load also the suggestions that their beginning substring is “similar” to the query that the user is typing. For example, the system can compute the Edit (Levenshtein) distance between user’s input and the beginning substring of each full query in the log. If this number is lower than a threshold, the system can add the corresponding queries to the list of suggestions and rank them as if no edit distance were employed. Specifically, in each user’s keystroke we can detect the active nodes of the trie. A node is considered active if the edit distance between its corresponding string and user’s input is lower than a threshold. For example, in the trie of Figure 5.1, for user’s input meri and edit distance = 1, the active nodes are the nodes that correspond to the character

\[ \text{http://en.wikipedia.org/wiki/Levenshtein_distance} \]

\[ \text{Note that the complexity of a common edit distance algorithm, e.g. a dynamic programming algorithm, is } O(nm), \text{ where } n \text{ and } m \text{ are the strings length (in our case } m \text{ is the length of user’s query and } n \text{ is the sum of the lengths of all queries in the query log).} \]
sequences ceri and mari. Thereby, the system can suggest the queries cerise, cerium, maria, and marilyn. For choosing the right edit distance threshold, we can take into account the length of user’s current input. For instance, we can allow one edit operation (insertion, deletion or substitution) per three characters, i.e. threshold = input\_length/3. Moreover, we start searching for similar queries when user has typed at least the third character (there is no need to find typos if user has typed too few characters).

(B) Ignoring Typo in the First Character. Note that if we want to find suggestions including typo in the first character, the system has to compute the edit distance between user’s input and the beginning substring of many queries in the log (in order to find the active nodes). Specifically, in the worst case the system has to visit all the nodes that their corresponding string has length equal to the input length plus the edit distance threshold. For example, for the input “meri” and in case we allow one edit operation per three characters the system has to check the nodes with maximum string length equal to five characters (i.e. all the nodes of level ≤ 5). However, this costs a lot and is proportional to the total number of distinct queries that lie in the index. If we would like to handle this efficiently we would have to create a character-based suffix tree of all queries in the log but that would increase the space requirements. Furthermore, in case of PET and PTIE, the system has to load and access all the subtries, overloading in this way the system.

One method to reduce this cost is to find the active nodes only for the part of the trie that contains queries starting with the letter that user’s input starts (e.g. if user has typed merilyn, find all similar queries that start with “m”). For SET and STIE, this requires one more trie traversal of the part of the trie that its
Table 5.2: Synopsis of the implementation approaches of typo-tolerant search ($k$ is the number of the first characters that determine trie partitioning, $n$ is the length of user’s input string, $edt$ is the edit distance threshold for the current input length, $N$ is the set of all nodes of the trie with level $\leq n + edt$, and $N_c$ is the set of all nodes of the trie with level $\leq n + edt$ that their corresponding string starts with input’s first character).

Queries start with a specific character (in order to detect the active nodes), and the computation of the edit distance between the corresponding string of each node and user’s current input. This costs about $|N_c|(\frac{n+edt}{2})n$, where $n$ is the length of user’s input string, $edt$ is the edit distance threshold for the current input length, and $N_c$ is the set of all nodes of the trie with level less than or equal to $n + edt$ that their corresponding string starts with input’s first character. In PET and PTIE and for $k > 1$ (where $k$ is the number of the first characters that determine trie partitioning), the system may need to access more subtries. For example, for input “merilyn” and $k = 2$, the system has to access the subtries that correspond to the character sequences “ma”, “mb”, “mc”, etc. Thus, in this case we must check only the subtries that lie in the cache.

5.2.3 Synopsis

Table 5.2 summarizes the approaches for typo-tolerant search. In comparison to the general approach, ignoring typo in the first character reduces the trie traversals, the edit distance computations and the maximum number of subtrie loadings (for PET and PTIE). Especially for PET and PTIE, we must always ignore typo in the first character and traverse only the subtries that lie in the cache (for $k > 1$). At Section 6.4 we report indicative results regarding this functionality.

5.3 Combining Typos and Different Word Orders

Here we elaborate on how we can support both typo-tolerant and word-order independent search.
5.4. ON RECOMMENDATION RANKING

5.3.1 Motivation
Consider a user typing the query monroe merilyn (notice that the word marilyn was misspelled) and suppose that our index does not contain any query that starts with monroe, however it contains the queries marilyn monroe and marilyn monroe filmography. Unfortunately these queries will not be suggested neither by typo-tolerant nor by word-order independent search. Below we describe two possible approaches.

5.3.2 Implementation Approaches

(A) To tackle this problem the system can detect the active nodes of each word and then restrict the possible suggestions exploiting the knowledge of the remaining words. Specifically, we can use the incremental approach for catching different word orders. If the new input is part of the old input and if the user continues typing the same word (i.e. no space character is detected) then we just update the active nodes of the last word. When we detect that user is typing a new word, we “lock” the active nodes of the previous words (i.e. there is no need to update anymore their active nodes) and try to find the active nodes of only the new word. Furthermore, we filter the suggestions that are derived by each active node to those that contain at least one of the remaining words or one of the corresponding strings of the active nodes of each remaining word (in order to catch possible typos). In this way we can capture both different word orders and typos.

In each keystroke, the computation of the active nodes for a word (or a part of a word if the user has not finished typing it) costs about $|N|(\frac{w+edt}{2})w$, where $w$ is the length of the word, $edt$ is the edit distance threshold for the current word length, and $N$ is the set of all nodes of the trie with level less than or equal to $w + edt$. If the user has changed his initial input (or he starts typing very fast), then we must find the active nodes of each word (which costs $m|N|(\frac{w+edt}{2})w$, where $m$ is the number of words of user’s input). Furthermore, the filtering of the suggestions that derive from each active node to those that contain at least one of the remaining words or one of the corresponding strings of the active nodes of each remaining word, requires $\sum_{i=1}^{l} |A_i|(l - 1)$ string comparisons, where $l$ is the total number of active nodes and $A_i$ is the set of suggestions that derive from the $i$-th active node.

(B) To improve efficiency, we can ignore typo in the first character, as we saw at the second approach of typo-tolerant search (Section 5.2). Obviously, this approach reduces the trie traversals, the edit distance computations and the number of subtrie loadings for PET and PTIE.

5.4 On Recommendation Ranking
The ranking of recommendations is important since only a small number of the possible completions are prompted. One approach is to rank the suggestions that
correspond to different word orders or to similar strings as if they were “normal” suggestions. An alternative approach is to penalize the “approximate” suggestions, i.e. to promote those suggestions that correspond exactly to the user’s input string, and penalize the rest taking into account also their difference to the user’s input string.

As regards the first approach, various ranking methods were introduced and analyzed in [33]. One of the methods that was proved effective (and has a clear probabilistic interpretation) is described next. Let \( q \leadsto q' \) denote that \( q \) is a prefix of \( q' \). Let \( q_u \) denote the query the user has typed. We assign a score to each candidate completion \( q \) (where \( q_u \sim q \)) that reflects the probability that the user will select \( q \) if he has typed \( q_u \). The estimation of the probability is based on the query log, and it is defined as:

\[
    \text{Score}(q) = \frac{\text{DeepFreq}(q)}{\sum_{q_u \sim q} \text{DeepFreq}(q')}
\]

where \( \text{DeepFreq}(q) = \text{freq}(q) + \sum_{q \leadsto q'} \text{freq}(q') \), and \( \text{freq}(q) \) is the frequency of \( q \) in the query log. We have to stress that this ranking does not affect the instant behavior because we precompute the scores of all queries and we store them in the trie (or subtries). So one approach is to use the above formula as it is for both exact and approximate recommendations.

The alternative approach is to penalize the approximate. This can be realized also by a simple mechanism. For instance, suppose that the system policy is to prompt the top-10 suggestions. A simple policy is to show the approximate matches always at the last positions of the 10 suggestions i.e. at the positions 6 to 10. As regards the ranking of the approximate recommendations, one approach is to rely solely on their original score. Alternatively we can reduce their score based on their distance with the user’s input string, e.g. taking into account the number of common words, the edit distance, or combinations of these.

A comparative evaluation of the above ranking choices as well as their personalization are beyond the scope of this work but certainly important topics of our future research agenda. However we can argue that for providing personalized recommendations (e.g. through a collaborative approach), it is necessary to support the index and the algorithms that we present in this thesis. Just indicatively, suppose that we log the queries that each user submits and maintain a bitmap of size \(|\text{Users}| \times |Q|\). One approach to compute personalized recommendations for a user \( u \) is to compute his similar users, say \( Usim \) (using the above matrix and the well known methods), and then define a ranking method for the recommendations that takes into account the queries submitted by the users in \( Usim \). For example, and according to the aforementioned ranking formula, instead of \( \text{freq}(q) \) we could use \( \text{freq}(q, Usim) \) the latter being the frequency of \( q \) in the log entries that correspond to users in \( Usim \) (or a combination of global and “local” frequency).
Chapter 6

Experimental Evaluation

In this chapter we first experimentally evaluate the proposed index structures (Section 6.1). Then we describe criteria that should be used and in the right order for selecting the most appropriate index, and we detail the selection according to several parameters of the precomputed information (Section 6.2). Subsequently we experimentally evaluate various caching policies for understanding how they affect the throughput that can be served and the response time (Section 6.3). Finally, we report indicative results regarding word-order independent and typo-tolerant search, and we quantify the benefits of this flexibility in terms of how many more suggestions users receive (Section 6.4).

6.1 Evaluation of Index Structures

The objective is to identify the more scalable indexing approach(es) and to clarify the main performance trade-offs.

6.1.1 Evaluation Aspects

We evaluated the four approaches (SET, PET, STIE, PTIE) according to the following aspects:

- **Trie Size to be loaded in main memory.** We will measure the size of the trie that has to be loaded in main memory. The key requirement for those approaches which are based on a single trie, is to fit in memory, since loading is done only once (at deployment time), so the loading time of the trie does not affect the performance at user requests. Instead, the loading time is very important for those approaches relying on multiple subtries, since each user request requires loading the appropriate subtrie from the disk.

- **Average retrieval time.** We will measure the average time for retrieving the suggestions, the results and the supplementary information (e.g. the cluster
6. EXPERIMENTAL EVALUATION

<table>
<thead>
<tr>
<th>Num. of log’s queries</th>
<th>Num. of unique queries</th>
<th>Avg num. of words per query</th>
<th>Num. of distinct words</th>
<th>Avg num. of chars per query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>578</td>
<td>2.23</td>
<td>950</td>
<td>15.5</td>
</tr>
<tr>
<td>10,000</td>
<td>5,341</td>
<td>2.3</td>
<td>6,225</td>
<td>16</td>
</tr>
<tr>
<td>20,000</td>
<td>10,518</td>
<td>2.34</td>
<td>10,526</td>
<td>16.2</td>
</tr>
<tr>
<td>40,000</td>
<td>20,184</td>
<td>2.35</td>
<td>17,179</td>
<td>16.2</td>
</tr>
</tbody>
</table>

Table 6.1: The query logs used on experiments.

label tree) for a specific current user’s query. This time does not contain the network’s delay time and the javascript running time.

- **Construction and Update Time.** We will measure the time required to process the query log and construct the corresponding index. This task has to be done once, however it should be redone periodically since the query log changes. We will also investigate the time required to update the index structures (after query log change) instead of constructing them from scratch.

6.1.2 Data Sets and Setup

We used four query logs of different sizes. One with 1,000 queries, one with 10,000 queries, one with 20,000 queries and one with 40,000 queries. Each file is a subset of a random log sample of about 45,000 fully anonymized queries from a real query log (from Excite\(^1\) web search engine). Table 6.1 illustrates some features of these files (the average number of words per query and the average number of characters per query concern all the queries of the log, not only the distinct). We should note that it is not necessary to have a very big set of distinct queries for an IOS functionality. Note that in web search engines query logs the query repetition frequency follows a Zipf distribution [58], i.e. there are few queries that are repeated frequently and a very large number of queries which have very small frequency. Obviously that latter are not very useful to be logged and used as suggestions for an IOS functionality in the sense that they will not be suggested as completions (their rank will be very low). Regarding trie partitioning, we created subtries each corresponding to the first two characters. This partitioning yields subtries whose sizes are very close (Figure 3.2a). Finally, in all experiments, for each query in the log we precompute and store the top-100 results that are returned from Google web search engine and their cluster label tree using the algorithm NM-STC proposed in [35].

All experiments were carried out in an ordinary laptop with processor Intel Core 2 Duo P7370 @ 2.00Ghz CPU, 3GB RAM and running Windows 7 (64 bit). The implementation of all approaches was in Java 1.6 (J2EE platform), using Apache Tomcat 6 and Ajax JSP Tag Library.

\(^1\)http://www.excite.com
6.1. EVALUATION OF INDEX STRUCTURES

6.1.3 Results on Trie’s sizes

Figures 6.1 and 6.2 illustrate how the size of the trie that is loaded in the main memory grows in each approach.

In the SET approach (Figure 6.1a), the size of the trie increases linearly with the query log size and can get quite high. In particular, we observe that SET requires about 40 MB for storing the results and the clusters of a query log of 1000 queries (578 distinct queries), i.e. it needs about 70 KB per query.

In the PET approach (Figure 6.1b), we examine how the size of a subtrie grows as a function of the minimum number of entries we decide to store in it (in our setting an entry consists of the top-100 results and their cluster label tree of a query). Since the subtries do not have the same size, the diagram presents the worst case (max size), the best case (min size) and the mean case (average size) of a subtrie’s size (for a query log of 40,000 queries). The smaller this number is, the more subtries (of smaller size) have to be created. For example, in our setting selecting to store at least 50 entries in each subtrie, 170 tries have to be created. On the other hand, selecting to store as least 800 entries per subtrie, only 24 subtries are created of much bigger size. However, note that we cannot select a very small number (especially if we use a very large query log) because the sizes of the subtries that will be created will have big differences, i.e. some subtries will contain too many queries while other will contain much less. Specifically this depends on the number of distinct substrings of size \(k\), where \(k\) is the number of the first characters that determine trie partitioning (for more see [33]).

In the STIE approach (Figure 6.2a), the number of entries we select to store in each separate random access file does not affect the size of the trie that is loaded in the main memory. We observe that this approach leads to a very small trie’s size. In particular, STIE requires about 1.8 MB for a query log of 1,000 queries (578 distinct queries), i.e. it needs only about 3 KB per query. For a query log of 40,000 queries, selecting to store up to 50 entries in each file leads to the creation of 404 external files (with average size of about 3.5 MB each one). At the same
time, selecting to store up to 400 entries in each file, 51 files have to be created (with average size of about 28 MB each one). However, as mentioned above, the maximum number of entries we select to store in each separate file does not affect the size of the trie and thus our experiments.

In the PTIE approach (Figure 6.2b), we combine trie partitioning and trie with indexes to external random access files in order to further reduce the size of the trie. As in PET, we examine how the minimum number of entries we select to store in each subtrie affects its size. Since the subtries have not the same size, the diagram presents the worst case (max size), the best case (min size) and the mean case (average size) of a subtrie’s size (for a query log of 40,000 queries and selecting to store up to 400 entries in each external file). We observe that even for the worst case, the size of a subtrie is extremely low.

**Conclusion.** Figure 6.3 compares the four approaches where the y-axis is in logarithmic scale. For the PET and PTIE approach, we choose to depict the best case (i.e. we store at least 50 entries in each subtrie), so the average size of a subtrie is almost constant and independent of the query log size. As expected, SET requires the more main memory space, which can be very big for large query

![Figure 6.2: Size of STIE and PTIE](a) STIE  
(b) PTIE

![Figure 6.3: Comparison of the trie’s size.](c)
logs. The best approach (regarding only the size of the trie) is PTIE with great
difference from the others. STIE follows but for query logs of less than 40,000
queries. Finally, PET requires less space than STIE only for very large query logs
(of more than 40,000 queries).

6.1.4 Average Retrieval Time

Figures 6.4 and 6.5 depict the average retrieval time for each implementation
approach (for the PET approach, the corresponding diagram concerns a query log
of 40,000 queries).

We can see that SET has very low average retrieval time (almost constant),
taking only a few milliseconds to retrieve the required data even for very large query
logs (Figure 6.4a). However, it requires that the entire trie fits in main memory.
In PET, the average retrieval time highly depends on the size of the appropriate
subtrie that needs to be loaded in the main memory, i.e. the minimum number
of entries per subtrie. We observe that for all cases, PET is much slower than
SET (Figure 6.4b). The STIE approach was implemented using random access
files and therefore the average retrieval time does not depend on the size of the
external file. We observe that this approach is very efficient even for large query
logs with average retrieval time lower than 40 ms (Figure 6.5a). Finally, as in STIE
approach, PTIE was implemented using random access files. For this reason, its
average retrieval time depends mainly on the size of the query log (Figure 6.5b).
Note that contrary to PET, the subtries of PTIE have very small size (since as we
saw they store only three numbers and not the entire information) and thus the
minimum number of entries we select to store in each subtrie slightly affects its
average retrieval time. We observe that this approach is a bit worse than STIE.
The reason is that it requires one more disk access in order to find and load the
appropriate subtrie.

Conclusion. Figure 6.6b compares the average retrieval time of all approaches
(for PET we consider the best case of at least 50 entries per subtrie). It is obvious
that the SET approach is much more efficient than all the other approaches. However, as mentioned above, for large query logs its size is huge and it does not fit in main memory. For this reason, one may argue that the best approach is STIE, as it combines low trie size and very fast retrieval time. Moreover, PTIE is a very good approach as it offers small subtrie sizes and low retrieval time, thus it can be used even for very large query logs. Finally, PET approach is the worst although its retrieval time is not unacceptable (about 200 ms). Figure 6.6a compares only the SET, STIE and PTIE approaches for various number of queries in the log, excluding PET (as its efficiency highly depends on the minimum number of entries we want to store in each subtrie). We observe that in all approaches, the average retrieval time is very low. SET is the more efficient approach, followed by STIE, and finally by PTIE.
6.1.5 Large Scale Evaluation of PTIE

We conducted experiments of PTIE (which according to the previous results is the more scalable approach) using a very large query log and large amounts of precomputed information. In particular, we ran IOS over a meta-search engine that offers clustering of the results. We used a synthetic query log of one million distinct queries and synthetic precomputed information for each query (the synthetic log retains the main features of a real query log; the average number of words per query is 2.3 and the average length is 15 characters). Regarding trie partitioning, we created subtries each corresponding to the first two characters. Moreover, we chose to store at least 1,000 entries in each subtrie and 1,000 entries in each random access file. PTIE created 344 subtries of 615 MB and 992 random access files of about 1 terabyte.

We measured the average time for retrieving the suggestions, the results and the clusters for a random input string without using any cache (this time does not contain the network’s delay time and the javascript running time). To this end we sequentially selected 10,000 random queries from the query log. For each random query, we kept only its first three characters, we found the subtrie that corresponds to this character sequence, we loaded it, traversed it and found all the suggestions. For the top suggestion, we accessed the corresponding random access file and retrieved its first page of results and its cluster label tree. The average retrieval time was about 135 ms, proving that PTIE is very efficient even for very large query logs and precomputed information.

At last we should mention that with a modest personal computer the other indexing approaches cannot be used in such a big number of queries and precomputed information. PTIE is very scalable because the data that it has to load at request time has small size, its total cost of main memory is low and depends on the number of requests, and furthermore it has low response time.

6.1.6 Construction and Update Time

Table 6.2 reports the average times of the main construction tasks for various sizes of the query log. In these experiments, we chose to retrieve the top-100 results from the underlying web search engine and to offer clustering of these results (using the clustering algorithm NM-STC proposed in [35]).

The results are almost the same for all index approaches and for this reason we present only the results of the SET approach (for STIE and PTIE, the creation of the external files has very small time impact, while for PET and PTIE, the time for creating all the subtries is almost equal to the time SET requires to create a single big trie). We observe that the task requiring the most time is the retrieval of the results and their clustering. For example, for the query log of 1,000 queries (578 distinct queries), we can see that the retrieval of the results and the construction of the cluster label tree takes about 592 seconds, i.e. around 1 second per query. However, one can easily see that this time depends on the particular algorithms
that are employed (for the query evaluation and the clustering of the results).

**Incremental Update.** In the incremental approach, the first step is the creation of the new trie. As we saw, the time for creating a trie depends on the size of the query log (about 1 second for each query in our setting). Thus, if the new query log has 250 distinct queries then its trie creation time is about 4 minutes. For testing the time required for merging two tries, we run an experiment in SET with an initial trie of about 11,000 distinct queries (767 MB) and a new trie of 250 distinct queries (14 MB). The execution time was about 2 minutes. We can see that the incremental approach is much more efficient (6 minutes versus about 3 hours).

Nevertheless, if in each query submission we store the results and the supplementary information, then the update of the trie can be performed much faster.

### 6.2 Selecting the Right Index

The main conclusion is that the proposed functionality is feasible for real time interaction even if the query log and the preprocessed information have considerably high size. The selection of the implementation approach depends on the available main memory, the size of the log, and the size of the preprocessed information.

Below we describe criteria that should be used and in the right order.

1. If the entire SET fits in memory then this is the faster choice since no loading has to be done during user’s requests.

2. If SET does not fit in memory then, the next choice to follow is STIE since it offers faster retrieval time in comparison to PET and PTIE. However note that STIE is feasible only if the trie of the query log fits in main memory (which is usually the case); if not then PTIE approach has to be used.

3. Finally, we could say that the more scalable approach is PTIE, since it can be adopted even if the available main memory has very small size. Furthermore, the experiments showed that PTIE is very efficient (with low retrieval time) and can be used even with very large query logs. Figure 6.7 analyzes the retrieval time of PTIE’s main tasks. We notice that the most consuming
6.2. SELECTING THE RIGHT INDEX

The task is the loading of the subtrie, since it has to load it to main memory at request time. This also limits the throughput that is feasible to achieve. This problem can be alleviated by adopting a caching scheme, as we saw at Chapter 4.

Figure 6.8 summarizes the results and reveals the trade-off between (a) response time, (b) amount of bytes to be loaded in main memory for serving one request, and (c) cost (amount) of main memory that should be available, for the four indexes.
6. EXPERIMENTAL EVALUATION

Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Estimated Size (bytes)</th>
<th>In our setting (bytes)</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
</table>
| Results      | $RSize \approx N \times (results\_plain\_sz + results\_html\_sz)$ | $results\_plain\_sz \approx 150$  
              |                                 | $results\_html\_sz \approx 450$ | $N$                          | Num of top results            |
| Cluster Label Tree | $CSize \approx C \times (cluster\_plain\_sz + cluster\_html\_sz)$ | $cluster\_plain\_sz \approx 50$       
              |                                 | $cluster\_html\_sz \approx 200$   | $C$                          | Num of presented clusters     |
| Overall      | $OSize \approx RSize + CSize$   | $OSize \approx 600N + 250C$ | $results\_plain\_sz$         | Result’s plain text size      |
|              |                                 |                         | $results\_html\_sz$           | Result’s html data size       |
|              |                                 |                         | $cluster\_plain\_sz$          | Cluster’s plain text size     |
|              |                                 |                         | $cluster\_html\_sz$           | Cluster’s html data size      |

(a) Estimated size of the trie’s required data for 1 query.

Table 6.3: Estimated size of the trie’s required data.

of main memory contrary to the other three approaches that require much less.

6.2.1 Autonomic Selection of the Right Index

The knowledge of the values of some parameters could assist us in selecting (manually or even automatically) the more appropriate indexing approach according to the available main memory.

The size of the results for one query is about:

$$RSize \approx N \times (result\_plain\_sz + result\_html\_sz)$$

bytes, where $N$ is the number of the top results that we want to retrieve for a query. The $result\_plain\_sz$ represents the size of the plain text of a result, i.e. the text of the title, the snippet and the URL of a result and $result\_html\_sz$ represents the size of the HTML data that are needed in order to present properly a result in the browser. In our setting, $result\_plain\_sz \approx 150$ bytes and $result\_html\_sz \approx 450$ bytes, thus we could say that the results of a query require about $600 \times N$ bytes.

In case we want to offer clustering of the top results, the size of the cluster label tree is about:

$$CSize \approx C \times (cluster\_plain\_sz + cluster\_html\_sz)$$

bytes, where $C$ is the desired number of clusters. The $cluster\_plain\_sz$ represents the size of a cluster’s text and the $cluster\_html\_sz$ represents the size of the HTML data that a cluster requires, e.g. the javascript action when someone clicks on a specific cluster, the results in which the particular cluster appears, etc. In our setting, $cluster\_plain\_sz \approx 50$ bytes and $cluster\_html\_sz \approx 200$ bytes, thus we could say that the cluster label tree of a query requires about $250 \times C$ bytes.

The overall required size for one query is: $OSize \approx RSize + CSize$, i.e. approximately: $OSize \approx 600 \times N + 250 \times C$ bytes. Table 6.3 summarizes all the above.
6.3. EVALUATION OF CACHING SCHEMES

Therefore, the size of SET for a query log of \( Q \) distinct queries is about: \( S_{SET} \approx Q \times (\text{avg}_{q_{sz}} + O_{Size}) \), where \( \text{avg}_{q_{sz}} \) is the average size of the query string (about 10 bytes). In STIE, the results’ overview data are not stored in the trie and thus its size is: \( S_{STIE} \approx Q \times (\text{avg}_{q_{sz}} + \text{pointers}_{sz}) \), where \( \text{pointers}_{sz} \) is the size that the three pointers (file number, bytes to skip, bytes to read) require (3 bytes), i.e. STIE occupies about \( 13 \times Q \) bytes.

It is not hard to see that one can reduce the size of the required data, and thus the size of the trie, by:

- Removing the HTML data (in that case \( \text{result\_html\_sz} = 0, \text{cluster\_html\_sz} = 0 \)).
- Storing less top results for each query (i.e. adopting a smaller \( N \)).
- Storing less cluster labels (i.e. adopting a smaller \( C \)).
- Using smaller query log (i.e. adopting a smaller \( Q \)).
- Combining some of the above.

All the above can help us to decide which indexing approach to choose according to the available main memory capacity (\( MC \)) or to switch between different strategies while data is growing. In general, a simple algorithm could be:

\[
\text{if } S_{SET} < MC \text{ then, use SET,}
\text{else if } S_{STIE} < MC \text{ then, use STIE,}
\text{else use PTIE.}
\]

6.3 Evaluation of Caching Schemes

To evaluate the benefits of caching in our setting, we performed a comparative evaluation of the following choices:

1. Static cache
2. Dynamic cache
3. Hybrid (static: 30%, dynamic: 70%)
4. Hybrid (static: 50%, dynamic: 50%)
5. Hybrid (static: 70%, dynamic: 30%)
6. No cache
6.3.1 Served Queries

Figure 6.9 reports a) the percentage of queries that were served from the cache without delay (fast response since the request can be served instantly from the cache), b) the percentage of queries that were served from the cache with delay (in order to remove the less frequent subtrie and load the requested one), and c) the percentage of queries that could not be served (because of memory overloading). We see how the results change starting from a full dynamic cache and resulting in a full static cache. We observe that as the cache becomes more static, more queries are served fast and less are served with delay. However, in a full static cache, almost half of the queries cannot be served. On the contrary, this percentage is very low in the other policies (lower than 5%).

6.3.2 Average Response Time

Figure 6.10 illustrates the average response times for all approaches. We notice that as the cache becomes more static, the average response time gets faster, since more queries can be served instantly from the static cache. On the other hand, in a full static cache there is no space in order to serve the less frequent

For the experiments, we used a synthetic query log of **one million distinct** queries. We created 344 subtries of 615 MB using the PTIE approach with trie partitioning based on the first two characters and with minimum 1,000 entries per subtrie. In a loop without any “sleep” period, we run 10,000 random queries, selected from the query log. The query rate was about 8 queries per second. We chose to set the memory capacity ($MC$) to 60 subtries, i.e. 17.44% of all subtries can fit in main memory at the same time. The time threshold ($TT$) that a subtrie is considered in use (and thus it cannot be removed) was set to 10 seconds, i.e. about $10 \times 8 = 80$ queries have to be served at the same time. As we will see later, the values of these three factors (query rate, percentage of cached subtries and time threshold) highly affect the experimental results. As far as the static cache is concerned, we load the more frequent subtries after a query log analysis.
6.3. EVALUATION OF CACHING SCHEMES

Figure 6.10: Average response time of various memory partition policies.

queries and as a result the system gets overloaded and many requests cannot be served. In case we do not use any cache policy, i.e. the system loads the requested subtrie in each user’s request (keystroke) and removes it when the user has been served, the average response time is slightly higher than that of a dynamic cache. However, if many users request a subtrie at the same time, the system can easily get overloaded. Specifically, in our experiments the main memory capacity is set to 60 subtries, i.e. the system can serve at most 60 queries at the same time. Thus, since we have 80 queries that have to be served at the same time period, 20 of them will not be served (25%).

Discussion. We should clarify that the above results concern the case where each user (i.e. each iteration in the loop) requests only one subtrie, in particular the subtrie that contains the randomly selected query. The results do not take into account the common case in which the user continues typing a query and he is served instantly by the same subtrie. For this reason in realistic workloads the results are expected to be better.

6.3.3 Factors that affect the Experimental Results

As we mentioned before, beyond the aforementioned experiments, results and conclusions, we can say that there are three main factors that affect the experimental results:

1. Number of queries per second (Query Rate).

2. Time Threshold (TT), the time (seconds) that a subtrie is considered in use and thus it cannot be removed from the cache.

3. Percentage of cached subtrees (MC / Total number of subtries).

To understand how these factors impact the behavior of the system, we conducted additional experiments assuming a hybrid caching policy with 70% static and 30% dynamic cache. Figure 6.11 illustrates how each of the above factors affects the results keeping constant the other two factors. We observe that the smaller the query rate is, the less requests cannot be served (Figure 6.11a). As far
Figure 6.11: (a) How the number of queries per second affects the results, with 10 seconds time threshold and 17% cached queries. (b) How the time threshold affects the results, with 8 queries per second and 17% cached results. (c) How the percentage of cached queries affects the results, with 10 seconds time threshold and 8 queries per second.

as the time threshold is concerned, we can see that less threshold time results in less requests that cannot be served (Figure 6.11b). Finally, for bigger percentage of cached queries we expect better results. i.e. more fast responses and less requests that cannot be served (Figure 6.11c).

6.3.4 Synopsis

Concluding the above results, we could stress that: (i) The best caching policy is the hybrid with 70% static and 30% dynamic cache, as it combines low percentage of requests that cannot be served, high percentage of requests that are served fast and low average response time. In particular, regarding the settings of the aforementioned experiment, we found out that the particular caching scheme offered about 80% better throughput (since less than 5% of the queries could not be served, contrary to the 25% of no-cache case), and about 25% speedup for queries that lie in the index. (ii) The more percentage of cached subtries we have (that means more main memory capacity), the better results we expect. (iii) We cannot know in advance the number of queries per second, however we expect that for small numbers, the requests that cannot be served are few. (iv) The time threshold that a subtrie is considered in use is constant and thus we know in advance how this number affects the behavior of our system.

To sum up we saw that a caching scheme can significantly increase the throughput and alleviate overloading problems. By partitioning the cache into a static and a dynamic part, and giving more space to the static part, we can increase the throughput that can be served and speed up 10S by offering lower response times.
6.4 Evaluation of “Flexible” Recommendations

6.4.1 Retrieval Time

We decided to measure the average retrieval type of STIE and PTIE (since they are the most scalable). For STIE we used a synthetic query log of 200,000 distinct queries and for PTIE one of 1 million distinct queries. These logs retain the main features of a real query log: the average number of words per query is 2.3 and the average length is 15 characters. We run 1,000 random queries from the log of various number of words (for evaluating the word-order independent search) and various number of characters (for evaluating the typo-tolerant search). Furthermore, we consider the worst case in which we cannot retrieve incremental suggestions and we do not use any caching scheme (for PTIE).

Word-order independent suggestions. Regarding word-order independent suggestions, we run the experiment for 2-word, 4-word, 8-word and 12-word queries and we measure the time for retrieving suggestions that start from a word and contain at least one of the remaining words (approach B of word-order independent search). For each query we keep only the first two characters from the last word. Table 6.4a reports the results. We notice that STIE can efficiently support this functionality with average retrieval time lower than 60 ms, even for queries with many words. PTIE is efficient for 2-word queries with average retrieval time about 180 ms, but for queries with many words the time is very high. Consequently, for queries with more than two words and if the system cannot offer incremental suggestions, PTIE must search only the subtrees that lie in the cache.

Typo-tolerant suggestions. As regards typo-tolerant suggestions, we run the experiment keeping from each query the first-4, first-8, first-12 and first-16 characters. Table 6.4b reports the results of STIE for the common case (approach A of typo-tolerant search), and when ignoring typo in the first character (approach B). We notice that STIE can efficiently support the approach A if the query does not have too many characters (about less than 15 chars), and very efficiently when ignoring typo in the first character (with average retrieval time lower than 40
ms). Since we do not use any caching scheme, we do not report results of PTIE, because PTIE has to load a lot of subtries which is very time consuming. For this reason, PTIE must offer this functionality only by ignoring typo in the first character. Furthermore, if the trie has not been partitioned based on the first character ($k > 1$), we must check only the subtries that lie in the cache.

**Tolerating both typos and different word orders.** In this case we run the experiment for 2-word, 4-word, 8-word and 12-word queries and we measure the time for a) detecting the active nodes of each word ignoring typo in the first character (approach B), b) retrieving the suggestions of each active node, and c) filtering the suggestions exploiting the knowledge of the remaining (active) words. The results revealed that STIE can efficiently support the combination of both functionalities for queries with no more than 4 words (which is the common case). Furthermore, PTIE must offer this functionality incrementally or by checking only the subtries that lie in the cache.

### 6.4.2 Number of Additional Suggestions

In order to quantify the effect of the offered flexibility, we measure the average number of additional suggestions we get when we tolerate different word orders and spelling errors. Specifically, using a real query log of Excite search engine with 22,251 distinct queries, we performed the experiments described below.

**Word-order independent suggestions.** For each distinct query in the log that has more than two words and each word contains at least four characters, we keep only the first two characters of the last word and find its suggestions (initially with the same word order and then without considering the order). For example, for the query *marilyn monroe* we find all queries that start with *marilyn mo* and then the suggestions of the same query without considering the word order (approach B of word-order independent search). We do the same for all queries in the log and we compute the average number of the additional suggestions we get when we tolerate different word orders (number of suggestions ignoring word order minus number of suggestions without ignoring word order). We also run the same experiment keeping the first three and the first four characters of the last word, and using 3-word and 4-word queries.

Table 6.5 shows the results. Notice that even using a small query log (of only 22,251 distinct queries), the number of suggestions always increases. For queries with more than two words, even if we have typed many characters of the last word, the additional suggestions are significantly more.

**Typo-tolerant suggestions.** For each distinct query of the log that has at least four characters, we keep only the first four characters and compute its suggestions. For example, for the query *marilyn monroe*, we find all queries that start with *mari*. Afterwards, we find the active nodes of that query (approach A of typo-tolerant search with edit distance threshold equal to one) and get their corresponding suggestions. Following the same procedure for all distinct queries in the log, we find the average number of the additional suggestions we get when
6.4. EVALUATION OF “FLEXIBLE” RECOMMENDATIONS

<table>
<thead>
<tr>
<th>Query length</th>
<th>Having typed the first 2 chars of the last word</th>
<th>Having typed the first 3 chars of the last word</th>
<th>Having typed the first 4 chars of the last word</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-word queries</td>
<td>1.6</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>3-word queries</td>
<td>10.1</td>
<td>4.7</td>
<td>4</td>
</tr>
<tr>
<td>4-word queries</td>
<td>20.9</td>
<td>13.2</td>
<td>12.3</td>
</tr>
</tbody>
</table>

Table 6.5: Average number of additional suggestions per query when tolerating different word orders.

<table>
<thead>
<tr>
<th>Implementation Approach</th>
<th>Having typed the first 4 chars (edit distance=1)</th>
<th>Having typed the first 8 chars (edit distance=2)</th>
<th>Having typed the first 12 chars (edit distance=4)</th>
<th>Having typed the first 16 chars (edit distance=5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach A (detect the active nodes)</td>
<td>71.4</td>
<td>7.3</td>
<td>7.1</td>
<td>3.3</td>
</tr>
<tr>
<td>Approach B (detect the active nodes, ignoring typo in the 1st char)</td>
<td>48.6</td>
<td>6</td>
<td>5.7</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table 6.6: Average number of additional suggestions per query when tolerating spelling errors.

we tolerate spelling errors. We run the same experiment keeping the first 8, first 12, and first 16 characters, and allowing one edit operation per three characters. Table 6.6 (first row) reports the average number of additional suggestions in each case. Notice that we get much more suggestions, especially if we have not typed many characters. We run the same experiments ignoring typo in the first character (approach B of typo-tolerant search), e.g. for the query marilyn we found the similar queries that start with “m”. Table 6.6 (second row) reports the results. Notice that also here the suggestions are much more, and the difference with the first approach is small. This is a strong rationale for ignoring typo in the first character.

6.4.3 Conclusion

Undoubtedly, word-order independent and typo-tolerant search increase flexibility (and thus user’s satisfaction), and the degree of exploitation of the precomputed information. Furthermore, they increase the number of queries for which the system can assist the corresponding decision making process. However, both functionalities increase system’s response time since the system has to perform more trie traversals, several string comparisons and maybe (in case of PET and PTIE) to load and access other subtrees. Table 6.7 illustrates the applicability of word-order independent search, typo-tolerant search and their combination, over STIE and
Table 6.7: Efficient support of word-order independent search, typo-tolerant search and their combination over the proposed index structures (\(k\) is the number of the first characters that determine trie partitioning, \(m\) is the number of words of user’s input, and \(m_f\) is the desired number of the most frequent (in the log) words of user’s input that we want to check).

PTIE (since these are the most scalable indexes). STIE can efficiently support word order independent and typo-tolerant search since the whole trie is in main memory and both trie traversals and string comparisons do not cost a lot. Only in case of large input strings, we must ignore typo in the first character in order to efficiently tolerate typos. On the other hand, PTIE can efficiently support word-order independent search either incrementally or for the subtries that lie in the cache, and typo-tolerant search by always ignoring typo in the first character. Finally, for offering the combination of both functionalities, we must always ignore typo in the first character.

Also note that these functionalities are worth offering only when there are few or no suggestions of the current input string (meaning that system’s response time is not affected if there are many suggestions, since there is no need to search for typos or other word orders). Furthermore, we can achieve better response time if we stop retrieving suggestions when we reach a desired number. Thereby, the more initial suggestions the system retrieves, the less the system’s response time is affected in order to collect suggestions with typos and different word orders. This technique is beneficial specifically to PTIE, since it may need to load and access many subtries.
Chapter 7

Server’s Benefits

The objective of this Chapter is to justify our claim that IOS is beneficial also for the server side.

Less incoming queries, reduced computational cost per received query

Apart from the benefit for the user’s side, our approach is beneficial also for the server. In particular, we could point that an IOS functionality (a) reduces the number of incoming queries which are not really useful for the end users, since it assists them in avoiding wrongly typed queries (user adapts his query on-the-fly until the results match what he wants), and (b) reduces the computational cost because the same precomputed information is exploited in several requests and thus the engine has to evaluate less tasks at run time. The combination of the above increases the throughput of the server, since the number of incoming queries is much smaller and the response time is much less. This is true even if the user types the right query (for his information need) from the beginning. For instance, and for the case of clustering, in IOS for serving one request we have to load (if not already loaded in memory) and show (send via HTTP) the cluster label tree. Without IOS (e.g. in plain StellaSearch [35]), for serving one request we have to evaluate the query (either locally or send it to a remote engine), fetch the top-L snippets and apply the clustering algorithm NM-STC to derive the cluster label tree (and finally send the outcome over HTTP to the end user). Just indicatively, recall that in our experimental setting, the average time required for IOS is a few milliseconds (lower than 100), while in StellaSearch (which to the best of our knowledge uses the fastest real-time results clustering algorithm) is about 1 second, i.e. ten times slower. The case is analogous for the other kinds of precomputed information (e.g. metadata-based groupings, entities discovered in the search results, etc). In general we can say that the more expensive the “overview information” is to derive, the more beneficial for the server side our approach and indexes are. Notice that above we used the average time as a means to quantify the average computational cost of the server for one request.

In the context of real time results clustering, we could also say that the indexes
of IOS can be considered as a form of cache for speeding up StellaSearch. The trie over the query strings is actually the data structure allowing fast lookups in that cache. Under this perspective, we can say that in our work we exploit that cache not only after the user query has been submitted (i.e. after the user has pressed the submit button), but while the user types his query letter by letter, i.e. during typing. Also note that the size of that cache is large, therefore one part of it is in the main memory, while the rest is in the secondary memory.

**Less monetary cost (at meta search level)**

In a meta-search setting, the engine has to connect to the underlying search engines in order to get the results of a query, and such services are not for free (sometimes they are billed according to the number of served queries). An IOS approach reduces the queries sent to the underlying engines and thus can save money.

**Less network connections (at meta search level)**

If the construction of the “overview of the results” requires connecting to several external sources (e.g. web pages, LOD SPARQL endpoints, etc), then an IOS approach reduces the number of connections which are required.

And finally, we should note that all these services are instant (but this a actually a benefit for the user’s side).

**Expected Speedup**

The speedup that we expect is proportional to the percentage of the incoming queries that can be served from our index. For example, if for every 1,000 incoming queries, about 200 are served from IOS (i.e. they are not submitted to the server), then the 20% of incoming queries are served instantly in time less than 100ms and the 80% are served with delay about 1 second. However, the more queries we precompute the bigger the speedup is. The number of the precomputed queries depends on the system’s available main and secondary memory. In particular, the only real price to pay is actually the space required for storing the precomputed information.

In a typical standalone web search engine, if $V$ is the vocabulary of the dataset, then the size of the inverted file is $O(|V|^2)$, i.e. the square of dataset’s distinct words (Heap’s Law [3]). Considering that about 72% of distinct queries has up to 2 words [51], then the maximum number of two-word queries is $|V|^2$, i.e. in the magnitude of the size of the inverted file. Thus, with a disk space proportional to $O(|V|^2)$, we can have precomputed information for the 72% of the distinct queries. Moreover, note that contrary to the inverted file whose size grows according to the size of the collection, our index is affected only by the size of the query log and in particular by the number of distinct queries.
The required main memory depends on the caching mechanism and the index that we use. For instance, in PTIE, a subtrie of 1,000 queries has size about 1 MB, i.e. a machine with 1 GB available main memory can keep in memory at the same time about 1,000 subtries of 1,000,000 total distinct queries. Thus, with 100 MB (10%) more main memory, we can have instant search for 100,000 more incoming queries (10%).
Chapter 8

Applications of IOS

In this chapter we describe in brief possible forms of IOS for results clustering, metadata-based groupings, entity mining, Linked Data, and vertical search.

8.1 IOS and Results Clustering

[35] describes a MSE (meta-search engine) over Google that clusters at real time the retrieved results according to the words that appear in the title and the snippet of each result. Such a system could precompute and store the cluster label tree and the top-\(k\) results for each query of the log. In this way the system can provide an overview of the results allowing the user to adapt his query while he is typing. For instance, consider a user who wants to find information about the biography of Marilyn Monroe and for this reason he starts typing the query marilyn monroe biography letter by letter. After having typed the first two letters, i.e. the string ma, a set of query completions are prompted as shown in Figure 8.1a. At the same time, the first page of results and the cluster label tree that correspond to the top suggestion macintosh appear instantly (at the main area and the right bar of the screen respectively). At that point the user can either continue typing or select a suggestion from the list. If he selects one of the suggestions, the first page and the clustering of the results that correspond to the selected suggestion appear immediately. Moreover, the user is able to click on a cluster’s name and get instantly the results of only that cluster. Since there is the suggestion marilyn monroe that matches what he would type, the user selects it (Figure 8.1b) and then the cluster label tree and the first page of the query marilyn monroe appear immediately. We observe that the cluster label tree contains a cluster with label biography (3). The user clicks on that cluster and gets instantly the results that are relevant to his information need. Notice that the user typed only 2 letters and made only 2 clicks in total. Furthermore, he has seen how the results about Marilyn Monroe are clustered.

We should also note that in the context of a MSE the delivered results depend on the results of the selected underlying search engines. Since the results of the
same query may be different at different points in time, the precomputed and stored cluster label tree of the query results may be different from the cluster label tree of the current query results. This means that a particular stored label of a cluster may not exist in the cluster label tree of a future search of the same query. Therefore, when a user clicks on a cluster’s label, the system may not be able to perform the query and then focus on the selected cluster unless the results in which this cluster appears have been also stored beforehand. Of course that approach increases the amount of precomputed information that has to be fetched and stored. As we saw, the proposed index structures can be used to provide real time interaction also in such cases.

8.2 IOS and Metadata-based Groupings

In the context of a standalone web search engine (WSE), the engine can precompute and store not only the cluster label tree, but also the facets (metadata-based groupings [43]) of the top-k results of each logged query. For example, in [44] the engine categorizes the top-k results according to five facets: by clustering, by domain, by date, by filetype and by language. The provision of such information during typing can accelerate the search process and the performance of multifaceted browsing interfaces [46, 19, 8, 32]. For instance, consider a user seeking for java tutorials. While he types this query letter by letter, he notices that there is the suggestion java tutorials that satisfies his search need (Figure 8.2). Thus he selects it and instantly the first page of results and the facets that correspond to this query appear. User is now able to narrow the search space by clicking one or more facets. For example, he can choose only presentations (by filetype) of 2011 (by date) that are written in Greek (by language).
8.3 IOS and Entity Mining

Named Entity Recognition (NER) tries to locate and classify atomic elements in text into predefined categories such as the names of persons, organizations, locations, etc. An entity search engine locates and presents to the user entities that lie in the search results [22, 18, 17]. Figure 8.3 depicts a prototype meta-search engine that we have developed [22], which offers entity mining of the top hits. This system retrieves the top-K hits of a WSE, mines the content of each result (either the snippet or the full content) and presents to the user a categorized list with all the discovered entities. When the user clicks on an entity, the results of the specific entity are loaded instantly. Note that this computation costs a lot both in time and in main memory (especially if we select to mine the full content of the results). By exploiting IOS we precompute and store the entities that lie in the search results of the most frequent queries and offer them at real time. For example, in Figure 8.3 we actually see what the user is getting after having selected the suggestion tim berners-lee. We observe entities like Robert Cailliau, under the category “Person”, and MIT under the category “Organization”. By clicking one of the presented entities, e.g. the entity Robert Cailliau (Figure 8.4), user can instantly locate results related to that entity that it would be difficult to find because these results may not lie in the first or the second page (in practice users tend to look only at the first page of results).
Figure 8.3: IOS and entity mining.

Figure 8.4: IOS and entity mining - Limiting the search space.
8.4 IOS and Linked Open Data

IOS can also be useful for Linked Open Data (LOD). In general, LOD refers to data published on the web which are machine-readable, their meaning is explicitly defined, and they are linked to (or linked from) other external data [9]. For each query of the log, at the preprocessing phase we can compute and store the results (or visualization of the results) returned by various keyword-based search services over LOD that accept free text queries [1, 29, 16, 20, 55]. These search engines exploit the underlying structure of the data, bypassing the more complex “application-oriented” SPARQL endpoints in order to deliver to users more human friendly responses. The presentation of these hypertexts (which can also contain images and videos) to the users during typing, gives them a quick overview of the semantic space related to the current words, and allows them to stop typing (and instead click on a suggested link or change what they have typed so far).

8.5 IOS and Vertical Search

Roughly, a vertical search engine is a dedicated engine aiming at providing specialized access services about a specific topic, industry, type of content, piece of data, etc. Some of this content cannot be found, or is difficult to find, using general search engines, and quite often this content has some standard metadata. For example, an answer returned by a vertical search engine over movies returns for each movie its title, director, scriptwriters, actors, genre, language, release date, trailer, storyline, soundtrack, photos, awards, etc. If these metadata are not stored in a single repository, but they are fetched from several different sources, then
their collection and aggregation is a time consuming task. IOS can be exploited for precomputing this information and offering it instantly.

8.6 Synopsis

We have seen various possible applications and Figure 8.5 summarizes the key points. The figure shows the possible steps of a standalone and meta search process capturing the aforementioned applications. Note that if the user’s query has been indexed with the proposed structure, then all steps which are enclosed by the rounded rectangle can be bypassed according to the IOS approach (they are just loaded from the index).

Various running prototypes that we have developed are available to use through http://www.ics.forth.gr/isl/ios.
Chapter 9

Concluding Remarks

In this thesis we elaborated on methods for enabling the instant provision of informative query recommendations that can accommodate the results of various tasks (like autocompletion, search-as-you-type, results clustering, metadata-based groupings, entity mining, etc.). These methods are generic in the sense that they are independent of the precomputed information (i.e. independent of the ranking, autocompletion, clustering, etc. method), meaning that the precomputed information can alternatively contain queries in a structured way [28, 30], diversified search results [47], etc. It is also important to clarify that the performance of these methods is independent of the size of the collection; it is affected only by the size of the query log (in particular by the number of the frequent distinct queries).

These informative recommendations give the users an overview of the information space and allow them to discover fast what is popular among other users, to decide fast which (of the suggested) query completions to use, and what hits of the returned answer to inspect (alleviating the users’ bias in the first page of results or the possible biases of the ranking method of the search system). Essentially, the interactive mechanism that we propose can assist the user in all steps of the search process. Specifically, in a keystroke basis, the user is getting information about the available options (recommended query completions), aid for choosing an option (appearance of supplementary information), aid for acting (no need to type the entire query so user actions become less laborious), aid for evaluating the choices (the user can see the number of results, cluster, entities, etc.), and aid for changing his situation (mental state may accompanied by a refined or different information need).

We paid special attention to the perspectives of scalability and flexibility. Regarding scalability we considered an approach based on precomputed information and we comparatively evaluated four trie-based index structures. The experimental results showed that a partitioned trie-based index structure can efficiently support recommendations for millions of distinct queries and terabytes of precomputed information. Then we described criteria that should be used and in the right order for selecting the most appropriate index according to the available main memory.
and the desired performance. For improving the throughput that can be served we analyzed experimentally various caching policies. According to our results a hybrid (70% static and 30% dynamic) caching policy seems to be the more appropriate choice yielding a throughput increment of around 80% and a 25% speedup.

For reducing user’s effort and increasing the exploitation of the precomputed information, we provided algorithms (appropriate for the proposed trie-based index structures) which can tolerate different word orders and spelling errors. The experimental results revealed that such functionality significantly increases the number of recommendations especially for queries that contain several words (the latter usually have none or very few suggested completions).

A direction for further research is to analyze how exactly users exploit the precomputed information that appears instantly. This requires very fast eye-tracking equipment for measuring how many times (and under what conditions) the user glances at the displayed precomputed information, and methods appropriate for analyzing the gathered information. That will also assist deciding where [42, 12] and how [15] it is worth displaying the recommended information. Another direction is to extend our approach for providing personalized recommendations (e.g. according to the collaborative approach [45]).
Bibliography


