Searching Application Data

Author(s):
Duda, Cristian I.

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005844429

Rights / License:
In Copyright - Non-Commercial Use Permitted
Searching Application Data

A dissertation submitted to the
ETH ZURICH
SWISS FEDERAL INSTITUTE OF TECHNOLOGY

for the degree of
Doctor of Sciences

presented by
CRISTIAN IOAN DUDA
M.Sc. University “Babeș-Bolyai” Cluj-Napoca, Romania
born 17 July 1980
citizen of Romania

accepted on the recommendation of
Prof. Dr. Donald Kossmann (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Prof. Dr. Gaston Gonnet (ETH Zurich), co-examiner
Prof. Dr. Thomas Hofmann (Google Inc. Zurich), co-examiner

2009
Abstract

Traditional Search engines solve well the problem of Web Search. On the other side, with the advent of new functionality in the Web and on the Desktop, data has new characteristics which cause search engines not to search it correctly. Versioned data, annotated data, AJAX Web Sites and Rich Internet Applications cause traditional search engines to return false positives or false negatives, since they do not understand application logic embedded in the data. This work addresses this challenge and builds search engines which see the data with eyes of the user, i.e., as a user would see it through the application. As opposed to traditional search, the new challenges posed by this goal are: modeling the data, indexing the content correctly, searching the content and, finally, understanding links which point inside application data, but are not simple URIs anymore.

The contribution and innovation of this work are techniques which correctly index application views. Views consist of instances or states (such as multiple versions for desktop applications, or client-side states for Web Applications). This is achieved through the following: new models are defined and built for the application data, either explicitly (using rules) or implicitly (during crawling), enhanced index structures and enhanced query processors are built, and instances are ranked using specially developed ranking methods.

The contributions are innovative, but are at the same time meant to be easily integrated into existing search engines. Enhanced index formats are defined as enhancements of traditional inverted files, i.e. predicate-based indexes. Optimization techniques such as normalization and memoization are developed and used in order to explicitly aim for a trade-off between the gain in result quality and the overhead in performance. New algorithms such as AJAXCrawl, AJAXRank and AJAXHITS are also developed, by adapting existing ones in a new context. Furthermore, experimental results are used to practically showcase the benefits of indexing application data on YouTube, Amazon, Wikipedia in terms of result quality, performance and functionality.

As a whole, this work introduces and defines the concepts, terminology and challenges, i.e., the toolkit needed in order to use, search, optimize and application data. By doing this, it defines the base for further novel research avenues.
Zusammenfassung

Die traditionellen Suchmaschinen lösen das Problem der Suche im klassischen Web und auf dem Desktop mit relativ guter Verlässlichkeit. Mit der Einführung neuer Funktionalitäten im Web und auf dem Desktop, erhalten die Daten jedoch neue Eigenschaften, die dazu führen, dass die traditionellen Suchmaschinen die Suche in diesen Daten nicht mehr korrekt ausführen. Versionierte und annotierte Daten, AJAX Web Seiten und Rich Internet Applikationen sind Fälle, in denen die traditionellen Suchmaschinen inkorrekte Suchergebnisse (false positives und/oder false negatives) liefern. Der Grund liegt daran, dass traditionelle Suchmaschinen die in den Daten eingebettete Anwendungslogik nicht berücksichtigen, obwohl ein Benutzer die Daten in der Anwendung korrekt sieht.

Die vorliegende Arbeit widmet sich diesem Problem und entwirft Suchmaschinen, die die Daten aus der Sicht eines menschlichen Benutzers sehen. Dabei müssen neue Herausforderungen bewältigt werden, die in traditionellen Suchmaschinen nicht existieren. Diese sind: Modellierung der Anwendungsdaten, korrekte Indexierung des Inhaltes, Suche, und Erkennung von “Links” die nicht mehr nur Verweise auf URLs sind, sondern welche in die Anwendungsdaten hineinzeigen oder sogar Zustandsüberführungen ausdrucken.

Der Beitrag und die Innovation dieser Arbeit sind Verfahren, die Anwendungsansichten (Application Views) korrekt indexieren. Die Ansichten basieren auf states oder instances (wie zum Beispiel mehrere Versionen für Desktop Anwendungen oder Client-Side States für Web Anwendungen). Dieses Ziel wird durch die folgenden Methoden erreicht: neue Modelle für die Anwendungsdaten werden definiert, entweder explizit (using rules) oder während des Crawlings, erweiterte Indexstrukturen und Query Proces-sors werden entwickelt, und die finalen Suchergebnisse (Instances) werden durch speziell angepasste Rankingmethoden bewertet.


Diese Arbeit präsentiert und definiert neue Konzepte, Terminologien und Herausforderungen, d.h. ein gesamtes Toolkit für die künftige Arbeit mit Anwendungsdaten, für ihre korrekte Nutzung, reibungslose Suche und fortwährende Optimierung. Somit definiert diese Arbeit auch eine Basis für angehende Forschungsvorhaben.
## Contents

1 Introduction xv
   1.1 Background and Motivation ........................................... xv
   1.2 Contributions .......................................................... xvi
   1.3 Structure of the Dissertation ........................................... xvii

2 Related Work xix

I Predicate-Based Indexing of Application Data 1

3 Problem Statement 3
   3.1 Motivating Examples ..................................................... 3
   3.2 Problem Description ..................................................... 5
   3.3 Plan of Attack ............................................................ 5

4 Patterns in Desktop Application Data 7
   4.1 Excluded content ........................................................ 8
   4.2 Comment ................................................................. 8
   4.3 Alternative ............................................................... 9
   4.4 Version ................................................................. 10
   4.5 Placeholder ............................................................. 12
   4.6 Field ................................................................. 12
   4.7 Multiple Rules and Conflicts .......................................... 13

5 Query Language 15

6 Normalization 17
   6.1 Examples ............................................................... 18
6.2 Normalization Algorithm .................................................. 18
6.3 Complexity Analysis ...................................................... 21
6.4 Benefits of Normalization ................................................. 22

7 Enhanced Indexing .......................................................... 23
7.1 Example ........................................................................ 23
7.2 Index Creation Algorithm ............................................... 24
7.3 Complexity Analysis ....................................................... 26

8 Enhanced Positioning ......................................................... 27
8.1 Example ........................................................................ 27
8.2 Definitions ..................................................................... 27
8.3 Assigning Dewey ID Positions: Algorithm ......................... 29
8.4 Optimizing Storage of Dewey IDs with Level ...................... 29

9 Skip Spaces ...................................................................... 31
9.1 Adding Skip Spaces: Algorithm ...................................... 32
9.2 Benefits of Enhanced Indexing ......................................... 32

10 Enhanced Query Processing .............................................. 35
10.1 Processing Keyword Queries .......................................... 35
   10.1.1 Example ................................................................ 35
   10.1.2 Merge Algorithm .................................................. 36
10.2 Complexity Analysis ..................................................... 37
10.3 Processing Phrase Queries ............................................ 37
10.4 Ranking ........................................................................ 39
   10.4.1 Analysis of a query result ....................................... 39
   10.4.2 Ranking functions ................................................ 40

11 Experiments ...................................................................... 41
11.1 Precision and Recall ..................................................... 42
11.2 Index Size ..................................................................... 42
11.3 Index Creation Time ...................................................... 44
11.4 Space Gain through Normalization ................................. 45
11.5 Query Processing Time .................................................. 46
11.6 Partial Indexing ............................................................ 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Basic Crawling Algorithm</td>
<td>77</td>
</tr>
<tr>
<td>18.2 Addressing Crawling Challenges</td>
<td>78</td>
</tr>
<tr>
<td>19 A Heuristic Crawling Policy for AJAX Applications</td>
<td>81</td>
</tr>
<tr>
<td>19.1 Intuition</td>
<td>81</td>
</tr>
<tr>
<td>19.2 Javascript Invocation Graph</td>
<td>82</td>
</tr>
<tr>
<td>19.3 Optimized Crawling Algorithm</td>
<td>83</td>
</tr>
<tr>
<td>19.4 Complexity Analysis</td>
<td>84</td>
</tr>
<tr>
<td>19.5 Simplifying Assumptions</td>
<td>86</td>
</tr>
<tr>
<td>20 An AJAX Search Engine</td>
<td>87</td>
</tr>
<tr>
<td>20.1 AJAX Crawler</td>
<td>87</td>
</tr>
<tr>
<td>20.2 Indexing</td>
<td>87</td>
</tr>
<tr>
<td>20.3 Query Processing</td>
<td>88</td>
</tr>
<tr>
<td>20.3.1 Processing Simple Keyword Queries</td>
<td>88</td>
</tr>
<tr>
<td>20.3.2 Processing Conjunctions</td>
<td>88</td>
</tr>
<tr>
<td>20.3.3 Ranking</td>
<td>89</td>
</tr>
<tr>
<td>20.4 Result Aggregation</td>
<td>89</td>
</tr>
<tr>
<td>20.5 Parallelization</td>
<td>90</td>
</tr>
<tr>
<td>21 Experimental Results</td>
<td>91</td>
</tr>
<tr>
<td>21.1 Experimental setup</td>
<td>91</td>
</tr>
<tr>
<td>21.1.1 YouTube Datasets.</td>
<td>91</td>
</tr>
<tr>
<td>21.1.2 Algorithms.</td>
<td>93</td>
</tr>
<tr>
<td>21.2 Crawling Time</td>
<td>93</td>
</tr>
<tr>
<td>21.3 Size of Application Model</td>
<td>95</td>
</tr>
<tr>
<td>21.4 Query Processing Time</td>
<td>96</td>
</tr>
<tr>
<td>21.4.1 Queries</td>
<td>96</td>
</tr>
<tr>
<td>21.4.2 Query Processing Time</td>
<td>96</td>
</tr>
<tr>
<td>21.5 Precision</td>
<td>97</td>
</tr>
<tr>
<td>21.6 Disadvantages of a full non-AJAX Approach</td>
<td>98</td>
</tr>
<tr>
<td>21.7 Recall</td>
<td>98</td>
</tr>
<tr>
<td>21.8 Partial Crawling. Setting Crawling Threshold.</td>
<td>98</td>
</tr>
<tr>
<td>22 Discussion: Integrating AJAX Crawl into Existing Search Engines</td>
<td>101</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>23</td>
<td>Demonstration</td>
</tr>
<tr>
<td>23.1</td>
<td>The AJAX Applications</td>
</tr>
<tr>
<td>23.2</td>
<td>Demo Scenarios</td>
</tr>
<tr>
<td>24</td>
<td>Related Work</td>
</tr>
<tr>
<td>25</td>
<td>Summary</td>
</tr>
<tr>
<td><strong>III</strong></td>
<td>Improving Hidden Web Crawling and Coverage using AJAX Suggestions</td>
</tr>
<tr>
<td>26</td>
<td>Problem Statement</td>
</tr>
<tr>
<td>26.1</td>
<td>Motivating Example (Amazon)</td>
</tr>
<tr>
<td>26.2</td>
<td>Problem Description</td>
</tr>
<tr>
<td>26.3</td>
<td>Plan of Attack</td>
</tr>
<tr>
<td>27</td>
<td>Qualitative Evaluation of Hidden Web Crawling</td>
</tr>
<tr>
<td>27.1</td>
<td>Metrics for the Quality of Hidden Web Crawling</td>
</tr>
<tr>
<td>27.2</td>
<td>Probabilistic Estimation of the Size of the Crawled Database (Random Sampling)</td>
</tr>
<tr>
<td>28</td>
<td>AJAX Suggest Crawling</td>
</tr>
<tr>
<td>28.1</td>
<td>Traditional Hidden Web Crawling</td>
</tr>
<tr>
<td>28.2</td>
<td>AJAXSuggest Crawling Model</td>
</tr>
<tr>
<td>28.3</td>
<td>Combined Crawling Model</td>
</tr>
<tr>
<td>29</td>
<td>Experimental Section</td>
</tr>
<tr>
<td>29.1</td>
<td>Experimental Setup</td>
</tr>
<tr>
<td>29.2</td>
<td>Effort</td>
</tr>
<tr>
<td>29.3</td>
<td>Quality of Crawling: Relative Coverage</td>
</tr>
<tr>
<td>29.4</td>
<td>Probabilistic Estimation of the Absolute Coverage</td>
</tr>
<tr>
<td>29.5</td>
<td>ROI: Return on Investment</td>
</tr>
<tr>
<td>30</td>
<td>Related Work</td>
</tr>
<tr>
<td>31</td>
<td>Summary</td>
</tr>
</tbody>
</table>
### IV  Ranking in AJAX Applications 135

#### 32 Problem Statement 137
- 32.1 Motivating Example 137
- 32.2 Problem Description 138
- 32.3 Plan of Attack 139

#### 33 Capturing Application Logic. 141

#### 34 AJAX Rank: Ranking States, Events and Application Structure 143
- 34.1 Ranking Requirements 143
- 34.2 Ranking Application States using PageRank 143
- 34.3 Ranking Application Events 144
- 34.4 Ranking Application Structure 146
- 34.5 Cumulative AJAXRank 148
- 34.6 Rank Aggregation 148

#### 35 AjaxHITS: Smart Crawling 151
- 35.1 The AjaxHITS Algorithm 151

#### 36 Experiments 153
- 36.1 Experimental setup 153
- 36.2 AJAXRank vs. PageRank 155
- 36.3 Influence of the Click-Distance 155
- 36.4 AJAXRank Iterations 156
- 36.5 Choosing the AJAXRank Parameter 156
- 36.6 Ranking Application Structure with AjaxHITS 157

#### 37 Discussion: Integrating AJAXRank and HITS into existing Search engines 161

#### 38 Related Work 163

#### 39 Summary 165

### V  Conclusions 167
- 39.1 Summary of the Work 169
- 39.2 Future Work 170
Acknowledgements

First of all, I would like to thank my advisor Donald Kossmann, for his vision and pragmatic view, for his guidance and also for the freedom he decided to gave me. These years have been quite an experience - for both of us I am sure, but I must say these have been the years when I truly grew up.

My co-advisors, Gustavo Alonso, Gaston Gonnet, Thomas Hofmann for their precious advice.

To Chong Zhou, with whom a lot of the work in the AJAX Crawling part of the dissertation has been refined and published. My master and semester students: Björn Jarisch, Gianni Frey, David Graf, Reto Matter, Florica Serban for their great combination of skills, fresh insights, resilience to stress and early-morning deadlines, and for the sometimes twisted points-of-views to which we arrived.

To my friend and colleague Marcos Vaz Salles, for an always insightful and detailed view on things, a master of everything from LaTeX spacing and join algorithms to Brazilian music, Jazz and wine. To Irina and Alex, my close friends and PhD colleagues, for the constant, invaluable support and traditional coffee breaks. Peter, who has been there since the beginning of my PhD with support and for showing me the ropes; to the whole group and friends, Eric Lo, Angela Nicoara, Lukas Blunschi, Ghislain Fourny, Martin Hentschel, Philipp Unterbrunner, Stefan Hildebrand, Carsten Binnig, Nesime Tatbul, Tim, Karin, Jens Dittrich, Simonetta Zysset, Heidy Schümperlin and Jena. Andreas Grunhagen and Andrea Rachel - my close German connection, and the whole Heidelberg group, where my PhD started.

To Deniz, for constantly calling me “Herr Doktor” and for keeping my feet on the ground, to Urs for his energy and to Jacques for his constant support. To my Romanian connection Ali, Simo & Cozmin, Cori & Calin, Luci and their families - for making me really feel like home.

I am so very grateful to my parents, who I hope understand how important they are to me, to my brother Adrian, thank you for always being there for me.
Chapter 1

Introduction

1.1 Background and Motivation

Current search engines, such as Google [67] and Yahoo! [115], solve well the problem of crawling the Web. However, searching application data is still an unsolved problem. Users access data through an application such as Word, Excel (desktop), E-Mail Client or a Web browser. However, Google and related products do not see the application data with the eyes of the user. Versioned documents are indexed as plain documents, annotated documents do not distinguish the main text from the annotated text, Rich Internet Applications contain client-side code that is ignored by search engines and their crawlers. This results in false negatives and false positives returned by the current search engines.

This work aims to make search engines index application data correctly, i.e., to have search engines see the data exactly as a user sees it, as shown in Figure 1.1. Indexing application data correctly is a multi-facet problem. It involves:

- Modeling the Data.
- Indexing the Content.
- Creating links inside the application data.

Modeling the Application Data involves either explicit description of the data using user-defined rules or implicit building of the model (e.g., during crawling, such as in the case of Web Applications, which results in an automaton of states. Modeling assigns therefore, the necessary semantics to the data.

Indexing the Content. Based on the previously developed model, an enhanced index must be built, so

![Figure 1.1: The Search Engine View differs from the Application View of the user](image-url)
that reflects the additional information available from the data. This index will be then queried, and results are returned as documents or as actual links to the correct parts inside the application data.

Creating links inside the application data. Search involves returning correct results to the user, but, as opposed to traditional search, also means reconstructing the specific item from the application data that is needed. This can be either a specific version or a specific state in an AJAX page, constructed as if the user had clicked on a series of events on the page. The notion of statefulness comes also into play. The consequences of indexing application data results of course in a gain over traditional search engines in terms of results quality. At the same time, however, there is a decrease in performance caused by the additional information to be crawled, incorporated into the index, and processed during search. The results quality vs. performance tradeoff is addressed in this work both in case of desktop and web application data. Optimization techniques such as memoization and normalization are used in order to gain performance compared to naïve approaches and traditional approaches. Partial indexing is a way to balance the quality gain with the performance quality.

1.2 Contributions

This dissertation addresses the challenge of indexing Application Data correctly. By doing this it brings the following contributions:

New models for Application Data. Declarative rules are used to model the patterns in desktop application data, such as versioned data. AJAX Search Web sites are modeled explicitly in a transition graph, during crawling.

Increased Quality of Search Results. Correctly describing application data means obtaining correct search results.

Normalization. In order to avoid to redundantly materialize common parts in the application data such as versioned parts, with the purpose of indexing application views correctly, an embedded data + rules format is developed, called normalized view. The new format does not need to be materialized, but brings considerable optimizations in time and space compared to a naïve approach and almost no overhead compared to the traditional, non-enhanced, search.

Memoization. In case of crawling AJAX Web sites in order to build the enhanced model of the data, where network calls and duplicate elimination of possible very granular events takes a crucial role in performance, this technique is used in order to timely optimize the application model size and crawling performance of the AJAX Crawler.

Enhanced Indexing Structures. Extended inverted files are built, which encode the additional properties of the application data. Enhanced index structures just enrich the structure of traditional index files, making the possible integration of the new techniques into existing search engines transparent.

Predicate-based indexing. One way to encode additional properties of the data is the simple and at the same time powerful method of predicate-based indexing, which associates boolean predicates to index entries in order to encode additional properties in the data.

Enhanced Query Processing. Query processing must take into consideration the additional information about the data. Join algorithms (such as line-sweep) are adapted in order to perform a correct computation of conjunction queries or phrase search.
Enhanced Ranking Methods. Results in application data are not simply document or URLs, but links inside the data. This work presents first studies on how traditional document-based ranking models such as $tf*idf$ have been adapted to these data instances. In particular the AJAXRank and AJAXHITS adapt the well-known PageRank[22] and HITS[83] algorithms to AJAX Web Sites with the purpose of building “Smart” AJAX Crawlers – another optimization possibility.

Result Quality vs. Performance Tradeoff. Building extended desktop and Web data models with the goal of increased quality of query results leads to an increase overhead over traditional search. We address it by studying partial indexing.

Result Aggregation. Returning results means being able to “click” on the link in order to “see” the result. Returning either the initial document itself or, preferably, the exact state inside the data which corresponds to the result (i.e., a version or a state to which a user would have clicked through in a Web site) is a challenge. We call this technique Result Aggregation and we derive it from the application models Normalized View and Transition Graph. The challenge is to take into account the notion of statefulness of search results in application data.

Improving Traditional Web Search. Enhanced Application Model provides better quality of search. The presented techniques require minimal improvement over traditional search techniques. Furthermore, we show how using the application logic which lately has been added into traditional contents (such as AJAX Suggest in applications such as Amazon) can actually highly contribute to the well-known problem of indexing Hidden Web by increasing precision and coverage.

1.3 Structure of the Dissertation

This thesis is structured as follows:

Part I. Predicate-Based Indexing of Application Data. This part of the dissertation presents how the patterns in desktop application data can be modeled using declarative rules and how a framework which indexes application data can be built. The generic notion of instances in application data is introduced, which covers all types of patterns including versions and annotations. The novel optimization technique called Normalization is described, used in order to avoid materialization of instances by indexing the application view directly. An enhanced indexing method called Predicate-Based Indexing is shown to be the key to encode instance (and rule) information into the index, in order for it to be searched using an enhanced Query Processor. Experimental results on Wikipedia, E-Mail and Software repositories confirm the gain in result quality and the minimal overhead in performance. Finally, Chapter 13 illustrates the extensibility of the framework, in order for it to be applied to enterprise data (JSP pages).

Part II. Crawling, Indexing and Searching AJAX Applications. After Part I focused on desktop search, Part II brings the focus on Web application data, in particular on AJAX Web Sites. Just as, for example, desktop versioned documents, AJAX and Rich Internet Applications (RIAs) are also not correctly handled by existing search engines. The application logic resides mostly on the client, and is driven through client-side events that create more application states under a same URL, thus breaking the traditional search paradigm. The first contribution of this work is a new model for AJAX Web sites. The second contribution is an AJAX crawler, which builds this application model. Crawling AJAX Applications is a difficult problem. First, the tools are missing, and second, possible large amount of states cause an explosion of the application model. We show that building an AJAX Crawler is possible, we propose optimization
techniques based on memoization, which address the challenges of large number of states, the amount of network requests as well as the problem of duplicate elimination. The experimental part which applies the crawler on YouTube [117] shows the increase in quality and the benefits of optimization techniques while a demonstration shows how the application behaves in a real situation, on the Yahoo!Mail and YouTube AJAX crawling. Both Part I and Part II study the trade-off between the goal to increase search result quality and that of maintaining a reasonable overhead.

**Part III. Improving Hidden Web Crawling and Coverage using AJAX Suggestions.** After studying the crawling strategies of an AJAX Crawler and the results in terms of result quality, Part III shows how AJAX Crawling (i.e., a crawler which is aware of the application logic) can be used in order to help solving the problem of Hidden Web Crawling. Finally, this part studies the notion of coverage in terms of crawling application data and its importance in estimating the quality and return on investment of an AJAX Crawler. All results in this paper show that it is possible and worth to invest in correctly indexing application data in several domains.

**Part IV. Ranking in AJAX Applications.** The first three parts of this dissertation focused on defining the model and on achieving coverage and quality. A component however which is still missing is the actual importance of application logic in returning result to the user, and also in the actual crawling operation itself. The main purpose of Part IV is to show how to fully take advantage of the semantics of application data, by defining a new ranking scheme for states of application data. The goal is first to correctly rank states based on application logic (for example, the order of states matters) and the second it to be able to define a smart crawling mechanism, so that only relevant parts of a page with potentially many AJAX events are crawled. For this purpose, AJAXRank and AjaxHITS algorithms are described. They extend and adapt existing Web ranking algorithms to AJAX Search, PageRank [22], HITS [83], XRank [73] and open a clear way to optimization possibilities.
Chapter 2

Related Work

This work is innovative since it offers for the first time the notion of searching application data, the related challenges and solutions for them. It relates to ideas presented in different existing works in different contexts, the most representative being presented here, with details in the corresponding parts of the thesis. This related work part gives pointers to works that see data and application logic as a unit.

Data formats that contain annotations have been addressed by [10] which solves phrase search in XML Data with annotations. Annotation and comments are just one of the patterns presented this work. ColorfulXML [78] addresses data which can be viewed as a set of instances and to which queries are specified on just one or more instances at the same time. We take a similar approach by providing a rule language which describes the multiple instances in the so called application views, to which we perform user queries. As opposed to our work, ColorfulXML's purpose is to provide a richer model for XML data, and an extension of the XQuery [21] language without optimizing for boolean retrieval and keyword search, and also for showing an optimized storage model for this data. The purpose of our work is similar, but the semantics associated to the data are based on the predicate-based indexing scheme [47] and covers the model of the ColorfulXML. Querying views on relational and XML databases as in [102] and ActiveXML [1], is a related field which we addressed in the specific context of information retrieval, while XQuery Full Text [8] is the standard query language for semistructured data with IR extensions.

Extracting and annotating relevant entities in the data has been IBM project Avatar [81] also at the border between databases and information retrieval, through their rule-based approach. The existing predicate-based approaches have been developed with another purpose in mind, [9]. Techniques to avoid redundancy in data, which has been the purpose of the patterns defined in this work, have been suggested by the XNF [14] in a formalization effort similar to Codd's Relational Databases normal forms [41]. More similar to our approach based on patterns, Broder e. all [6] have addressed redundancies, and again, this became a subcase of our proposed framework, as well as other patterns such as versions, addressed separately in [19].

Keyword search is a natural way to search for information retrieval which we also adopted in this work. This has been addressed in the database community by [3] or [20] in the context of relational databases. Indexing structures [28], storage compression [30], [85], join processing [6], [15] and XML Storage [78], [56] are common preoccupation of both the database and information retrieval community.

Ranking methods that take into account implicit properties of the data, such as for example XML's hierarchical structure - usually derived from a schema with associated semantics - have been addressed...
by XML Ranking schemes such as XRank [73]. Furthermore, adapting algorithms such as PageRank [22] to XML and other contexts than just the Web has also been addressed by XRank and [38]. Existing ID schemes for XML [72], [75], [95] have been at the base for developing in our work a new positioning scheme in Application Data. All Identified scheme for XML already take into account and embed in the annotation itself the functionality that is associated to the data they have been created for, e.g., XML, and the associated operations, path queries, inserts and deletes.

The transition of the Web itself from traditional to an application-based one (AJAX) has been addressed by [91] work which studies the effect of condensing an entire Web site into a single AJAX page. On the side of correct search, the advance of Web crawling from traditional to AJAX crawling has, of course, been addressed, but mostly at a basic engineering level. The change also occurs at the level of the commercial search engines which start to incorporate basic AJAX and Rich Internet Application search. Our work which studies the tradeoff between the amount of AJAX to be crawled and the improvement in result quality is in this context all the more important. Crawling client-side application logic and maintaining client-state and corresponding software toolkits have been addressed in works such as [86] but for testing purposes and not exhaustively. On the Web-related side, work on Web crawlers and Web Ranking Algorithms is relevant Google’s crawler and PageRank [22], HITS [83] and extensions [11]. Hidden Web [98] is an actual direction of research that we actually innovatively improve and quantify, by actually making full use of the new AJAX functionality being incorporated in new Web Sites.

Finally, we mention that the work presented in this dissertation has been presented in the following research conferences: Predicate-based Indexing in ICDE 2007 [47] and ETH Technical Report [48], Indexing Enterprise Applications Data in CIDR 2007 [55], AJAX Crawling in ICDE 2009 [53] and VLDB 2009 [54].
Part I

Predicate-Based Indexing of Application Data
Current search engines, such as Google [67], Yahoo! [115], Live Search [87], solve well the problem of crawling the Web. However, searching application data on the desktop is still an unsolved problem. Users access data through an application such as Word, Excel, Wiki (Web browser) or an E-mail client. However, Google and related products do not see the application data with the eyes of the user. The reason is that applications encode in the data certain properties such as annotations (e.g., comments in Word), versions (e.g., in a Wiki), E-mail threads and folders, and so on. Users see this with the applications, i.e., they see the application view. Search engines, on the other hand, crawl the data, without knowledge of the special properties (e.g., annotations and versions) encoded into the data. There are many challenges involved in searching application data: crawling the application data correctly, efficient indexing and query processing of the user view. The goal of this work is to extend today's search technology so that a search engine can see the data with the eyes of a user of an application. Rather than indexing the raw data, the extended search engine indexes views on application data and produces correct search results.

3.1 Motivating Examples

Example 1: Versioned Documents. Office applications use versioned data extensively (the Track Changes option). Community Wiki sites are another example, as well as the Open Office XML format. The application data encodes multiple sequential changes (inserts and deletes) in the same file. For example, Figure 3.1 shows an office document which contains all versions which have been created through a sequence of insertions and deletions, from 5am (original version) to 7am (final version). The user view is however composed of a set of successive versions. The consequence is that a query such as Mickey likes Daisy should not return a result since these two keywords never appear together in the same version. The application logic dictates this, although both keywords physically appear in the document stored on disk. Content converters for Microsoft Office and Open Office [96] partially solve this problem but cannot return the individual versions of a document as a result.

The versioning case also raises particular challenges for the case of phrase search. The phrase query "Donald likes Daisy" should indeed return the document as a result since the two keywords appeared consecutive in version 2, although they are not consecutive on disk. This example also shows the
problem of instantiation: the document might be relevant, but not the correct version is presented to the user. A naïve way to solve the problem and represent all versions separately would take redundant space to represent all versions (i.e., instances), since the versions have much content in common. As shown later, we will be able to index the application view without actually materializing it.

Example 2: Cross-Application Search. Searching data belonging to more applications is an unexplored venue in search. Mail merge functionality of office applications such as Word and Excel used in order to create bulk letters is one such example. Figure 3.2a shows a letter written in Word, containing placeholders for the recipient name, and an Excel spreadsheet containing names. When the Mail merge function is applied using the Word application, the connection between the documents is encoded in the Word document, but the actual letters are possibly never stored on disk. The application view (Figure 3.2b), however, contains all generated letters, since this is the meaning Word associates to this relationship. The consequence is that traditional search generates False Negatives (Figure 3.2c). The query Dear Paul should return the word document as a result, although the two keywords never appear together in the same document. The desired results are presented in Figure 3.2d.

(a) Application Data

<table>
<thead>
<tr>
<th>Id</th>
<th>Text Subpart</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance 1:</td>
<td>Mickey likes Minnie.</td>
<td>Time: 2/16/2008 (original version)</td>
</tr>
<tr>
<td>Instance 2:</td>
<td>Donald likes Daisy.</td>
<td>Time: 3/28/2008</td>
</tr>
</tbody>
</table>

(b) User View

<table>
<thead>
<tr>
<th>Query</th>
<th>Result of Trad. Search Engine (FP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mickey likes Daisy</td>
<td>versions.doc</td>
</tr>
</tbody>
</table>

(c) False Positives on the Application Data.

<table>
<thead>
<tr>
<th>Query</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mickey likes Daisy</td>
<td>-</td>
</tr>
<tr>
<td>Donald likes Daisy</td>
<td>versions.doc [Instance 2]</td>
</tr>
</tbody>
</table>

(d) Expected Query Results on the Application View.

Figure 3.1: Versions function on a group_by and order_by scenario.
3.2 Problem Description

An application view is a set of instances which can be generated from the original data. Our challenge is to create, index and search application views. Keyword queries applied on the application data should return results in terms of instances in the application view, and not in terms of the original document. The motivating examples demonstrate why search engines should index application views rather than the raw data. Furthermore, the examples show the need to normalize (compress) the data in an application view. It would be wasteful to index all instances of Figure 3.1 individually because the versions contain redundancy. The common parts of the two instances should be factored out and indexed only once; only the variable parts (the recipient) should be indexed separately. The more instances are generated, the more important this optimization becomes. Also, by doing this, search patterns are formalized at the data level and this allows search functionality to be factored out from the actual application.

3.3 Plan of Attack

In order to solve the problem of correctly searching the application views on desktop data, we propose to extend desktop search engines as shown in Figure 3.3. View Definition, Indexing and Search are defined, and they work on top of a common data model for application data.
**View Definition.** In case of desktop application data, indexing and search is performed in the granularity of instances. To this purpose, we describe the patterns appearing in the data and the instances generated by the patterns. Furthermore, we specify patterns using declarative rules. The rule language that must be used in this work is based on XSLT and XQuery. Finally, we also specify the query language and operators used for the application data.

**Normalization.** The declarative Rules define the application view (e.g., Instances). However, since the application views may be very large (even for only a few Megabytes of raw data (Section 11), the instances are never materialized. Instead, the view is directly normalized. The view definition and normalization are described in Section 6 and results in one normalized view for each application view. The normalized view is typically only slightly larger than the original data (as shown in Section 11) so that it can be materialized.

**Enhanced Indexing.** Indexes (extensions of inverted lists) are built on the normalized view. The structure of these extended inverted files is described in Section 7. Again, these extended inverted files are moderately larger than inverted files for traditional information retrieval (Section 11).

**Enhanced Query Processing.** At query time, the extended inverted files are used in order to find matching documents for a keyword query. Again, an extended version of the classic *merge* algorithm of inverted lists is used. This extended algorithm is described in Section 10.

**Result Aggregation.** The normalized view is used in order to compute the query results. Using the normalized view, it is possible to generate the instance of Figure 3.1 a (i.e., a single Version) and refer to the Word document of Figure 3.1 (i.e., the definition of the version). Both kinds of results may be relevant for the user.

The rest of this part is organized as follows: Chapter 4 presents the patterns appearing in application data and a declarative language for defining views on application data. Chapter presents the query language on application data, while Chapter 6 presents the main contribution of this work, i.e., normalization. Chapter 7 presents the predicate-based indexing. Two supporting indexing techniques are introduced in Chapter 8: a new positioning scheme based on Dewey IDs, and Chapter 9: skip spaces. Together with Chapter 10 on enhanced query processing, they constitute the basic operations and contributions of this enhanced IR framework. Chapter 11 presents an experimental evaluation of the enhanced search framework, in terms of performance and improvement in precision and recall over traditional search. Chapter 12 addresses enterprise application data and specific patterns, while Chapter 13 describes a demo of the framework. This is followed by Related Work and Conclusions in Chapter 14 presenting related work and Chapter 15 which conclude and present future work.

![Figure 3.3: Enhanced Desktop Search](image-url)
Chapter 4

Patterns in Desktop Application Data

As presented in Chapter 3, the challenge of Indexing Annotated Data is to understand and optimize the semantics of the data. Concretely, this is expressed as an application view, consisting of a set of instances. The purpose of this section is to define this view generically, so that it covers all use cases.

As explained before, from our experience, application data exhibits certain patterns, i.e., a few types of view definitions are sufficient to define the application views on this data. Conceptually, a pattern is a function from the application data to the instances. We identified these patterns and created an XML-based language which defines rules. A rule declaratively associates patterns to the data. We propose one rule language based on XSLT and XQuery which fulfills this purpose. The syntax of the language is however not important for the concepts in this work. Furthermore, since the proposed rule language is not a transformation language, it does not have (or need) the full expressive power of XSLT and XQuery.

Typically, the same set of patterns can be applied to all documents of the same kind. For instance, only a subset of the patterns is necessary for all Word documents; a different set of patterns would apply to the E-Mail repository of an IMAP server. In our experience, it is very rare that pattern applications need to be customized for individual documents; instead, the patterns are applicable to the data format as a whole. Therefore, in general, the pattern applications (i.e., rules) could be defined by the application provider and made available to users. This section gives a brief overview on some of these patterns by examples and show their corresponding rules.

Figure 4.1 showcases the existing patterns and their rationale. There is a rule corresponding to each

![Figure 4.1: The Rule Language](image-url)
pattern which declaratively associates the pattern to the data. The patterns and their corresponding rules are explained in this section, following the order of complexity. We also mention that the patterns which describe Examples 1 and 2 from Section 3 are Version and Field.

### 4.1 Excluded content

The first pattern is *Excluded*. It specifies that parts of a document should not be considered in a search query. Examples are formatting instructions which a user is typically not interested in for searching.\(^1\) A similar pattern to exclude certain fragments from an XML document for phrase search has been proposed in the PIX project [10]. Figure 4.2 gives an example. Figure 4.2a shows the original data with the formatting instructions. Figure 4.2b shows the view (without formatting instructions) as it is relevant for most users. Figure 4.2c shows the rule that is used in order to declare that the formatting instructions of the original document should not be included in the view. The name of a rule describes the pattern and the *match* attribute contains an XQuery expression that defines to which elements the rule should be applied.\(^2\)

### 4.2 Comment

Another pattern is *Comment*. Examples for Comments are footnotes or inlined annotations in a text. Figure 4.3 gives an example. Figure 4.3a shows the original document. Figures 4.3b and 4.3c show two instances that could be of interest to the user: one which contains the comment (identical to the original) and one that does not contain the comment. Figure 4.3d shows the rule that users can use in order to declare that they are interested to query the two instances separately. As in the case of Excluded, the motivation for this pattern was provided by the PIX project [10]: Only the application of this pattern makes it possible to apply a phrase search for “Mickey likes Minnie”, since the original document would not match. Both instances in Figure 4.3 are therefore part of the view.

\(^1\)Users who wish to, say, search for all documents that use a particular font would use a different view to query the data.

\(^2\)Here and in the remainder of this work, XML namespace declarations are omitted for brevity. Of course, the rule language defines its own namespace in order to avoid ambiguity with other XML names.

```xml
<office:font-decls>
  <font-decl style:name="Lucida Grande"/>
</office:font-decls>
Mickey likes Minnie.

(a) Original Data
Mickey likes Minnie.

(b) Instance
<excluded match="//office:font-decls" />

(c) Rule (R1)
```

Figure 4.2: Excluded Example.
4.3 Alternative

The Alternative pattern specifies that one out of several options of a text is chosen. For example, a song could contain markup that specifies for which audience (e.g., adults or general public) certain parts are targeted. Likewise, an electronic health record could contain markup that indicates which doctor is allowed to get what kind of information from the patient. The Alternative pattern is also useful to specify that a Web page could have versions in English and Chinese; the images and tables of the Web page would be identical for both versions, but the text should either be only in English or only in Chinese.

Figure 4.4 gives a simple example. The original document contains markup with option elements that indicates whether text fragments belong to the world of Mickey Mouse or Donald Duck. If the text is viewed from the perspective of the world of Mickey Mouse, it should read “Mickey likes Minnie” (Figure 4.4b); otherwise it should read “Donald likes Daisy” (Figure 4.4c).

The rule in Figure 4.4d specifies exactly these semantics. It contains three properties:

- **match**: The match attribute contains an XQuery expression that describes the target items of the document that are affected by the rule; option elements in this example. Furthermore, the XQuery expression in the match attribute binds a variable $m$ that is used in the key specification.

- **key**: The key attribute contains an XQuery expression that uniquely identifies an alternative. For example, all text fragments that belong to the world of Mickey Mouse can be identified by the value “mouse” in the world attribute (the key) of option elements (the target of the rule).

- **optional**: This attribute contains a Boolean value and specifies whether an alternative should be generated that does not contain any option elements. In this example, the optional attribute is set to false. If it had been set to true, a third instance with content “likes .” would have been generated (not shown in Figure 4.4).
Logically, Alternative rules are evaluated in the following way. First, the whole document is inspected in order to find all key values (typically, atomic values such as strings and numeric values). For each such key value, an instance is generated by inspecting the document again and ignoring all target nodes that do not match that key value. If the optional attribute is set to true, an additional instance is generated that ignores (i.e., excludes) all target nodes.

Alternatives partition the document. In order to draw an analogy, the expression in the key attribute corresponds to the expression of a GROUP BY clause of a SQL query with the match expression acting as the FROM clause. As a result, the number of instances in a view specified using this pattern grows linearly with the number of key values specified in the document. It is worth mentioning that the Comment is a special case of the Alternative pattern. It is given by an alternative with two options, one of which is empty.

4.4 Version

The Version pattern is useful for Example 1 of Section 3. It is also useful for Wiki data and for documents generated by Word and OpenOffice which allow the tracking of changes. In contrast to the Alternative pattern which partitions the elements that match an Alternative rule, the Version pattern orders all matching elements and specifies that the view contains an instance for each subsequence.

We are giving here a more complex and more frequently met in practice versioned document. Figure 4.5 shows a document in the simplified OpenOffice data format. Initially (before 6h), the document was “Good news is rare” (Figure 4.5b). After the sequence of insertions and deletions, the document became “No bad news is good news” at time 6h (Figure 4.5c) and “No news is good news” at time 7h (Figure 4.5d). The versioned elements are nested exactly as in the original office format.

The rule shown in Figure 4.5e declares that elements selected by the match expressions, i.e, ins and del elements, are versioned. The moments of time when versions occurred are defined by the key expression, which select the values 6h and 7h. The action attribute specifies where on the timeline, the
content in the ins and del elements is valid. For example, the content of ins elements is valid on and after the time indicated by the value of the key attribute, specified by the literal AFTER_AT in the action attribute. Similarly, the content of the del attribute is valid before the time indicated by the value of the key attribute, indicated by the literal BEFORE in the action attribute. The other values for the action attribute are the literals BEFORE, BEFORE_AT, AFTER_AT, AFTER corresponding to the values <, ≤, ≥, > This way, it is possible to see the period in which a document contained a certain phrase and query for the last version, ignoring content from previous versions. The nesting in the format is used in order to ensure that the content that was inserted and further deleted has clearly defined time borders.
4.5 Placeholder

Placeholders specify that data stored in a different document or data stored within the same document at a different location should be inlined. The location can be in the same file or external, which means that the Placeholder pattern enables cross-application search. In addition to a match attribute which specifies the target of the rule (as for all other rules), a Placeholder rule has a ref attribute that contains an XQuery expression, which specifies the data that should be inlined. Figure 4.6 gives a simple example. It shows an Office document with a footnote (Figure 4.6a). In this document, the texts of footnotes are stored separately from the actual text. The Placeholder rule (Figure 4.6c) brings a copy of the footnote text to the right location (Figure 4.6b). As in the Alternative rule, the match expression binds a variable $m which can be used in the ref expression.

An application of a Placeholder rule transforms an XML document into exactly one instance. It is possible that the ref expression refers to a different document (e.g., a spreadsheet). Any XQuery expression is allowed in the ref expression. Furthermore, it is possible that the ref expression evaluates to a sequence of several items (values and nodes, possibly hundreds or even thousands). In that event, the whole sequence is inlined. How to generate a set of instances if the ref expression evaluates to a sequence is described in the next subsection.

4.6 Field

A Field rule combines the behavior of an Alternative and a Placeholder rule. It is a useful rule to implement Example 2 of the introduction. Figure 4.7a shows the Word document (a letter) referring to the Excel spreadsheet of Figure 4.7b that contains a list of names; one instance of the letter is generated for each name (Figure 4.7c and Figure 4.7d).

Figure 4.7e gives an example of a Field rule and how it is applied to the XML document. Like a Placeholder rule, a Field rule has a match and a ref expression. Furthermore, a Field rule has a key and an optional expression like an Alternative rule.

The Field rule of Figure 4.7e specifies that for each line in the Excel spreadsheet “names.xls” an instance

```xml
<note id="ftn0">He is a Disney Character.</note>
Mickey Mouse <ref id="ftn0"/> likes Minnie.

(a) OpenOffice Document with a Footnote (Original Data)

<note id="ftn0">He is a Disney Character.</note>
Mickey Mouse
<ref id="ftn0">
  <note id="ftn0">He is a Disney Character.</note>
</ref> likes Minnie.

(b) Instance with a Copy of the Footnote Text

<placeholder ref="/note[@id=$m]/id]'/>

(c) Rule
```

Figure 4.6: Placeholder Example.
should be generated. In a Field rule, the ref expression binds a variable $r$ which can be used in the key expression. In this example, the key of each instance that is generated is the line number (i.e., the attribute id).

As in all other classes, the match expression binds a variable $m$ which can be used in the ref and key expressions. The use of the $m$ variable is not shown in the example rule of Figure 4.7d. Furthermore, this example rule does not specify the optional attribute. By default optional is set to false so that it need not be specified if that is the desired value.

Figures 4.7c and 4.7d show the two instances that are generated if the Field rule of Figure 4.7e is applied to the document of 4.7a. Just like Alternative rules, the number of instances generated by a Field rule grows linearly with the domain of the key expression.

### 4.7 Multiple Rules and Conflicts

Section 4.4 already gave an example in which several rules were applied to one document. Furthermore, each rule can be applied several times to different matching items of the same document. In such an

```xml
<word>
  Dear <recipient/>
  The meeting is today at 2am.
  Peter Jones Project Manager.
</word>

(a) Word document (Original Data)

```xml
<excel>
  <line id="1">Paul</character>
  <line id="2">Mary</character>
</excel>

(b) Excel Spreadsheet (Original Data)

```xml
<word>
  Dear <recipient>Paul</recipient>
  The meeting is today at 2am.
  Peter Jones Project Manager.
</word>

(c) "Dear Paul..." (Instance 1)

```xml
<word>
  Dear <recipient>Mary</recipient>
  The meeting is today at 2am.
  Peter Jones Project Manager.
</word>

(d) "Dear Mary..." (Instance 2)

```xml
<field match="//recipient" key="$r/@id"
    ref="doc("names.xls")//line />

(e) Rule

Figure 4.7: Field Example.
environment, two questions arise: (a) in which order should those transformations be applied; and (b) what happens if two (conflicting) rules should be applied to the same matching item.

Following the SQL standard [104] and the proposal for the XQuery Update Facility of the W3C [33], we use Snapshot semantics, which specify that rules are applied in two phases. In the first phase, all matching items are marked and the expressions referring to each matching item are pre-computed using the original version of the document(s). In the second phase, the instances of the document may be constructed, thereby combining the pre-computed transformations of the matching items of the first phase. This step is implementation dependent. Our approach does not physically construct the instances. In other words, all rules are always applied to the original document and rules are never composed. If two rules are in conflict and match the same item, then the process aborts with an error.

Snapshot semantics served the purposes of all use cases that we have looked at so far very well. Nevertheless, it is worth-while to study alternative models. For instance, a model that supports composability of rules can reduce the number of classes that are needed; e.g., the Field rule could be expressed as a composition of Alternative and Placeholder rules. We plan to study such models as one avenue for future work.
Chapter 5

Query Language

This section defines the operators which will be part of the queries performed on the Application Data. As opposed to traditional keyword search, the results of all these operations are instances, instead of files.

Retrieval Model. This works falls into the boolean retrieval model. This means an instance is returned if the query keyword(s) appear in this content. However, we relax this strict requirement by adding a ranking model, extending classical ranking methods from documents to instances of documents.

The two main operations of the query language are presented in Table 5.1. In this work we will focus on operators AND and NEXT.

AND. This operation corresponds to basic keyword search on application data. The input of this operator is a list of two or more keywords. The result is a list of instances which contain all of the keywords given as input.

OR. This operation corresponds to basic disjunction on application data. The input of this operator is a list of two or more keywords. The result is a list of instances which contain either of the keywords given as input.

NOT. This operator corresponds to basic negation. The input of this operator is a single keyword. The result is a list of instances which do not contain the keyword given as input.

NEXT. This operation corresponds to phrase search on application data. As opposed to keyword search, this operation returns all instances where all keywords given as input are contained and are adjacent. From this point on, we define adjacent content the content that is adjacent in at least one of the instances.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w_1 \text{ AND } w_2)</td>
<td>Conjunction</td>
<td>Instances containing both keywords.</td>
</tr>
<tr>
<td>(w_1 \text{ OR } w_2)</td>
<td>Disjunction</td>
<td>Instances containing either keyword.</td>
</tr>
<tr>
<td>NOT (w_1)</td>
<td>Negation</td>
<td>Instances not containing the keyword.</td>
</tr>
<tr>
<td>(w_1 \text{ NEXT } w_2)</td>
<td>Phrase Search</td>
<td>Instances where keywords are adjacent.</td>
</tr>
</tbody>
</table>

Table 5.1: Query Language on Application Data
of the application data. Phrase search in application data is not a trivial problem for which we devised a new positioning scheme, as it will be explain in Section 7.
Chapter 4 described the patterns and rules in desktop application data. The goal of this work is to index application views, i.e., the instances described by the patterns. The question is how to index application views efficiently.

**Naïve Approach.** The most straightforward implementation is to actually *materialize* on disk the instances determined by the patterns in the data. The consequence is that instances exist as documents on disk and they can be then directly accessible and indexable by a traditional search engine. The advantage is that although the engine does not understand the semantics of the initial application data, it can now index the application view and return instances as a result. The Naïve approach has negative consequences:

1. *Inefficient use of space.* Materializing all instances causes common content to be redundantly materialized several times. This effect can be especially observed in case of patterns such as versions, for which there is big overlapping. The same negative effect can be seen in case of multiple rules applied to the same document.

2. *Increased indexing time.* The materialization step has a negative influence on indexing time. This is especially true in case of more complex rules applying to nested elements, especially Versions.

**Normalization.** The alternative to the Naïve Approach is the process of Normalization (see Figure 3.3). Its role is to avoid the actual creation of instances, but still to be able to eventually generate an index in the granularity of instances. Normalization gets a document and a set of rules as input and generates a normalized document that represents all instances of the corresponding application view on that document. As a result, indexing and query processing can directly operate on the normalized view and need not be aware of the set of rules that were applied.

The challenge of normalization is to keep the normalized view as compact as possible, which is done by *factoring out* the common parts of a document which are not affected by the rules. Such a compact representation exists for all the patterns described in Section 4.
6.1 Examples

Normalization can be best described using an example. Figure 6.1 shows the normalized view for the example of Figure 4.4. This normalized view encodes the two instances “Mickey likes Minnie” (Figure 4.4b) and “Donald likes Daisy” (Figure 4.4c). The key idea of normalization is as follows: markup the variable parts of the data using special select elements. In this example, all option elements are tagged in this way, indicating that all option elements are variable. Common parts of the original data (e.g., “likes”), which are not affected by any rule, are not tagged.

As shown in Figure 6.1, each select element contains a predicate that specifies in which kind of instances of the view its content should be considered. The predicate $R3=\text{mouse}$, for instance, specifies that the $\text{option world = "mouse"}>\text{Mickey} <\text{option}>$ and $\text{option world = "mouse"}>\text{Minnie} <\text{option}>$ elements should be included into the instance that represents Mickey Mouse’s perspective on the original data; however, these two elements should not be included into the instance that represents Donald Duck’s view of the world. Note that $R3$ is a variable with a generic name that is generated for the Alternative rule of Figure 4.4d. mouse and duck are the values of the evaluation of the key expression of the rule of Figure 4.4d. The predicate specifies an equality because the rule of Figure 4.4d is an Alternative rule, and equality corresponds to the semantics of the Alternative pattern.

Another example of a normalized view is given in Figure 6.2. This normalized view encodes all the instances of the view described in Figure 4.5. Again, select elements define the variable parts of the document and predicates specify the inclusion of such variable content in instances. In case of ins elements, which correspond to the match expression of the Version rule with an AFTER_AT action (Figure 4.5e), the predicates are ”$\geq$” predicates. Similarly, del elements match the Version rules with an BEFORE action, and the predicates are ”$<$” predicates (rather than equalities for the Alternative pattern). $R4$ is an identifier generated for the Version rules of Figure 4.5e; the time values in the predicates are computed from the key expressions of that rule. The results of the key expression must belong to a domain on which a total ordering can be defined.

6.2 Normalization Algorithm

We detail in Algorithm 6.2.1 the construction of the Normalized View of a document based on given rules.

```
<select pred="R3 = mouse">
    <option world="mouse">Mickey</option>
</select>
<select pred="R3 = duck">
    <option world="duck">Donald</option>
</select>
<select>likes</select>
<select pred="R3 = mouse">
    <option world="mouse">Minnie</option>
</select>
<select pred="R3 = duck">
    <option world="duck">Daisy</option>
</select>
```

Figure 6.1: Normalized View - Alternatives
Algorithm 6.2.1 Normalization Algorithm

1: Input: Document d, Rule[] R
2: Output: Document NV {Normalized View}
3: 
4: Function constructTaggingTable(Document d, Rule[] R): TT
5: Output: TT(ruleId, pattern, nodeId, keyvalue, op, optional) {Tagging Table}
6: LET matches = {}
7: for all rule ∈ R do
8: if rule.type = "Alternative" OR rule.type = "Comment" OR rule.type = "Excluded" then
9: rule.operator = "="
10: for all m ∈ rule.match(d) do
11: if m ∈ matches then
12: 'ERROR: Conflict!' {Node already matched by other rules.}
13: else
14: if rule.ref = NULL then
15: key = rule.key(m)
16: if key ≠ NULL OR rule.type = "Comment" then
17: INSERT INTO TT(ruleId, pattern, nodeId, keyvalue, op, optional)
18: VALUES (rule.ID, rule.pattern, m.nodeId, key, rule.operator, rule.optional)
19: else
20: refs = rule.ref(m) {Evaluate key on each referenced node}
21: for all $r ∈ refs do
22: key = rule.key(m, $r)
23: if key ≠ NULL then
24: INSERT INTO TT(ruleId, pattern, nodeId, refId, keyvalue, operator, optional)
25: VALUES (rule.ID, rule.pattern, m.nodeId, key, rule.operator, rule.optional)
26: end if
27: end for
28: end if
29: end for
30: matches := matches ∪ rule.match(d) {Adding matches of current rules to set of matched items.}
31: end for
32: end Function

33: Function constructNormalizedView(Document d, Rule[] R, TaggingTable TT): NV
34: Output: Document NV {Normalized View}
35: for all $n ∈ d do
36: if ∃ $row ∈ TT where $n.nodeID = $row.nodeID then
37: if $row.pattern = "Alternative" OR $row.pattern = "Version" OR $row.pattern = "Comment" OR $row.pattern = "Field" then
38: $select = new Node(<option pred = "$row.ruleID $row.operator $row.keyvalue"/>)
39: $select.content = $n
40: NV.append($select)
41: if $row.pattern = "Comment" OR $row.optional = "true" then
42: $select := new Node(<option pred = "$row.ruleID $row.operator NULL"/>)
43: NV.append($select)
44: end if
45: else if $row.pattern != "Excluded" then
46: NV.append($n)
47: end if
48: end for
49: end Function
Chapter 6. Normalization

Figure 6.2: Normalized View - Versions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pattern</th>
<th>NodeId</th>
<th>KeyValue</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>Alternative</td>
<td>1</td>
<td>mouse</td>
<td>=</td>
</tr>
<tr>
<td>R3</td>
<td>Alternative</td>
<td>2</td>
<td>duck</td>
<td>=</td>
</tr>
<tr>
<td>R3</td>
<td>Alternative</td>
<td>4</td>
<td>mouse</td>
<td>=</td>
</tr>
<tr>
<td>R3</td>
<td>Alternative</td>
<td>5</td>
<td>duck</td>
<td>=</td>
</tr>
</tbody>
</table>

Figure 6.3: Tagging Table (Alternatives Example).

The Normalization Algorithm has two phases:

**Phase 1 - ConstructTaggingTable**: TT Constructs a Tagging Table with the schema TT(ruleId, pattern, nodeId, key, operator, optional).

1. For each node \( n \in d \) returned by the *match* expression of a rule \( r \in R \), insert a tuple \( t \) in \( TT \) and make \( t.RULE \leftarrow r \), \( t.PATTERN \leftarrow r.PATTERN \), and \( t.NODEID \leftarrow n \) (lines 11-30).

The Tagging Table of Figure 6.3 is constructed for the example of Figure 4.4, and Table 6.4 is constructed for the example in Figure 4.5. Every tuple in the Tagging Table contains the ID of a

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pattern</th>
<th>NodeId</th>
<th>KeyValue</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4</td>
<td>Version</td>
<td>1</td>
<td>6h</td>
<td>&lt;</td>
</tr>
<tr>
<td>R4</td>
<td>Version</td>
<td>3</td>
<td>6h</td>
<td>≥</td>
</tr>
<tr>
<td>R4</td>
<td>Version</td>
<td>5</td>
<td>7h</td>
<td>&lt;</td>
</tr>
<tr>
<td>R4</td>
<td>Version</td>
<td>8</td>
<td>6h</td>
<td>&lt;</td>
</tr>
<tr>
<td>R4</td>
<td>Version</td>
<td>10</td>
<td>6h</td>
<td>≥</td>
</tr>
</tbody>
</table>

Figure 6.4: Tagging Table (Versions Example).
matched node (e.g., 1 is the NodeId for the `<option world="mouse">Mickey</option>` element in Figure 4.4a) and the identifier of the rule that matches that node.

2. Evaluate the key expression of $r$ on $n$ and set $t.KEYVALUE$ to the value of the key expression and $t.OP$ to the operator implied by rule $r$. For instance, in Figure 6.3, for node id 1, we set the key value to "mouse" and, as rule R3 is an Alternative, the operator to equality. Similarly, in Figure 6.4, for node id 2, we set the key value to "6h" and the operator to $\geq$, corresponding to an insert and "AFTER AT". (lines 16-18 and 24-25).

3. If a rule $r' \in R$ matches a node $n$ previously matched by a rule $r \in R$, then a conflict occurs. As mentioned in [49], one possible way to deal with conflicts is to return an error (line 12). For the purpose of this work, we follow exactly these semantics, as explained in Section 4 and used in XQuery Update Facility [33].

Phase II - ConstructNormalizedView($d$, $R$, $TT$): constructs the normalized view $V$ by scanning through each tuple $t$ of Tagging Table $TT$ and executing the following transformations on document $d$, depending on $t.PATTERN$:

1. If $t.PATTERN$ is either Alternative or Version, then a select element is generated that embraces the node of $d$ which is identified by $t.NODEID$ (lines 39-41). The pred attribute of this select element is set to “$t.RULE \ t.OP \ t.KEYVALUE$” (e.g., R4 $\geq$ 6h from Figure 6.4).

2. If the rule specifies an Excluded pattern, then the matching node is simply not included in the normalized view document and no select element is generated in this case (lines 47-48).

3. If the rule specifies a Comment pattern or the rule has an optional attribute set to true, then an empty node is additionally generated, marking the fact there there is an instance without any included content (lines 42-45).

4. If $t.PATTERN$ is any of the other patterns, then we address them accordingly in a straightforward way. This is possible since their semantics is based on that of Alternatives or Versions.

Both the construction of the Tagging Table and the construction of the normalized view can be implemented quite easily using XSLT or XQuery. For instance, the implementation used for the performance experiments (Section 11) was based on Microsoft’s XSLT processor which is integrated into the .NET framework.

This work overloads the term Normalization as used in the work of Libkin [14] on the XML Normal Form (XNF). XNF proposes a model used for eliminating redundancies from XML data. We also use the term Normalization for describing an operation which eliminates redundancy: instead of materializing instances and redundantly storing their common parts, takes the common parts only once, and tags the variable instance parts separately. Relationships between instances and the original data can be modeled as functional dependencies. This has not been the purpose of this work though, and we leave it as an avenue for future work.

6.3 Complexity Analysis.

Normalization either enhances the application data with additional information or materializes a join in case of referred content. We define $N$ the size of the initial document and $M$ the size of the (eventually)
Chapter 6. Normalization

referred data.

**Complexity in terms of size.** Complexity of Normalization depends on the pattern(s) in the application data. The most complex pattern is Field, which expresses a join between application data and external content. Although the expression in the referred content can be arbitrarily complex, the usual case is more simple: i.e., external content is included once in the application data (such as the Bulk Letter example in Section 3). If \( N \) is the size of the data, and \( M \) is the size of the referred content, the size of the normalized View is, therefore, \( O(N + M) \).

**Generation Time.** Generation time depends on the complexity of the XQuery expressions. In practice, and for a single rule, the complexity is \( O(N) \).

### 6.4 Benefits of Normalization

The normalized view, as described in this section, brings the benefits described below:

**Completeness.** The normalized view encodes all the instances of the (potentially huge) application view, in a way which is independent of the rules that specify the view. This way, the normalized view can serve as a basis for indexing and all further query processing. Having such a rule-independent representation of data makes it possible to extend the rule language as new patterns become important, without adjusting the indexing and query processing components.

**Compactness.** The normalized view is a compact representation of the original data. If materialized in an unnormalized way, the space requirements for storing the application view grows exponentially with the number of rules. In contrast, the size of the normalized view is always in the same order as the size of the original document: at most two additional select elements are generated for each element of the original document. The normalized view can be further compressed in order to reduce its size. For example, dictionary-based compression was used in the implementation in order to represent the two selections mouse and duck by a single Bit in the example of Figure 6.1. Furthermore, it is not necessary to serialize the normalized view as XML; in our implementation, we used a serialized XML format that is based on tokenization [61]. Such an implementation reduces the size of the normalized view and significantly speeds up query processing because no parsing is required.

**Instance Computation.** The key idea of normalization is to tag variable parts of a document, and to give predicates that specify which instances of the view involve which parts of the document. As a consequence, specific instances of the normalized view can be retrieved by instantiation of the variables of these predicates. The “Mickey likes Minnie” instance of the normalized view of Figure 6.1, for example, can be retrieved by instantiating the \( R3 \) variable to “mouse”. Similarly, the second version of the document in Figure 4.5 can be computed by instantiating the \( R4 \) variable to “6h”. This way of referencing individual instances is exploited for indexing, which is described in Section 7.
Chapter 7

Enhanced Indexing

In this Chapter, we describe how to extend a conventional inverted file to provide a fine-grained representation for individual instances. A naïve approach toward fine-grained indexing is to create index entries for each instance of the application view separately and to include all these entries into the inverted file. However, this approach is equivalent to naïve materialization of the application view and is clearly not viable. The key idea to achieve both fine-grained indexing and tolerable index sizes (and thus good query performance) is, again, to factor out common parts, index those only once, and to index only the variable parts individually.

7.1 Example

Figure 7.1 shows the extended inverted file of the example in Figure 4.4. In addition to the document ID ($d_1$ is used in this example to refer to the original data of Figure 4.4a), keyword, and score (plus possibly other properties such as position information for phrase search), the enhanced inverted file keeps a predicate that logically specifies in which instances the keyword appears. These predicates are derived from the predicates specified in the \texttt{select} elements of the normalized view. The keyword “likes” appears in all instances, and it has associated the predicate \texttt{true}. All other keywords inherit the predicate of their surrounding \texttt{select} element. We mention that in the index the positional information of the keyword is also maintained, as it will be shown in Section 8.

Figure 7.2 shows the inverted file for the versions example in Figure 4.5. The keywords \textit{news} and \textit{is}

<table>
<thead>
<tr>
<th>Keyword</th>
<th>DocId</th>
<th>Predicate</th>
<th>Score</th>
<th>Dewey Pos.</th>
<th>Next Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mickey</td>
<td>$d_1$</td>
<td>$R3 = &quot;mouse&quot;$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Donald</td>
<td>$d_1$</td>
<td>$R3 = &quot;duck&quot;$</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>likes</td>
<td>$d_1$</td>
<td>\texttt{true}</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Minnie</td>
<td>$d_1$</td>
<td>$R3 = &quot;mouse&quot;$</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Daisy</td>
<td>$d_1$</td>
<td>$R3 = &quot;duck&quot;$</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7.1: Inverted File - Alternative (Figure 4.4).
Chapter 7. Enhanced Indexing

<table>
<thead>
<tr>
<th>Keyword</th>
<th>DocId</th>
<th>Predicate</th>
<th>Score</th>
<th>Dewey Pos</th>
<th>Next Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>good</td>
<td>$d_1$</td>
<td>$time = (\infty, 6am)$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>good</td>
<td>$d_1$</td>
<td>$time = [6am, +\infty)$</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>no</td>
<td>$d_1$</td>
<td>$time = [6am, +\infty)$</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>bad</td>
<td>$d_1$</td>
<td>$time = [6am, 7am)$</td>
<td>1</td>
<td>1.1</td>
<td>0</td>
</tr>
<tr>
<td>news</td>
<td>$d_1$</td>
<td>$time = (\infty, +\infty)$</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>news</td>
<td>$d_1$</td>
<td>$time = (\infty, +\infty)$</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>is</td>
<td>$d_1$</td>
<td>$time = (\infty, +\infty)$</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>rare</td>
<td>$d_1$</td>
<td>$time = (\infty, 6am)$</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7.2: Inverted File for Versions (Figure 4.5).

appear in all instances. As a result, the predicate is true for this keywords, equivalent to $(-\infty, +\infty)$, indicating there is no restriction on the instances. The keyword bad only occurs in certain instances, in particular identified by the predicate $time = [6am, 7am)$. A score can be computed (count in this example) for the keywords. Of course, this example only serves for illustration. Typically, inverted files index many documents. Furthermore, inverted files are partitioned into inverted lists (one per keyword) in order to speed up query processing.

7.2 Index Creation Algorithm

In order to create an inverted file for a set of normalized views, each normalized view is processed separately (as in traditional information retrieval). Algorithm 7.1.1 describes the two phases of the index creation process. Indexing is done while reading the Normalized View sequentially (lines 5-19).

Phase I - CreateIndexEntries($NV$): $I$ creates from the Normalized View $NV$ an entry $e$ in the inverted file $I$, for each occurrence of each keyword $k$ (possibly disregarding stop words and applying stemming as in traditional IR). The predicate for $e$ is set in the following way:

1. If the keyword is not inside a select element in $NV$, then the predicate of $e$ is simply true, maintained in the base of the predicate stack (line 3). Examples are likes in Figure 7.1 or news and is in Figure 7.2.

2. If the keyword is inside a select element in $NV$, then $e$ inherits the predicate of this select element. Examples are the two occurrences of no in Figure 7.2 (line 13).

3. If the keyword is inside nested select elements (as for the del element nested inside the ins element), then the predicate of $e$ is the conjunction of the predicates of the select elements of the normalized view. For example the conjunction between the predicates $time = [6am, +\infty)$ of the outer ins element and $time = (\infty, 7am)$ of the inner del elements results in the predicate $time = [6am, 7am)$ for the keyword bad. Initially the score of all entries is 1; it is adjusted when the entries are merged (lines 7-10).

Phase II - MergeIndexEntries($I$): For an index $I$ obtained as a result of Phase I, it is possible that several such entries involve the same keyword (or stem) and have equivalent or overlapping predicates. Entries are merged using the following strategies:
Algorithm 7.1.1 Index Construction Algorithm

1. Function createIndexEntries(Document NV) : I
2. Output: I \{ Inverted File \}
3. $\text{predicates} = \{\}$
4. $\text{predicates}.\text{push}(\text{true}) \{ \text{Stack contains the predicate of the topmost select element.} \}$
5. while not EOF(NV) do
6.    Read $w$: next keyword OR start markup of element $e$ OR end markup of element $e$
7.    if predicates.top() \( \not\equiv \) false then
8.        if $w$ is the start markup of select element $e$ then
9.            { Topmost predicate is the conjunction of all embedded predicates. }
10.           $\text{predicate} := \text{predicates}.\text{top()} \land \text{e}.\text{predicate}$
11.           $\text{predicates}.\text{push}(\text{predicate})$
12.        else if $w$ is a keyword then
13.            { Insert a new keyword posting, with top of the stack as predicate. }
14.            insert into INDEX(keyword, docId, predicate, score) VALUES($w$, docId, predicates.top(), 1)
15.        end if
16.    end if
17.    if $w$ is the end markup of select element $e$ then
18.        $\text{predicates}.\text{pop()} \{ \text{Go one level up.} \}$
19.    end if
20. end while
21. End Function

22. Function mergeIndexEntries(I) : I \{ Merging entries with compatible predicates \}
23. Output: I \{ Merged Inverted File \}
24. for all $e_1 \in \text{INDEX} \land e_2 \in \text{INDEX}$ do
25.    if (($e_1\text{.predicate AND } e_2\text{.predicate}) \not\equiv \text{false}) \land (e_1\text{.keyword }= e_2\text{.keyword}) \text{ then}$
26.        $\text{entry}_1 = (e_1\text{.keyword}, \text{NOT } e_1\text{.predicate AND } e_2\text{.predicate}, e_1\text{.score}, e_1\text{.pos})$
27.        $\text{entry}_2 = (e_2\text{.keyword}, e_1\text{.predicate AND } e_2\text{.predicate}, \text{AGGR}(e_1\text{.score}, e_2\text{.score}), e_1\text{.pos})$
28.        $\text{entry}_3 = (e_1\text{.keyword}, e_1\text{.predicate AND NOT } e_2\text{.predicate}, e_1\text{.score}, e_1\text{.pos})$
29.        $\text{DELETE } e_1, e_2 \text{ FROM INDEX}$
30.        INSERT INTO INDEX $\text{entry}_1, \text{entry}_2, \text{entry}_3$
31.    end if
32. end for
33. End Function

1. If two entries for the same keyword have equivalent predicates, then these two entries can be merged into one entry; the score of the merged entry is adjusted accordingly, using the search engines specific scoring functions (for simplicity, we use the term frequency as score, but other metrics could also be used [16]).

2. If the predicates of two entries, say $p_1$ and $p_2$ overlap, then three entries are generated: one entry for $p_1 \land p_2$, one entry for $p_1 \land \neg p_2$, and one entry for $\neg p_1 \land p_2$ (with scores assigned accordingly). If $p_1$ implies $p_2$, then $p_1 \land \neg p_2$ is always false and this entry can be discarded so that only two entries are generated. Furthermore, $p_1 \land p_2$ is equivalent to $p_1$ if $p_1$ implies $p_2$. Since the predicates in the index entries are always conjunctions and negations of simple variable/value comparisons, the equivalence, implication, and overlap of predicates can easily be computed.
7.3 Complexity Analysis

We consider $N$ to be the size of the data (eventually including the referred content, as mentioned in the Normalization Section 6.3).

**Complexity in terms of size.** If $N$ is the size of the normalized data, the size of the generated predicate based index is $O(N)$. The predicates add an overhead over traditional inverted files which reflects in a high constant of the notation. If the size of the normalized view is in the same order as the size of application data, then indexing is also in the size of the application data.

**Generation time.** The index is also generated in $O(N)$, directly from the normalized view. The higher constants apply again here.
Chapter 8

Enhanced Positioning

Positioning is an important part of Information Retrieval systems, used for processing phrase queries. A flat positioning scheme cannot cope with the fact that some parts of the content may not appear in some instances (e.g., the position of a term is variable across instances). As part of Indexing, we introduce Dewey ID with Level, a positioning scheme helpful in simplifying the problem of computing adjacency for positions which are apparently not consecutive. The advantage of this position is that it can be directly applied to the normalized view, is adapted for taking into account the instance information, and uses an easy calculus for computing adjacency.

8.1 Example

Figure 8.1 shows how Dewey IDs are associated to the versioning example of Figure 4.5. Dewey IDs [106] are hierarchical IDs which have been used recently in XML databases to model element IDs. In our case, Dewey ID Positions are assigned sequentially to keywords, as in traditional information retrieval, but also to “select” elements. Each level of the Dewey ID indicates a nesting in the structure of the “select” elements. The additional component of the IDs of a keyword or “select” element, the level, indicates the nesting level of the next consecutive keyword or “select” element, whichever comes first. As a consequence, each select element leaves a gap of one unit position within its surrounding elements.

8.2 Definitions

We formalize Dewey IDs and how adjacency information can be computed. We define the semantics of the = operation and of the increment (+1) operation on Dewey IDs with level.

Definition 1: Level. We call Level the nesting level of annotated content. The root of the document has nesting level zero, comments have nesting level one, while del elements nested inside ins elements have nesting level two. We will assign to each keyword and “select” element in the Normalized Document a hierarchical position with as many levels as its nesting level. In order to compute adjacency correctly, we will also mark in the position information the level of the next inline element or keyword, whichever appears first.
Definition 2: Dewey ID with level. We call Dewey ID with level of an inline element or keyword \( t \), a tuple \((p_0.p_1,\ldots,p_n,1)\) writing \((p_0.p_1,\ldots,p_n)\) where \( n \) is the nesting level of \( t \), \( p_n \) is the position of \( t \) inside its inline element or root of the document, whichever may be the case, and \( l \) is the level of the next consecutive inline element or keyword, whichever comes first. The value of \( l \) for a Dewey ID of size \( n + 1 \) is considered implicitly as being \( n \) (i.e. the index of the last component, starting from 0, assuming that most of times the next keyword is at the same nesting level).

Definition 3: Increment (+1) of Dewey ID with level. The increment of a given Dewey ID with level position \((p_0.p_1,\ldots,p_n)/l\), written \((p_0.p_1,\ldots,p_n)/l + 1\), is defined as follows:
\[
(p_0.p_1,\ldots,p_n)/l + 1 = (p_0.p_1,\ldots,(p_n + 1)/l).
\]
When the level is not given, and is implicitly \( n \), the following applies:
\[
(p_0.p_1,\ldots,p_n) + 1 = (p_0.p_1,\ldots,(p_n + 1)/n).
\]
This allows to compute the next position of any keyword in the document. For example, for elements on the same level, just increase the last component. (e.g., \( 3.1 + 1 = 3.2 \)). The next position for elements on different levels (e.g. \( 3.3/0 \)) can be determined by using the indication of the level information (i.e., 0), and increasing the corresponding component of the path, and zeroing the rest. \( 3.3/0 + 1 = 4/0 \).

The following properties are used in order to compute adjacency of Dewey IDs with level.

Definition 4: Equality of Dewey IDs with level (=). Two Dewey IDs with level \((p_0.p_1,\ldots,p_n)/l\) and \((q_0.q_1,\ldots,q_m/p)\) are equal, writing \((p_0.p_1,\ldots,p_n)/l = (q_0.q_1,\ldots,q_m/p)\) iff the following condition is fulfilled:
\[
\exists t, 0 \leq t \leq \min(n, m) : p_i = q_i, \forall i \leq t \text{ and } p_i = 0, \forall i > t \text{ and } q_j = 0, \forall j > t
\]
This definition says that any Dewey IDs are equal if they share a common prefix, and the rest of the entries are zero (ignorable). This property can be used to compute the adjacency \( 2 + 1 = 3.0 \) which is equivalent to \( 3 = 3.0 \) (equal, despite two different nesting levels). As a consequence, the Dewey IDs and Level do not uniquely identify a keyword occurrence in the document.

Definition 5: Adjacency of two Dewey IDs with Level. Two Dewey IDs with level: \((p_0.p_1,\ldots,p_n)/l\) and \((q_0.q_1,\ldots,q_m/p)\) are adjacent iff:
\[
(p_0.p_1,\ldots,p_n)/l + 1 = (q_0.q_1,\ldots,q_m/p)
\]
The definitions justify the role of the level information along with the Dewey IDs. The following example illustrates this: without level information, the next position of \( 5.0 \) would be erroneously computed as \( 5.1 \).
instead of 6. This could cause errors during query processing. For example, during the processing of
the phrase query “good B”, the candidate pair of positions {5.0, 6} for “good” and “news” is retrieved;
without level information this entry would be discarded, although they are physically consecutive in the
instance corresponding to the last version (I2). When the level of the next keyword is maintained (i.e., 0),
the adjacency test succeeds since the increment works on the first component of the ID.

8.3 Assigning Dewey ID Positions: Algorithm.

Dewey IDs with levels are assigned sequentially to each keyword or “select” element encountered in the
document. Algorithm 8.3.1 computes Dewey IDs with level. Position computation is part of indexing,
therefore these steps can be performed at the same time as those of algorithm 7.1.1. Since the level
component of one position indicates the level of the next “select” element or keyword in the document,
a position is assigned only when the next keyword or “select” element has been encountered (lines 12, 27
and 47). When the indexer reads the end of a “select” element, the level is decreased, and the Dewey
ID is shrunk (the last component of the Dewey ID is removed) (lines 37-41). Only then can one assign
the position and level to the previously encountered keyword. As a consequence of the Dewey creation
algorithm, all keywords embedded in a “select” element are sequential (lines 17 and 32) and have the
same prefix. Furthermore, there is a distance of exactly one between content surrounding one “select”
element.

8.4 Optimizing Storage of Dewey IDs with Level.

Dewey IDs with level introduce an overhead over traditional sequential positioning, in terms of size and
required computation. Based on the definitions of Dewey ID with level, there are ways to reduce the size
of the Dewey IDs and level: reduce the length of Dewey IDs, and eliminate the need to store the level
information.

1. Do not store trailing zeroes for Dewey IDs. Based on Definition 2, any trailing zeroes are not
important when computing equality. However, it is important to correctly evaluate the level of the current
position despite having removed trailing zeroes, when the next position is computed. Therefore the level
information must be maintained in this case.

2. Only store the level information for the elements which are followed by a keyword with a
smaller position level. By default, unless trailing zeroes are removed, no level information is needed.
The assumption is that the next consecutive position (for example for 2) is on the same level and has
the value 3. It is also clear that level information is required when the positioning level changes from a
bigger to a smaller depth, (as in the example of 5.0 and 6). What is left is to demonstrate is that level
information is not necessary for passing from a lower depth to a higher depth. An example is it 3 and
4.0. This is due to the fact that passing from the position \((p_0, p_1, ..., p_n)\) at level \(n\) to a higher level occurs
because an inlined element was encountered immediately following this position. Also, the increment
in level can only be one. Therefore: (i) the first consecutive position \(p_0, p_1, ..., (p_n + 1)\) is assigned to the
inlined element as a whole and (ii) the position \((p_0, p_1, ..., (p_n + 1), 0)\) is assigned to the first keyword inside
the inlined element (a level higher). Based on the definition, both following equations:
\[(p_0, p_1, ..., (p_n + 1)) = (p_0, p_1, ..., (p_n + 1)\) and \((p_0, p_1, ..., (p_n)) + 1 = (p_0, p_1, ..., (p_n + 1), 0)\) hold and do not need the require to store a
Algorithm 8.3.1 Compute Dewey IDs With Level

1: Input: Document NV \{ Normalized Document \}
2: Output: positions[] \{ Positions of each keyword or "select" element \}
3: \( S = \{ \} \) \{ Set of inlined elements whose next level is not yet known. \}
4: \( pd = -1 \) \{ Previous Dewey ID. \}
5: \( d = -1 \) \{ Current Dewey ID. \}
6: \( l = 0 \) \{ Current level. \}
7: \( lastKeyword = null \)
8: while not EOF do
9: \hspace{1em} Read \( w \): next keyword or start markup of "select" element \( e \) or end markup of "select" element \( e \)
10: \hspace{2em} if \( w \) is the start markup of "select" element \( e \) then
11: \hspace{3em} if \( lastKeyword \neq null \) then
12: \hspace{4em} position(lastKeyword) = \((pd/l)\) \{ Assign position and level of previous keyword or element. \}
13: \hspace{3em} end if
14: \hspace{2em} for all \( el \in S \) do
15: \hspace{3em} position(el).level = \( l \)
16: \hspace{2em} end for
17: \hspace{2em} \( d = d + 1 \)
18: \hspace{2em} \( d = d.\text{deweyID} = d \)
19: \hspace{2em} \( l = l + 1 \)
20: \hspace{2em} \( pd = d \)
21: \hspace{2em} \( S = \{ \} \)
22: \hspace{2em} \( lastKeyword = null \)
23: \hspace{2em} end if
24: \hspace{2em} if \( w \) is a keyword then
25: \hspace{3em} if \( lastKeyword \neq null \) then
26: \hspace{4em} position(lastKeyword) = \((pd/l)\) \{ Assign position and level of previous keyword or "select" element. \}
27: \hspace{3em} end if
28: \hspace{3em} for all \( el \in S \) do
29: \hspace{4em} position(el).level = \( l \)
30: \hspace{3em} end for
31: \hspace{3em} \( d = d + 1 \)
32: \hspace{3em} \( pd = d \)
33: \hspace{3em} \( S = \{ \} \)
34: \hspace{3em} \( lastKeyword = w \)
35: \hspace{3em} end if
36: \hspace{2em} end if
37: \hspace{2em} if \( w \) is the end markup of "select" element \( e \) then
38: \hspace{3em} \( d = d.\text{parent} \) \{ Go one level up. \}
39: \hspace{3em} \( l = l + 1 \)
40: \hspace{3em} \( S = S \cup \{e\} \) \{ Store element until next level is known. \}
41: \hspace{3em} end if
42: \hspace{2em} end while
43: \hspace{2em} for all \( el \in S \) do
44: \hspace{3em} position(el).level = \( l \)
45: \hspace{2em} end for
46: \hspace{2em} let position(lastKeyword) = \((pd/l)\)

level for \( p_0.p_1.\cdots.p_n \).

3. Do not store level information and trailing zeroes for keywords which are single words in an inlined content. The idea is easily illustrated by the keyword "good" at position 5.0/0. Since it is alone in the inlined content, it is clear that the only position which can follow is 6.0. The same effect can be achieved if both the trailing zeroes and the level information are not stored in the first place.
Chapter 9

Skip Spaces

In order to allow the index to also answer phrase queries (as described in Section 10), we introduce a new entity in the index, called Skip Space. A Skip Space is a special keyword introduced at indexing time, and marks the fact that some elements do not appear in all instances. A Skip Space acts as a placeholder for an entire “select” element, in the instances where the element is skipped. We formalize the concept of Skip Space by using the following definition.

**Definition 6: (Skip space).** A Skip Space written “ ” (empty space) is a special keyword whose occurrences are created at indexing time. A Skip Space has the following properties:

(i) An instance(occurrence) of a Skip Space is created for each element “select” node $e$ in the Normalized Document. Let $e\.pred$ be the predicate characterizing this element (as in Section 6), and $e\.pos$ the unique position of this element (as shown in Section 8).

(ii) At least one index entry, called Skip Entry is created for each Skip Space entry in the predicate-based index, defined as follows.

**Definition 7: (Skip Entry).** A Skip Entry associated with an occurrence of a “select” element $e$, whose predicate is $e\.pred$ and position $e\.pos$, is a tuple $skip\.entry=(docId, pred, score, position)$ whose components are defined as follows:

(i) $skip\.entry\.position = e\.pos$ (i.e., a Skip Space has the same position of the element it replaces.)

(ii) $skip\.entry\.pred = \neg e\.pred$: (i.e., a Skip Space is only associated to the instances where the “select” element does not appear.)

(ii') if at step (ii), $skip\.entry\.pred$ is in DNF (Disjunctive Normal Form) and then add a Skip Entry for each term (CNF) of the DNF.

(iii) $skip\.entry\.score = 1$

This definition specifies that a Skip Entry has an associated complement predicate to the predicate of the element it replaces, possibly generating more entries with disjunct predicates.
Chapter 9. Skip Spaces

The Skip Spaces are added directly in the indexing phase. This is reflected by adding an additional step in the index creation algorithm:

**AddSkipEntries(I):** For each select element in the Normalized Views, with predicate pred, compute the predicate ¬pred (with the restrictions in the definition of Skip Entries) and introduce a Skip Entry:

\[
\text{skip entry} = (\text{docId}, \neg \text{pred}, \text{position})
\]

To the index. If necessary, a MergelIndexEntries operation needs to be performed.

In case of Versioning, a skip entry is created for each select element surrounding the ins and del elements. We show in Figure 9.2 how the Skip Spaces introduced for versioning are determined. As mentioned before, Skip Spaces inherit the position of the element they replace, but the predicates corresponding to their original elements must be negated. For example the nested del element, with predicate \((6h, 7h]\), has two associated entries, with predicates \((-\infty, 6h]\) and \((7h, +\infty)\). Physically, Skip Spaces are stored in the same inverted file as the entries for the actual keywords in the document.

### 9.2 Benefits of Enhanced Indexing

The enhanced index for application views, as described in this section, brings the benefits described below.

**Compactness.** If correct query results are to be produced, for each keyword in the inverted file, we must attach information about in which instances that keyword appears. The merging phase of index construction expands the set of instances covered by each individual predicate. Thus, the necessary number of entries in the index is minimized.
Ease of Implementation. An enhanced index extends a traditional inverted file by associating predicates to keyword entries. Therefore, existing search engine implementations may be easily extended to include in the generated index the additional information. As we will see in Section 10, query processing on the enhanced index may also be implemented with reasonable additional effort.

Fuzziness. The procedures described in the previous section build a space-efficient index over all instances of the input application views. One approach to reduce the number of entries in an enhanced index is to fuzzify the index; for example, some entries could simply be discarded. This fuzzification might result in false positives and false negatives, thereby trading index accuracy for index size. Since the performance experiments reported in Section 11 show that even a precise index scales well and has a tolerable size, we did not study this fuzzification in more detail. It is, however, an interesting avenue for future research.
Chapter 10

Enhanced Query Processing

As in traditional information retrieval, this work focuses on queries that are composed of keywords. Also, these queries can be processed using an extended inverted file, in almost the same way as in any traditional information retrieval system [16]. However, in this case a user is interested in all instances (as opposed to all documents) containing the given keywords. This operation results in merging one or more inverted lists containing the entries of the given keyword queries. This section presents the main components of query processing: (i) Processing Keyword Queries (Conjunctions), (ii) Processing Phrase Queries and (iii) Ranking.

10.1 Processing Keyword Queries

A Keyword Query consists of a list of keywords. The result of a keyword queries is a list of Instances which contain all keywords of the query.

10.1.1 Example

As an example, a user might be interested in all instances that contain the keywords good and bad, from the versioned document whose inverted file is represented in Figure 4.5. First, the (extended) inverted file is queried in order to find all entries for the keyword good. Using the inverted file of Figure 7.2, the result is \( \langle d_1, 1, time = (-\infty, 6), (0.0/0) \rangle, \langle d_1, 1, time = [6, +\infty), (5.0/1) \rangle \) (the keyword can be omitted at this point). Likewise, the inverted file is queried in order to find all entries for the keyword bad. The result is \( \langle d_1, 1, time = [6,7), (1.1/0) \rangle \). In the next step, these two lists are merged. In traditional information retrieval, the merge for conjunctive queries implements a join (or intersection), thereby trying to find matching documents from both lists. For the extended inverted files (e.g., Figure 7.2), this logic must be extended in order to process the predicates that specify the relevant instances. Therefore, in addition to a join on the document identifiers, a conjunction of the predicates must be carried out. As a result, the following list is constructed for the query good bad:

\[
\langle d_1, 1, time = (-\infty, 6) \land [6,7)\rangle, \{0.0/0), (1.1/0)\}\n\langle d_1, 2, time = [6, +\infty) \land [6,7)\rangle, \{(5.0/1), (1.1/0)\}\n\]

and the final result, after eliminating the predicates which result in false is: \( \langle d_1, 1, time = [6,7)\rangle, \{(5.0/1)\}, \{\text{false}\} \).
Chapter 10. Enhanced Query Processing

(1.1/0).

From this list and the normalized document, the result instances can be generated and they will be sorted according to their score. Different scores can be computed, as illustrated in detail further in Section 10.4. If more keywords are used, a left-deep tree is constructed, as shown in Figure 10.1. We formalize the merge operation on predicate-based indexes as follows:

**Definition 8:** (MERGE Step). Let $w_1$ and $w_2$ be two keywords and $w_1, i = 1..n$, $w_2, j = 1..m$ their inverted lists. Each entry $w_k$ has the form $w_k = (docId, predicate, \{([positions], ...\})$. $\{([Positions], ...)\}$ is a concatenation of one or more position lists, each entry corresponding to the positions of a given keyword. The merge step of two index entries $w_{1i}$ and $w_{2j}$ is a tuple defined as follows:

$$MERGE(w_{1i}, w_{2j}) = \begin{cases} (doc, score, w_{1i}.pred \cap w_{2j}.pred, \text{concat}(w_{1i}.pos, w_{2j}.pos)) & \text{if } doc_{1} = \text{doc}_{2} \land w_{1i}.pred \cap w_{2j}.pred \neq \emptyset, \\ \emptyset, & \text{otherwise} \end{cases}$$

Figure 10.2 is a more complex example involving two rules so that the predicates involve two variables (R1 and R2). For each document, the same set of variables is used in the predicates of all inverted lists because the same rules were applied to the document. In Figure 10.2, for example, instances of Document $d_1$ are specified by constraining the values of Variables R1 and R2. The first inverted list (List 1) has three entries for Document $d_1$; the second list (List 2) has two entries for $d_1$. Scoring and positioning information have been omitted for clarity. Logically, all six combinations of entries of the first list and entries of the second list are considered. Some of these combinations result in false; e.g., $R1=2 \land R2>3 \land R2=3$ is false. In an inverted list, a predicate with false indicates the empty set of instances, so such entries can be evicted. As a result, only four entries are part of the result of this merge operation.

Disjunctions can be processed in the same way as conjunctions, by using a left-deep tree. As an optimization, the predicates of the entries in the index contain only conjunctions, operation ensured by the `MergeIndexEntries (I)` described in Section 7.

### 10.1.2 Merge Algorithm

There are more alternatives for implementing a merge join of inverted lists. Currently, the `MERGE` operator in the left-deep tree is implemented using a sweep-based algorithm (spatial join) [97]. It is implemented as such since possibly more variables occur in the same document, as shown in the more complex example of Figure 10.2. The goal is to intersect interval predicates, but process the input sequentially and only once, the metaphor of “Sweep area” [51] can be used. In joining intervals, the largest interval in the two lists is kept in the join area until all other included intervals have joined with it and the intersection of intervals is output. A similar algorithm is reflected in the stack-based or structural

![Figure 10.1: Left deep tree for joining three keywords](image)
10.2 Complexity Analysis.

The merge operation requires its input to be sorted. Because of this, specific requirements affect the indexing operation, i.e., entries in the index are stored sorted on the document ID [28], then for each keyword in that document, the entries are sorted based on the least selective variable for that document. This ensures that the largest number of joins can be determined by maintaining one side of the merge stable. Further details on the generic sweep-line algorithm can be found in [97, 15, 51].

In all experiments that we conducted so far, the merge algorithm used was efficient enough to meet the performance requirements of modern information retrieval (Section 11). There are, however, several additional optimizations that can be applied. First, if only points (rather than intervals) are applied, then a traditional sort-merge join can be applied. This opportunity arises if no Version rules have been applied to the relevant documents. Second, in some situations, further multi-dimensional join techniques (e.g., [50]) can be applied. Third, more clever, cost-based optimization techniques can be applied, thereby reordering the sequence in which the inverted lists are merged. Studying such optimization techniques is beyond the scope of this work; essentially, the same techniques are applicable as in traditional information retrieval systems. We plan to study the trade-offs of the different join techniques as part of future work.

10.2 Complexity Analysis.

The merge algorithm of more inverted lists is polynomial in the number of merged lists. However, two observations are to be made: the sweep-line algorithm can perform the computation of \( k \) intersections of \( N \) intervals on the sweep-place in \( O(kN) \) with a \( O(N) \) space requirement, which is the complexity that our current implementation has. Although not implemented yet, the complexity of the sweep-line algorithm can be implemented in the order of complexity \( O((k + N)\log N) \) and \( O(N) \) in terms of space.

10.3 Processing Phrase Queries

This section describes the implementation of phrase search using Skip Spaces and Dewey IDs. We show in an example how processing phrase queries takes place with the help of the new identifiers, and then formalize it. Phrase search is performed in two steps: (i) Computing Keyword Correlations. and (ii) Intercalating Skip Spaces.

(ii) Computing Keyword Correlations. The first step of query processing consists of producing the predicates which match all keywords in the phrase (e.g., identifying the instances where all query keywords appear together). We exemplify Phrase Search by using the Versioned Data example of Figure 4.5. First, as shown in Figure 10.3, basic keyword search is performed in order to compute the keyword
correlations.

The results of the basic keyword search are entries with candidate position lists. In Figure 10.3, none of the candidate lists are sequential. We can compute this by testing if the positions in each pair of results are adjacent. The first two (1/1 and 2/0) are not adjacent since 1.0/1 + 1 = 1.1 ≠ 2.0). The same can be said about (1/1, 6/0) since 1/1 + 1 = 1.2 ≠ 6/0. A second step is therefore needed. Intercalating a "Skip Space" solves the problem. We perform the operation in two steps, as shown in Figure 10.4. The final result is the last version, V2, with the positions (1/1, 1.1/0, 2.0). They are sequential because 1/1 + 1 = 1.1 and 1.1/0 + 1 = 2.0 We formalize the second step of the computation.

(ii) Intercalating Skip Spaces. Based on the example, we can actually say that we express phrase search as follows:

"w1 w2" = "w1(" "") + w2"

The semantics of this formula are: try to intercalate none, one or more Skip Spaces between entries of "w1" and "w2" in order to decide whether they are adjacent. Two occurrences of these keywords may be consecutive, but if not, they might be separated by one or more Skip Spaces. This must be checked for any pair of positions "(p1, p2)" which are not consecutive. If a skip space occurs in a certain instance between the two positions, than the positions are adjacent despite their values being at a distance higher than one. If no Skip Space occurs, then the positions are not adjacent.

This can be formally computed by defining the operation on nodes in the Query Plan. The following definition for SKIP.MERGE (Figure 10.5) makes use of the MERGE(w1, w2) operator defined in the previous section.

Definition 9: (SKIP.MERGE). Let "w1 w2" be a phrase query. w1 and w2 are keywords and let w1i, i = 1..n, w2j, j = 1..m be their inverted lists. Let skipk, k = 1..p be the inverted list of the Skip Spaces in the index. Each entry wk has the form wk = (docId, predicate, positions). Positions is a candidate list of one or more sequential positions. The entries read from the index have a single position in this list. The Skip Merge step of two index entries w1i and w2j is a tuple of the same form, defined as follows:

\[
\text{SKIP.MERGE}(w_{1i}, w_{2j}) = \begin{cases} 
\text{MERGE}(w_{1i}, w_{2j}) & \text{if } \text{last}(w_{1i}, \text{pos}) + 1 = w_{2j}, \text{pos}, \\
\text{SKIP.MERGE}(\text{skip}_{k}, w_{2j}) & \text{if } \exists k : \text{last}(w_{1i}, \text{pos}) + 1 = \text{skip}_{k}, \text{pos}, \\
\emptyset & \text{otherwise}
\end{cases}
\]

This definition is based on the fact that in order to match the phrase "w1 w2" the entries which compose a match can be concatenated starting with the entries of w1 until an adjacent entry of w2 is found. In the process, intermediate "Skip Spaces" may be used in the concatenation process before reaching an entry of "w2" is reached. In more detail, the definition above attempts first to decide whether the current entries from the two inverted lists are directly adjacent (first case), then return the merged entry MERGE(w1i, w2j) (as defined by Definition 1). If the two are not adjacent, an adjacent Skip Space is searched, and if found, no tuple is yet returned; instead the concatenation is continued from the currently found Skip

![Figure 10.3: Basic Keyword Search in Versioned Data returns no consecutive positions](image-url)
10.4. Ranking

A ranking scheme was introduced in order to support ranking of results in the predicate-indexing-framework. As opposed to ranking documents in a traditional search, ranking must be able to address instances, as opposed to just documents.

### 10.4.1 Analysis of a query result

Each result is expressed as a `<doc, predicate, scores, positions>` pair, where `predicate` can have multiple variables. In the following examples, for computing ranking, we consider that a dictionary-based-compression has already been applied. For example, the search result `⟨d, $R_3 = [1, 5]⟩` may correspond to versioned data spanning 5 consecutive time moments, independently of the actual time values.

Each result expresses either one or more instances. Generally, in a document `d` with `n` variables `$v_i \in [v_{i1}, v_{i2}]`, i = 1, n`, the result with `m` variables: `R = ⟨d, v_j = [v_{j1}, v_{j2}]⟩ (j = 1, m)` expresses a total number of instances:

\[
N_{inst}(res) = \prod_{j=1, j \neq m} (v_{j2} - v_{j1} + 1) \times \prod_{i=1, n} (v_{i2} - v_{i1} + 1) [v_i \not\in R].
\]

For example, in a document `d` with two variables `$1$ and `$2$ with the domains $1 \in [1, 5] \land 2 \in [3, 4]$, the total number of instances is ten ($10 = (5-1+1) \times (4-3+1)$), and the predicate `$1 \in [4, 5]$ expresses 4 instances ($4 = (5-4+1) \times (4-3+1)$).
10.4.2 Ranking functions

We consider two possible ranking functions based on the \( tf \times idf \) pattern from traditional information retrieval. The first one is document-centric and the second one is instance-centric, because of the two components expressed in a single result.

1. **Instance-based.** In this case, we rank the results based on the relevance of the keywords among the set of documents of the collection. The ranking function is:

\[
\text{rank}(doc, res_{doc}) = \sum_{t \in \text{query}} tf(t, res_{doc}) \times iif(t, res_{doc}) \times idf(t), \text{ where:}
\]

- \( tf(t, res_{doc}) = \text{term frequency of term } t \text{ in a query result} = res.score(t) \times N_{inst}(res) \)
- \( iif(t, res_{doc}) = \ln(1 + |N_{inst}(t, doc)| / |N_{inst}(doc)|) \)
- \( idf(t) = \ln(1 + |N_{docs}(t)| / |N_{docs}|) \)

(this formula computes relative rankings)

2. **Document-based.** In this case, we rank the results based on the relevance of the result among the set of instances of the collection. The ranking function is:

\[
\text{rank}(doc, res_{doc}) = \sum_{t \in \text{query}} tf(t, res_{doc}) \times iif(t, res_{doc}) \times idf(t), \text{ where:}
\]

- \( tf(t, res_{doc}) = \text{term frequency of term } t \text{ in a query result} = res.score(t) \times N_{inst}(res) \)
- \( iif(t, res_{doc}) = \ln(1 + |N_{inst}(t, doc)| / |N_{inst}(doc)|) \)
- \( idf(t) = \ln(1 + |N_{docs}(t)| / |N_{docs}|) \)

(this formula computes relative rankings, and differs by the \( idf(t) \) component.)
Chapter 11

Experiments

We implemented the techniques presented in the previous sections (normalization, indexing, and extended query processing) and experimentally compared our implementation with the traditional search, naïve search and a commercial search engine. We call our approach “Enhanced” (with rules), the baseline inverted index “Traditional” (no rules apply) and the approach which materializes the instances for indexing “Naïve”. The goal of the experiments was to show that (a) our extended approach gives better search results than traditional search (b) space and processing overhead over traditional search is clearly smaller than the overhead of naïve approach and (c) we are comparable to a commercial search engine (Windows Desktop Search (WDS), which provided the best customization possibilities).

Experimental Environment. The indexing engine (i.e., crawling, normalization and index creation) and query processor were implemented in Visual Studio.NET using the Microsoft .NET Framework 2.0. All experiments were performed on an IBM ThinkPad T42, with a Pentium-M 1.7GHz processor and 1GB of RAM under Windows XP Professional.

Experimental Data. For the experiments reported in this chapter, we crawled data collections which exhibit the patterns in Table 11.1. Data sizes appear in Table 11.2.

Queries on each collection return False Positives or False Negatives, depending on whether traditional search sees the instances correctly or not: as shown in Table 11.1, searching versions (Twiki, CVS, Wikibooks) produces False Positives (FPs), if no version contained the searched keyword, but the document as a whole does. On the other side, for \LaTeX, E-mail and Java code traditional search produces False Negatives (FN). \LaTeX files contain Annotations and external includes, modeled using Placeholders. Outlook produces a single file containing many E-mails, which through the application can be seen either as single messages - traditional search - or grouped by conversation thread using the Alternative rule. FNs are produces since traditional search does not see the text of a conversation as a whole. Java code expresses inheritance relationships modeled using Placeholder. FNs are produced since subclasses are not returned when searching for content inherited from superclasses.

The characteristics of the data collections are specified in Table 11.2. The data varies from 5.54 MB in case of TWiki to 182 MB in case of Wikibooks and contains several levels of nesting of the elements carrying patterns. Versioned data but also messages and conversations contains many overlappings which will negatively influence the quality of the Naïve approach.

Queries. Queries were chosen to be meaningful with respect to the data collection, and contain from two to five keywords.
Table 11.1: Problems with the Accuracy of State-of-the-art Desktop Search

<table>
<thead>
<tr>
<th>Data Collection</th>
<th>Patterns</th>
<th>Description</th>
<th>Trad.Search Produces False Positives</th>
<th>Trad.Search Produces False Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>\LaTeX{}</td>
<td>Annotation, Placeholder</td>
<td>Annotated footnotes, External includes</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>E-mail (no attach.)</td>
<td>Alternative (two rules)</td>
<td>Group by message and by conversation thread</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Java code</td>
<td>Placeholder, Annotation</td>
<td>Modeling inheritance</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Twiki</td>
<td>Version</td>
<td>Versioned documents</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>CVS</td>
<td>Version</td>
<td>Versioned software repos.</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Wikibooks</td>
<td>Version</td>
<td>Versioned information</td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>

Table 11.2: Experimental Data Collections

<table>
<thead>
<tr>
<th>Data Collection</th>
<th>Size (MB)</th>
<th>No of Files</th>
<th>No.of Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>\LaTeX{}</td>
<td>13.2</td>
<td>1093</td>
<td>1473</td>
</tr>
<tr>
<td>E-Mail</td>
<td>45.9</td>
<td>1</td>
<td>26692</td>
</tr>
<tr>
<td>Java code</td>
<td>182</td>
<td>6624</td>
<td>8005</td>
</tr>
<tr>
<td>Twiki</td>
<td>4.54</td>
<td>410</td>
<td>3743</td>
</tr>
<tr>
<td>CVS</td>
<td>5.92</td>
<td>827</td>
<td>1442</td>
</tr>
<tr>
<td>Wikibooks</td>
<td>79.7</td>
<td>9846</td>
<td>72576</td>
</tr>
</tbody>
</table>

11.1 Precision and Recall

The main goal of modeling application data correctly in a search engine was to increase the result quality. We measure Precision and Recall for the above queries and compare Traditional (i.e., no knowledge of the rules), Enhanced and Na"ıve Approaches (both with full knowledge of the model) as shown in Figure 11.2. One way to compute Precision and Recall of Traditional search is to consider how many relevant Documents are returned when compared to the Enhanced and Na"ıve methods, as shown in Figure 11.2a and Figure 11.2b. This measure describes better the collections with False Negatives (\LaTeX{}, E-mail, Java code) and shows just a 50% Recall for Traditional search in case of Java code. This is caused by the fact that traditional search ignores inheritance semantics, which is very frequent in the code. In case of not so frequent Placeholder patterns (e.g., \LaTeX{}), the Recall of Traditional Search is 80 %, still 20% worse than that of the Enhanced Approach.

As shown in Figures 11.2c and Figure 11.2d, we also compute Precision and Recall by considering how many relevant *instances* are contained in the result if the whole document is returned, as opposed to how many are supposed to be returned. This measure describes better the collections with False Positives (i.e., the versioned collections: Twiki, CVS, Wikibooks). In case of Versioned documents, Trad. generally returns the entire document, i.e., all versions, and causes just a Precision between 40% and 50% in the best case, since most of the returned versions are not relevant.

11.2 Index Size

As shown in Tables 11.3 and 11.4, we created indexes for the data set and measured the index size using the traditional approach (no rules, but with imprecise query results), the enhanced approach, which uses
11.2. Index Size

<table>
<thead>
<tr>
<th>Queries Latex</th>
<th>Queries E-Mail</th>
<th>Queries JavaCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 query data shipping</td>
<td>Q1 meeting sigmod</td>
<td>Q1 public hashcode</td>
</tr>
<tr>
<td>Q2 query shipping</td>
<td>Q2 kossmann meeting</td>
<td>Q2 main class thread run</td>
</tr>
<tr>
<td>Q3 database settings</td>
<td>Q3 important message</td>
<td>Q3 abstract query</td>
</tr>
<tr>
<td>Q4 sans serif typeface</td>
<td>Q4 delis vorschlaege cnr pisa</td>
<td>Q4 interface equals</td>
</tr>
<tr>
<td>Q5 data</td>
<td>Q5 ecoll healthcare phone conf</td>
<td>Q5 abstract gelstais</td>
</tr>
<tr>
<td>Q6 query</td>
<td>Q6 sigmoid conference</td>
<td>Q6 public static void main</td>
</tr>
<tr>
<td>Q7 data and query shipping</td>
<td>Q7 reviewer feedback</td>
<td>Q7 getsource main</td>
</tr>
<tr>
<td>Q8 query data structure</td>
<td>Q8 important message</td>
<td>Q8 interface class protected public</td>
</tr>
<tr>
<td>Q9 database structure</td>
<td>Q9 phone conference</td>
<td>Q9 int close abstract</td>
</tr>
<tr>
<td>Q10 setting the structure</td>
<td>Q10 meeting conference</td>
<td>Q10 public private protected getstore</td>
</tr>
</tbody>
</table>

**Figure 11.1: Experimental Queries.**

**Figure 11.2: Precision and Recall for Experimental Data Sets**
rules and gives correct results, and the naïve approach (rules, correct results, but using materialized instances).

<table>
<thead>
<tr>
<th>Collection</th>
<th>Trad Index</th>
<th>Enh Index</th>
<th>Naïve Index</th>
<th>Overhead</th>
<th>Space Gain Enh/Naïve</th>
<th>Ix.Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>l³t²X</td>
<td>2.19</td>
<td>2.81</td>
<td>3.7</td>
<td>x1.28</td>
<td>24%</td>
<td>19.1</td>
</tr>
<tr>
<td>E-mail</td>
<td>21.1</td>
<td>33.4</td>
<td>50.4</td>
<td>x1.56</td>
<td>34%</td>
<td>484</td>
</tr>
<tr>
<td>Java Code</td>
<td>6.6</td>
<td>7.08</td>
<td>8.16</td>
<td>x1.07</td>
<td>13%</td>
<td>68</td>
</tr>
<tr>
<td>Twiki</td>
<td>0.70</td>
<td>1.30</td>
<td>4.62</td>
<td>x1.86</td>
<td>72%</td>
<td>27.3</td>
</tr>
<tr>
<td>CVS</td>
<td>1.05</td>
<td>1.18</td>
<td>2.23</td>
<td>x1.36</td>
<td>46%</td>
<td>17.7</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>22.1</td>
<td>43.6</td>
<td>256</td>
<td>x1.97</td>
<td>83%</td>
<td>664.5</td>
</tr>
</tbody>
</table>

Table 11.3: Index Size(MB)

Table 11.3 shows the Size in MB for the Traditional, Enhanced and Naïve Approaches if predicate-based indexing is used without positional information is used. The benefit of correct results is payed by the price in Enhanced Index which can be even more that twice larger that in case of the Traditional approach in case of Versioning (Wikiboks), but is generally comparable. The Naïve approach always performs much worse than the enhances approach because of redundancy, especially in case of Versioning information.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Trad Index</th>
<th>Enh Index</th>
<th>Naïve Index</th>
<th>Overhead</th>
<th>Space Gain Enh/Naïve</th>
<th>Ix.Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>l³t²X</td>
<td>6.81</td>
<td>10.3</td>
<td>13</td>
<td>x1.51</td>
<td>21%</td>
<td>19.1</td>
</tr>
<tr>
<td>E-mail</td>
<td>33.9</td>
<td>60.8</td>
<td>96.8</td>
<td>x1.51</td>
<td>21%</td>
<td>484</td>
</tr>
<tr>
<td>Java Code</td>
<td>22.9</td>
<td>27.1</td>
<td>32.2</td>
<td>x1.83</td>
<td>15%</td>
<td>68</td>
</tr>
<tr>
<td>Twiki</td>
<td>2.23</td>
<td>4.68</td>
<td>11.7</td>
<td>x2.09</td>
<td>60%</td>
<td>27.3</td>
</tr>
<tr>
<td>CVS</td>
<td>2.77</td>
<td>3.51</td>
<td>6.91</td>
<td>x1.26</td>
<td>49%</td>
<td>17.7</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>54.7</td>
<td>207</td>
<td>581</td>
<td>x3.78</td>
<td>64%</td>
<td>664.5</td>
</tr>
</tbody>
</table>

Table 11.4: Index Size With Positional Information(MB)

When positional information is used, differences between Enhanced and Traditional increase, while the advantage over Naïve is still big, as shown in Table 11.4. This is explainable since Traditional and Naïve do not use the nested positioning scheme needed in the Enhanced Approach.

### 11.3 Index Creation Time

Index creation time is also influenced by introducing rule information. The index creation time in milliseconds is shown in Table 11.5 (without positioning) and Table 11.6. The overhead of Enhanced versus Traditional is still acceptable in both cases (increase by a factor of 1.5), but it is obvious that the Naïve Approach of first materializing instances and then indexing them causes an unnecessary overhead, especially visible in case of E-mail and Wikibooks. The overhead of positioning is only felt in case of Wikibooks (factor of 4 over Traditional), which is however an extreme case, caused by the fact that the data is heavily nested (an average depth of 500 versioned elements).
11.4 Space Gain through Normalization

As shown in Figure 6, Normalization is one of the contributions of this work, which allows to encapsulate both data and semantic information, and index it at the same time. First, Table 11.7 proves that the size of the normalized view is comparable to that of the original data, shown as a baseline. The most important gain is made when comparing the normalized version of the data to the naïve approach which materializes the instances. The benefit is of 80-90% for heavily redundant data such as Wikibooks and Twiki, but also E-mails, where even more overlappings occur (i.e., between conversations and all single messages).
11.5 Query Processing Time

Table 11.8 presents the average running times (in milliseconds) of keyword queries, without positional information. Table 11.9 shows the query processing times when the index with positional information is used. It is obvious that the running times of an enhanced search are longer than the running times of traditional IR since the indexes that need to be scanned are larger, and the logic that merges two inverted lists is more complex (Section 10). Both Table 11.8 and Table 11.9 also mention, for comparison, the query processing times for Windows Desktop Search in case of only non-positional queries (11.8) and respectively, for positional queries.

The differences in running times depend on the data set, on the rules applied, and more importantly, on the type of query. In the cases, where rules affect much of the data (i.e., Twiki, Wikipedia), the increased running time is compensated by the increased quality of the results returned by an enhanced search. Query processing times are, however, very acceptable for desktop search, and remain below 5ms in case of predicate-based indexing without positioning and 11ms if positioning is used. The overhead of positioning is expressed in Table 11.9. In this case, the Enhanced approach is better than the Traditional approach, it is generally performing worse than the Naive approach. This is caused by the overhead for position handling, compared to the small data quantity. For big data, however, such as Wikibook, the overhead of Enhanced versus Naive is clear and times are always comparable and generally much better WDS, as mentioned in Table 11.9.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Trad.</th>
<th>Enh.</th>
<th>Naive</th>
<th>Overhead</th>
<th>Overhead</th>
<th>WDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>\LaTeX{}</td>
<td>1.38</td>
<td>1.61</td>
<td>2.47</td>
<td>x1.20</td>
<td>34.8%</td>
<td>29.31</td>
</tr>
<tr>
<td>E-mail</td>
<td>0.93</td>
<td>4.73</td>
<td>11.08</td>
<td>x1.10</td>
<td>57.3%</td>
<td>7.28</td>
</tr>
<tr>
<td>Java Code</td>
<td>3.91</td>
<td>4.84</td>
<td>6.04</td>
<td>x1.29</td>
<td>19.8%</td>
<td>23.65</td>
</tr>
<tr>
<td>Twiki</td>
<td>0.82</td>
<td>1.24</td>
<td>2.68</td>
<td>x1.42</td>
<td>53.7%</td>
<td>31.56</td>
</tr>
<tr>
<td>CVS</td>
<td>1.75</td>
<td>2.01</td>
<td>2.81</td>
<td>x1.08</td>
<td>28.4%</td>
<td>14.27</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>2.35</td>
<td>3.20</td>
<td>12.86</td>
<td>x1.28</td>
<td>75.11%</td>
<td>26.46</td>
</tr>
</tbody>
</table>

Table 11.8: Query Processing Time(ms) without Positional Information

<table>
<thead>
<tr>
<th>Collection</th>
<th>Trad.</th>
<th>Enh.</th>
<th>Naive</th>
<th>Overhead</th>
<th>Overhead</th>
<th>WDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>\LaTeX{}</td>
<td>6.03</td>
<td>12.76</td>
<td>8.41</td>
<td>x2.12</td>
<td>0%</td>
<td>21.86</td>
</tr>
<tr>
<td>E-mail</td>
<td>3.15</td>
<td>12.88</td>
<td>13.45</td>
<td>x4.08</td>
<td>4.2%</td>
<td>8.94</td>
</tr>
<tr>
<td>Java Code</td>
<td>7.88</td>
<td>15.69</td>
<td>5.72</td>
<td>x1.99</td>
<td>0%</td>
<td>23.65</td>
</tr>
<tr>
<td>Twiki</td>
<td>1.38</td>
<td>9.28</td>
<td>3.43</td>
<td>x6.72</td>
<td>0%</td>
<td>29.64</td>
</tr>
<tr>
<td>CVS</td>
<td>3.72</td>
<td>8.64</td>
<td>2.75</td>
<td>x2.32</td>
<td>0%</td>
<td>11.86</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>3.69</td>
<td>8.62</td>
<td>14.47</td>
<td>x2.34</td>
<td>40%</td>
<td>26.46</td>
</tr>
</tbody>
</table>

Table 11.9: Query Processing Time(ms) with Positional Information.
11.6 Partial Indexing

As mentioned before, introducing rule information has a positive impact on recall but a negative impact on indexing time and query processing time. Therefore, we studied index size, index time, space gain through normalization and recall, when only a part of all the instances are considered in the indexing process. This leads to a lower (but still good) precision but better crawling and indexing time. In order to do this, we indexed the Wikipedia collection with a maximum of ten to a maximum of 100 versions per Wikibooks document, and illustrated the trends of indexing time, index size, size of normalized view, query processing and precision.

**Index Creation Time.** Figure 11.3 shows the index creation time for 10 to 100 indexed versions. This process seems to be very scalable and the query processing time in case of the Enhanced approach increases from 105 to 206 seconds for a factor of 10 increase in number of indexed versions per document.

![Indexing Time (secs.) for Partial Indexing](image)

**Index Size.** Figure 11.4 showcases the increase in Index Size for ten to 100 indexed versions per document. Similarly to the crawling time, index size doubles (from 25MB to 50MB) when considering the non-positional approach but the number of indexed versions increases with a factor of ten. This shows both a comparable overhead compared to traditional, a significant improvement compared to the naïve approach, and also that including more rule information in the index increases the index moderately, but it clearly makes a difference in case of the Naïve approach.

**Space Gain by Normalization.** The most important gain that can be observed is the clear advantage of using Normalization even from a small number of indexed versions per document. Figure 11.5 shows the collection size (i.e., traditional approach), the Normalized View Size (i.e., enhanced approach) and the size of the Materialized Instances (i.e., Naïve Approach). The size of the materialized instances in the Naïve approach increases exponentially. Figure 11.4 shows that already for twenty indexed versions per document (factor of two scale-up), the size of the materialized view is already four times larger than that of the Normalized View.

**Query Processing Time.** The values for the query processing time for the three approaches are specified in Figure 11.6. The query time is constantly higher than in case of the traditional approach, but always under 4ms, and a factor of two smaller that in case of the Naïve approach which can exceed 10ms. This also validates the enhanced approach when compared to the naïve one.
Chapter 11. Experiments

Result Quality. Partial Indexing is a method to decrease the overhead of the Enhanced approach, but sacrifice the search quality. When studying the recall (returned instances) of the enhanced approach for ten to 100 indexed versions per document, it can be observed that half of the maximum Recall can already be obtained when indexing maximum 60 or 70 versions per document. This can be used to set
a margin on the desired search quality and, therefore, limit the overhead on the other parameters.

Figure 11.7: Recall (Returned Instances) in case of Partial Indexing.
Chapter 12

The Case of Enterprise Web Applications

Chapters 4 to 11 discussed the patterns and the framework implementing the enhanced search for desktop applications. In this chapter we discuss how the framework is extensible with other patterns and applicable to Enterprise Data, in particular Web Applications. Indexing enterprise application logic is generally undecidable. In this work, we take the example of dynamically generated Web pages written using frameworks and containers such as JSP J2EE. Furthermore, there is a clear reasons for implementing this additional functionality for enterprise application data.

Motivation: Lack of search engine functionality in applications. Extending enterprise applications with search capabilities is still an open research topic [45]. Large enterprise applications such as SAP and Oracle Finance implement their own search engines. Vendors of small applications cannot afford such an investment and, as a result, small applications either do not provide search facilities or have very imprecise search capabilities. The main problem is to efficiently and completely index dynamic pages which are not physically on disk, which we also call instances, as in Chapter 3. Our goal was (i) to provide search functionality which works in the granularity of final Web pages. (ii) support queries which contain keywords from both the static and dynamically generated part. Moreover, we aimed to do it generically, both independent of the language in which a Web page is written and without accessing the Web container.

The problem of desktop application search is extended to Enterprise Search as shown in the following example:

12.1 Example Revisited

We present Example 2 of Chapter 3 in the context of enterprise search. Consider a Web page which displays information about employees, such as the one in Figure 12.1a. Independently of the language in which it is described, it consists of some static, common content (the text ”Your list of Projects”), and a list with dynamically-generated content which appears in designated places on the page. Data is taken from a database, to generate two possible final Web pages. In this case, the page is dependent on the parameter $empid$ (the id of the employee) and there are as many generated pages as there are values...
for the parameter. The static content is shared between all pages.

The desired search functionality, exactly as presented in Chapter 10 supports keyword queries such as: "Hello" (returns both Paul's and Mary's pages); "Hello Paul", "Hello Mary" (each returns a single page - that of the corresponding employee) but also "Admin Connor" (which returns Mary's page). The queries return documents which are not physically on disk, and use combinations of words from both static and dynamic content. The challenges in case of enterprise search were: generality (i.e., abstraction from the language of the enterprise Web pages and independence from Web container, completeness (i.e., index all possible pages as dictated by the application logic) and efficiency (especially important since there can be a lot of pages). The following Chapter present the patterns in Enterprise Web Applications and the demo showcasing the capabilities of an application data search tool.

The same as in desktop application data, the importance of patterns can be summarized as follows: by using patterns we abstract away from the actual language used to define the Web page (e.g., JSP, PHP) in order to express the application logic. There are more patterns used in the construction of Web pages (e.g., If, List, Placeholder) and they will be categorized in Section 12. This can be considered a model-driven approach to building Web sites [32], an approach taken for example by WebRatio [110]. In particular, in choosing these patterns, we were also inspired by the elements of the Java Standard Tag Library [80] - an attempt to the encapsulate application logic of Web pages in reusable tag libraries.

We describe the few basic patterns we identified in Web applications. A very high percentage of the observed applications use only these patterns. We describe a single scenario for each pattern, and mention the further possible scenarios which it can also covers. First, however, we describe the conventions used to specify the patterns:

### 12.2 Content Descriptors

In order to abstractly describe the content of an enterprise Web page and in order to be able to specify possible pattern occurrences, we use a language-independent format. A file written in this format is called a content descriptor. The format contains typical elements of dynamic Web pages. The use of content descriptors does not restrict generality of the approach, it is however necessary in order to easily refer to elements of the page which exhibit a certain pattern, and to ignore parts of the page which do not contribute to the search result. In future, we intend to apply the pattern-based approach directly to the XML representation of JSP pages.

![Figure 12.1: Typical Structure of a Web Page in an Enterprise Web Application](image-url)
12.3 Datasources

In the enterprise world, we have access both to the content files and to the data used to generate dynamic content. The dynamic part of a Web page (written in a language such as JSP or PHP), also describes data access. As a convention, we use XPath and XQuery expressions for this purpose. This brings maximum decoupling from the data model, and is especially sustained by the fact that commercial databases such as Oracle and DB2 can provide an XML-View of any relational table, and allow XML-SQL queries to be performed on mixed content.

12.4 Parameters

Application logic might be dependent on parameters. Each value of a parameter creates another possibility to generate the existing page. For each parameter, its name and its domain must be known and declared in the content file. In the following example, the content file contains one parameter with the name \texttt{category\_id}. The domain of this parameter is loaded from an XML file, using an XPath or XQuery expression, as explained in Section 12.3.

```xml
<params>
  <param name="category\_id"
    domain="doc('petStoreData.xml')//Category/@id"/>
</params>
```

This way to specify parameters provides another level of abstraction as what regards the method used to transmit parameters to the Web page. The method (in particular \texttt{GET} or \texttt{POST}) is abstracted away and actually irrelevant for the indexing and for the search process.

12.5 Rules

The content descriptor encodes an abstract version of usual application logic. In order to enable the search functionality, it is necessary to specify where patterns occur in the abstract representation. Therefore, we use a special notation which marks specific elements as carrying the behaviour of certain patterns, independently of the language of the dynamic Web page. As explained above, using a content descriptor does not reduce generality. This could be applied to any XML-representation of a Web page which follows the specified guidelines for the content descriptor. In particular, we plan to adapt the framework to an XML-version of the JSP language. Rule examples can be found in Section 12. From a usability perspective, it is very likely that in future rules will be automatically generated by development tools, such as [110], a model-driven Web development tool or by an online Web-application-building tool such as Google Sites [70].

Here are the patterns we identified in enterprise Web applications:

12.6 “Output”

The \textit{Output} pattern specifies two instances: one for each possible value of its parameter \texttt{(emp\_id)} - represented in Figure 12.2b and 12.2c. An instance contains the common content of the page (the text
(a) **Dynamic Page:**

```
Hello <out expr="doc(...)//empl[emp_id=$emp_id]/Name"></out>
```

(b) Instance 1:

```
Hello <out>Paul Smith</out>
```

(c) Instance 2:

```
Hello <out>Mary Connor</out>
```

Figure 12.2: The "Output" pattern

*Hello* and the specific content of the instance (the name of each employee, as described by the result of the XPath expression). The instances of the *List* pattern are also two (i.e., the list of Projects of each employee). The *List* pattern is presented in Section 12.8.

### 12.7 “If”

This pattern is useful to describe that parts of dynamic Web pages which appear only depending on the value of one parameter. In the following document, the *if* element contains content that is dependent on the parameter *category_id*. The rule identifies matching elements (i.e., all *if* elements in the content descriptor). The variable $m$ will be associated to each one of these elements. Conditional branches are identified by the *case* subelements for each value of $m$, and the conditions as defined in the *cond* subelements of each *case* (associated to the variable $c$). In our case, there is a separate instance for each value of the parameter *category_id*.

Extensions of this pattern can describe *multiple choice*, *alternatives* (e.g., drop boxes), *try/catch* blocks, all implemented by our approach.

**Content Descriptor:**

```
<if>
  <case cond="$category_id=FISH">
    The param category_id is FISH
  </case>
  <case cond="$category_id=BIKES">
    The param category_id is BIKES
  </case>
</if>
```

**Rule:**

```
<if match="/if" cases="$m/case" condition="$c/@cond"/>
```

**Instance 1:**

```
<case cond="$category_id=FISH">
  The param category_id is FISH
</case>
```

**Instance 2:**
12.8 “List”

This pattern is used to represent a list of results from a query. Embedded elements mapped to the Output pattern are used for displaying the results. The reference in the list definition declares the query which specifies the actual elements of the list. Each of these elements can be accessed by using the symbolic value declared in the attribute item. List patterns may also be dependent on parameters: an instance (i.e., a list) is created for each possible value allocated to the parameters of the list. For each of these instances, the content of the element in the list element in the content descriptor is considered and eventual elements corresponding to the Out pattern are, at their turn, instantiated.

Content Descriptor

```
<li reference="doc('...')/products/[@cat_id=$category_id]"
     item="$p" params="('category_id')">
   <out expr="$p/Name/text()"/>
</li>
```

Rules

```
<list match="/li" ref="$m/@ref" item="$m/@item"
     params="$m/@params"/>
<out match="/out"/>
```

Instance1:

```
<list...> Cat1_product1 Cat1_product2 ...
</list>
```

Instance2:

```
<list...> Cat2_product1 Cat2_product2 ...
</list>
```

12.9 “Placeholder”

Imports another content file to which rules may also apply. A typical example are headers or copyright messages common to all pages, or even dynamic subpages which just contain common code for displaying the current product categories in a store.

Content Descriptor

```
<include path="versionComment.xml"/>
```

Rule

```
<include match="/include" path="$m/@path"/>
```

All desktop patterns presented in Chapter 4 are also applicable to enterprise application data.

12.10 Applicability in Practice

This section discusses several points sustaining the general applicability of the approach:
• **Collaboration of the application developer.** For this demo, the content descriptors have been manually generated. We think that along with a new wave of more complex enterprise Web applications, most part of these applications will be automatically generated, or generated using tools. This will alleviate the work of the application developer, who will need to describe the functionality only once.

• **Expressiveness of rule language.** Our current rule language can express a large part of the functionality in the Pet Store application. A significant exception are update pages (such as “Add to cart”), and for a good reason. In this case, content is indexed only if it does not depend on hypothetical values (e.g., we do not index the products a user might introduce in its Shopping Cart, or the amount the user might pay with his credit card). We aim however to address issues related to modeling workflows in Web applications. However, since update pages are relevant in this context, they will be eventually be included in the solution.

• **Tools and Automation.** Our implementation focused on a tool for indexing and query processing enterprise Web applications based on abstracting the application logic. This abstraction and the specification of rules could be in future supported by tools such as WebRatio [110], while current Web Application Framework such as Struts [105] or Java Server Faces [79] can serve as a base for deriving application architecture, the page models and workflows.
Chapter 13

Demonstration

Chapters 4 to 12 showcased the patterns in desktop and enterprise applications and the enhanced search framework for application data. We have implemented the predicate-based indexing framework in Visual Studio.Net 2005. We applied it to the J2EE PetStore application (Figure 13.1), implemented using JSP. We added indexing and search functionality to the application.

13.1 Test Environment

The framework was run on an IBM Thinkpad T42 Laptop, with 1 GB RAM memory and 70 GB hard disk. The demo shows how data can be indexed based on the content files and rules, and how keyword and phrase search can be performed. The Indexes are enhanced inverted files, as described in Section 7. A GUI is used for specifying the content file and the rules for performing indexing, or the the keywords or phrase query in case for performing search. Results are \( < \text{doc, predicate} > \) pairs, presented in a user-friendly way and with the possibility to view the initial page in the browser.


Figure 13.1: Example of the PetStore Application
13.2 Test Data

We manually generated content files for the relevant files in the J2EE PetStore application. A fragment from this page, when displayed in a browser with the parameter $category_id = BIRDS$, can be seen in Figure 13.1.

Content Descriptors

The content descriptor (Section 12.2) for this dynamic Web page contains the parameter definition and parameter domains, loaded from the original XML data file of the PetStore application:

```xml
<params>
  <param name="category_id" domain="doc('petStoreData.xml')/.../Category/@id"/>
</params>
```

The menu on the left of Figure 13.1 is the list of all categories, which is not dependent on parameters. It can be represented as follows:

```xml
<list ref="doc('...')/..//Category/CatDetails[lang='en-US']">
  <out expr="$r/Name/text()"/>
</list>
```

The `list` element is mapped to a `List` pattern and declares a list with categories loaded from an XML-file. The definitive content is defined with an XPath expression. The `out` subelement of `list`, corresponding to the `Out` pattern, will display the name of each category selected from the XML file by the list patterns. Specifically, category names are: Birds, Cats, Dogs, Fish, Reptiles.
13.3. Indexing and Search

The product list, displayed at the right in Figure 13.1, describes the products of a given category and, therefore, depends on the parameter \textit{category\_id}:

\begin{verbatim}
<list ref="doc(\ldots)/\ldots/Prod[@category = $category\_id]/\ldots"
    params="('category\_id')"
    value="$r">
    <out expr="$r/Name/text()"/>
    <out expr="$r/Description/text()"/>
</list>
\end{verbatim}

In the same way, this new list element selects \textit{product} elements for the given \textit{category\_id}, while name and description of each product are displayed by applying the \textit{out} pattern.

Rules

The rules for each pattern in the Java PetStore application are declared exactly as described in Section 12.5. It is worth mentioning that the rules associating the behaviour of the \textit{List} and \textit{Out} patterns to the \textit{list} and \textit{out} elements in the content descriptors are declared just once for the whole application, and, therefore, not for each content descriptor. This is made possible by applying patterns rigorously throughout the application and favored if the Web application is built with patterns in mind. In particular, the initial JSP pages of the PetStore application made use of tag libraries. This made the generation of content descriptors straightforward.

13.3 Indexing and Search

Indexing (described in Section 7) can be performed with several options. It is possible to add positioning and scoring information to the index or to save the index in a compressed or uncompressed way. These options are available through the GUI shown in Figure 13.2, a screenshot of the application during indexing process. Before the actual indexing is performed, the normalized view is created (but only materialized when required). In the normalized document, the dynamic parts are tagged with \textit{encoded} parameter information. Here is an example: the product names and descriptions for \textit{category\_id} = BIRDS. In this example, “1” encodes the parameter \textit{category\_id} and the value “4” represents the encoded value of the string “BIRDS” for this parameter.

\begin{verbatim}
<e:s v="1" k="4">
    Amazon Parrot: Great companion for up to 75 years
    Finch: Great stress reliever
</e:s>
\end{verbatim}

For maximum space gain, indexing also makes use of the same dictionary-based compression techniques as in the actual normalized view. The mapping between encoded and actual parameter values is maintained and will be used after query processing, when presenting the results to the user. Both \textit{keyword search} and \textit{phrase search} are possible on the enhanced inverted files. If enabled, results are ranked based on the relevance of the instance result among the whole set of instances. Since parameters are encoded, result are decoded and presented to the user as in Figure 13.3.
13.4 Statistics

One big advantage of predicate-based indexing is the small size of the index (for dynamic content). We compared it to the traditional index (all instances materialized):

**Original Data**: Database 40kb, Source files 4.9kb  
**Traditional Indexing**: Index 33.5kb, Materialized Pages 51.8kb  
**Predicate-based Indexing**: Index 10.8 kb, Normalized View 7kb

First, it is important to mention that the traditional index is significantly bigger than the predicate-based one because common content is indexed repeatedly in the traditional approach. Also, the overhead of predicates is not high. Second, normalization pays off and the space gain is significant compared to the traditional approach of materializing all instances. Actually, normalization achieves a compression ratio of almost 8 times as compared to full materialization. Third, generating all possible combinations of page content and database would also be unfeasible. Taking into account only the combinations allowed by the application logic (as abstracted by patterns) brings clear benefits in space. Query processing time is not included here because of possible lack of precision considering the small data size. It is however comparable to the traditional approach (i.e., the overhead of predicates is not high) and does not exceed 10 milliseconds.

13.5 Discussion

The previous sections described the framework for indexing enterprise applications and its use for indexing a real application (Sun’s Java Pet Store). This section discusses several points sustaining the general applicability of the approach:

- **Collaboration of the application developer.** For this demo, the content descriptors have been manually generated. We think that along with a new wave of more complex enterprise Web applications, most part of these applications will be automatically generated, or generated using tools. This will alleviate the work of the application developer, who will need to describe the functionality only once.

- **Expressiveness of rule language.** Our current rule language can express a large part of the functionality in the Pet Store application. A significant exception are update pages (such as “Add to cart”), and for a good reason. In this case, content is indexed only if it does not depend on hypothetical values (e.g., we do not index the products a user might introduce in its Shopping Cart, or the amount the user might pay with his credit card). We aim however to address issues related to
modelling workflows in Web applications. However, since update pages are relevant in this context, they will be eventually be included in the solution.

- **Tools and Automation.** Our implementation focused on a tool for indexing and query processing enterprise Web applications based on abstracting the application logic. This abstraction and the specification of rules could be in future supported by tools such as WebRatio [110], while current Web Application Framework such as Struts [105] or Java Server Faces [79] can serve as a base for deriving application architecture, the page models and workflows.
Chapter 14

Related Work

This work is closely connected to several other research projects spanning the world of Databases and Information Retrieval. Some patterns (e.g., Comments, Excluded) were inspired by the PIX [10] project at AT&T. Generically, other approaches trying to add metadata for describing relationships in and between documents are the objective of Semantic Web [99], [90] but use advanced query languages that support reasoning [58]. This work is different in that it aims to annotate documents for specifying their semantics and the multiple instances which can occur in the data with the purpose of performing keyword search correctly. The patterns addressed in this work span a larger range though. Semantic search engines for XML have also been proposed by [42]; the approach extends traditional IR techniques for defining correlations between a query and the returned XML fragments.

We are more related to other approaches in the database and IR community: Colourful XML was proposed in order to give an XML document several interpretations [78]. The idea of multiple instances can also be found in the Database world in probabilistic databases [2]. Our approach is more database-oriented from the point of view which can be generated (both structure and content-specific), but less powerful that a full-fledged probabilistic database by not defining probabilities. The work of Koch et. all [13] actually defines an Alternative pattern applicable to a relational world. Versioning is a particular case taken into account by several IR approaches [19], [12]. The problem of shared content in documents was also addressed by [26], generating a similar approach to placeholders, which in just another special case for us. Superimposed Information, Comments, Tags are addressed in [93] and are an integral part of community portals such as [60].

As what regards the View Definition and Normalization, the rule language is a particular case that can be expressed by a transformation in established languages such as XSLT [39], XQuery [21], while the queries are a candidate for XQuery Full-Text [8]. In particular, the rule language is optimized for implementation of the view generation, and does not need the expressive power of XQuery. The Normalization is also an optimized version of the application view. Its goals of eliminating redundancies is also addressed by relational normalization [41], XML Normalization [14]. Normalization is also a compression techniques for XML, only aimed for optimized indexing, as opposed to XMil [85] or [30], which aim to perform structured queries on compressed XML. Serializing XML version of the Normalized View. Several serialization approaches have also been proposed by ColorfulXML [78] and binary XML [62]. XML Views and XML summaries are also studied in [102] and DataGuides [66]. Special positioning techniques are part of work on relational [106], XML [95], or XML-IR worlds [73].
Advanced indexing techniques for XML are proposed by [9], while query relaxation techniques are present in [84], [31]-their aim is to increase specificity of results and to infer the semantics of the data by using IR-specific techniques. Optimal join processing algorithms have been studied in [6], [82], [34], and also top-k search [107] and ranking strategies in XML [73]. We adapted [15] for this work. [19] addresses indexing and querying versioned repositories. Phrase search at an aim to increase quality and locality of search results is also addressed in particular by Google [69]. As opposed to [9], we use the Dewey ID positioning technique instead of structural index in order to optimize the computation of results if XML element results are nested. Finally, works such as the XQueryFullText standard is relevant but is orthogonal and can be applied in the Indexing and Search phase of Figure 3.3.

Finally, we have presented an architecture which we showcased through a demo, which adds search capabilities to Web applications in a generic way. It is independent of the language of the application and does not require the collaboration of the Web container. Although preliminary, the approach is promising for its accuracy and its applicability to current enterprise standards such as Java Standard Template Library (JSTL[80]). The idea could also be applied in the context of indexing the Hidden Web [98], but our proposed approach does not require a running Web container for indexing and search. A particular disadvantage of hidden Web crawling is the necessity to “guess” possible input values for fields in Web forms (such as a login page), in order to have access to the pages returned after the form was submitted. Our framework eliminates this by having access to both source files and database (i.e., values are known). Next steps in future work are applying the framework to a complex enterprise application and fully adapting it to the JSP-XML and JSLT format are the next decisive steps. Also, security and privacy issues could be expressed in terms of predicates, which is a natural application of the ideas in this framework.
Chapter 15

Summary

This part of this work was motivated by the observation that current search engines look at desktop and data differently than applications. As a result, a state-of-the-art search engine might return a document although it is not relevant for a query and, similarly, it might miss relevant documents. Custom content converters are currently needed to allow indexing of annotated application data.

The key idea of this work is to extend current search engine technology to index and search application views generically. An enhanced search engine uses rules to define those views. A normalized representation for each application view is derived from these rules and the original data. The normalized view encodes the whole application view in a compact and generic way. The main contribution of this work is to define such normalized views, give an algorithm to generate them, show how extended inverted files can be constructed on normalized views, and define the extensions that need to be carried out to the merge algorithm of a keyword search query processor. IR-specific extensions such as Phrase Search have been implemented on the application data. The performance results are encouraging because they demonstrate that the overheads of such an enhanced search engine are tolerable. Custom content converters can be, therefore, replaced by a generic approach.

There are several avenues for future work. One avenue is to apply more powerful query paradigms to normalized views, i.e., extend keyword search to XQuery. Furthermore, XML information retrieval approaches, such as query relaxation and structure-based search, could be explored, as described in Section 14. Adjusting indexing and search based on the type of the annotations is an important source of optimizations. Another important direction for future work is to apply these techniques to other classes of applications, such as Scientific data, electronic health records, and even enterprise application data. Personalization, allowing multiple concurrent rules but also security issues are another step towards a more practical solution.

Another direction is to apply the techniques proposed in this work for application search to the “Hidden Web”. Rules could be seen as a modern version of robots.txt files that describe how to interpret the data found by the crawler. In order to be practical, it is important to create libraries of rules for documents generated by popular applications (e.g., Microsoft Office, Sun’s OpenOffice, Twiki, \LaTeX, E-Mail, CVS, etc.).
Part II

Crawling, Indexing and Searching
AJAX Applications
Chapter 16

Problem Statement

Google Mail[65], Yahoo! Mail[116], Google Maps[68] are well-known AJAX applications. Their goal is to enhance the user experience by running client code in the browser instead of always refreshing the Web page, and minimizing server traffic. Traditional Web Applications such as Amazon and YouTube! also start including AJAX Content in traditional pages in order to offer higher interactivity to the user. An AJAX Application is a dynamic Web application at a given URL, usually based on Javascript, which presents to the user different states changed by the user through UI events. Taken to the extreme, an AJAX application can present all the states of the application seamlessly, without changing the URL. This causes a mismatch with the current search model of Google-like search engines, where the presented results are uniquely identified by a URL. This causes a loss in result quality as shown on the following example.

16.1 Motivating Example

YouTube[117] is one Web site which has changed from a traditional, Web-page-based interface, to one which includes AJAX parts. This is a trend also followed by Amazon, for example, which dynamically includes excerpts from books or suggesting related products.

Figure 16.1 displays schematically the YouTube GUI for a video. The YouTube interface for a given video includes comments from the users. The first page of comments is displayed by default in a comment box,

Figure 16.1: YouTube: Comments load using AJAX.
below the movie. The rest of the comments are paginated. They can be accessed using a menu with the page number (1, 2, etc.) or using two links, next and previous. The comments are loaded from the server using AJAX and displayed in the same box, but the URL of the page remains the same. For each click, a Javascript event is triggered, and an AJAX call is made to the server which seamlessly loads the new content in the same area, leaving the rest of the page unmodified. The user view consists, therefore, of the movie and of the comment pages.

Current search engines do not index AJAX content. However, the following example showcases the benefits of searching the dynamic AJAX part of the application. The particular video of the band Morcheeba is called “Enjoy the Ride”; a piece of information included in the title.

**Traditional Search.** We focus on boolean retrieval. YouTube users can submit the boolean query Q1: “Morcheeba Enjoy the Ride”, using YouTube’s custom video search and will get the video as a result. This Morcheeba video is however special for a music fan, both because it is the newest video of the band with a new unknown singer and because of its chosen topic. Therefore, an interested user may not know all information needed for finding this video. This is when AJAX content (the video comments) becomes useful.

**AJAX Search.** In order to find the title of the new video, an unprepared user should be able to search for the boolean query Q2: “Morcheeba mysterious video” and get this video as a result. This only works if both the band name (non-AJAX) and the comment text are crawled. In a similar way, the name of the new singer can be obtained using Q3: “Morcheeba Enjoy the Ride Singer” using the band name and the text in the second comment page. Traditional search engines cannot do this because they do not crawl the AJAX Content. The question is how AJAX content can be crawled, and which is the performance overhead of crawling AJAX by invoking user events. This impacts all applications such as Amazon or YouTube that start including AJAX content in their traditional Web pages.

### 16.2 Problem Description

Current search engines ignore AJAX content. The reason lies because of the actual difference between traditional and AJAX crawl, as follows.

In traditional search engines one can build a model based on HTML Web pages and hyperlinks. AJAX Web sites also contain Web pages connected through HTML hyperlinks, but the difference is that each page contains “states”, generated when the user interacts with the state using Javascript events, i.e., content and the corresponding Javascript state (variables, etc.).

Since the AJAX page model is still a graph, as in traditional graph, albeit more complex, one could argue that states can be modeled as normal pages, and transitions exactly as hyperlinks. The following factors argue against this argument:

- **High fan-out.** AJAX gives application developers the possibility to include very granular interactions in the Web page. The granular events interactions cause a high fan-out of the graph modeling AJAX Web Sites.

  Negative ratio Events/Pages. The high fan-out of the graph modeling AJAX Web Sites cause a negative ratio events/pages, when compared to the situation when the Web Site had only been built with traditional Web technologies only.
Because of these challenges, current search engines do not crawl AJAX and provide the workarounds that are showed below. However, they are not generic, and second, not accurate enough or may require the collaboration of the application provider:

**Hand-Coded Web Pages.** Special Web pages can be set up in order to include an alternate view of the dynamic content. This page is hand-coded, less rich in content than the initial page and causes therefore a loss of information. Currently, Google finds this page and indexes it. Our goal is to avoid hand-coded solutions and be generic.

**Custom Search Engines.** Applications such as YouTube provide their own search engine. This engine does not usually have access to all dynamic content, therefore it is limited. Furthermore, implementing custom search engines is something small application providers cannot afford.

**Exposing Data to Search Engines.** Bigger Web Sites can agree to give search engines direct access to the data with generic credentials. This provides better accuracy, but may prove too coarse-grained since it might provide no specific data. We avoid this and we propose a generic solution for AJAX Crawl.

### 16.3 Plan of Attack

We address the problem of AJAX Crawling. We extend traditional search and bring the following contributions:

- **Modeling AJAX Sites.** We propose a model of an AJAX Web Site. We address text-based AJAX Applications without user input (i.e., no forms).

- **AJAX Crawler.** We propose an AJAX Crawler which crawls based on user events. We provide an optimization to the problems of caching and duplicate elimination of states.

- **Evaluating the gain in result quality.** We evaluate the improved recall of the AJAX search over traditional search. Results are obtained on a YouTube subset.

- **Evaluating performance tradeoff.** We evaluate the performance price payed for the improved search results on the YouTube subset and custom AJAX Application.

This part of the dissertation is organized as follows: Chapter 16 presented the motivation of AJAX Crawling. Chapter 17 describes the new model of AJAX Web Sites, pages and events. Chapter 18 and Chapter 19 are the main contribution of this work and describe the crawling algorithm and a solution to the problem of detecting duplicates and caching in AJAX crawling, using memoization. Chapter 20 describes the overall architecture of a search engine, including parallelization. Experimental results are shown in Chapter 21 and Chapter 22 discusses the inclusion of AJAX Search in current search engines. Chapter 23 presents a demo of the approach applied to an AJAX news application and Yahoo!Mail, Chapter 24 presents related work, while Chapter 25 presents contain conclusions and future work.
Chapter 17

Modeling AJAX

As mentioned before, this work focuses on AJAX Crawling. Crawling is an operation which is able to read a dynamic Web page and build its model, so that it is relevant for search. This extends traditional search, where crawling just indexes simple Web pages.

17.1 Event Model

When Javascript is used, the application reacts to user events: *click*, *doubleClick*, *mouseover*, etc. An example of such an event is displayed in Figure 17.2. The *onClick* event triggered on the (*div id = "next"*) HTML element *source*, applied to the *recent_comments* element (*target*). The content of the targets changes using the action: *recent_comments.innerHTML=*.... We will use these notations throughout this work.

17.2 AJAX Page Model

Section 16 presented an AJAX application as not only a simple page identified by an URL, but also as a series of states, events and transitions. This is the main difference between traditional search and AJAX Search and we model it correspondingly. The AJAX Page Model is a view of all states in a page (e.g., all

![Diagram of AJAX Web page states and transitions]

Figure 17.1: Model of an AJAX Web page.

![Event Structure in Javascript]

Figure 17.2: Event Structure in Javascript.
comment pages). In particular it is an automaton, a **Transition Graph**. The Transition Graph contains all application entities (states, events, transitions) as annotations. It is defined by:

<table>
<thead>
<tr>
<th>Event</th>
<th>Start State</th>
<th>End State</th>
<th>Source</th>
<th>Event</th>
<th>Target</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>s1</td>
<td>s2</td>
<td>“next”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
<tr>
<td>e1</td>
<td>s1</td>
<td>s2</td>
<td>“page 2”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
<tr>
<td>e2</td>
<td>s2</td>
<td>s3</td>
<td>“next”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
<tr>
<td>e2</td>
<td>s2</td>
<td>s3</td>
<td>“page 3”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
<tr>
<td>e3</td>
<td>s3</td>
<td>s2</td>
<td>“prev”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
<tr>
<td>e4</td>
<td>s2</td>
<td>s1</td>
<td>“prev”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
<tr>
<td>e4</td>
<td>s2</td>
<td>s1</td>
<td>“page 1”</td>
<td>onClick</td>
<td>recent_comments</td>
<td>innerHTML</td>
</tr>
</tbody>
</table>

Table 17.1: Annotation for the Transition Graph of an AJAX application.

- **Nodes.** The Nodes are application states. An application state is represented as a DOM tree. It contains at each stage in the application the current DOM with all corresponding properties (ID).
- **Edges.** The edges are transitions between states. A transition is triggered by an event activated on the source element and applied to one or more target elements, whose properties change through an action.

The Transition Graph is best explained using Figure 17.1, which models the next and previous events invoked on the corresponding buttons of the YouTube application. Traversals of this graph can be entered in a table constructed as Table 17.1. For each transition between Start State and End State, the Source, Event, Target(s) and Action(s) are entered in the table. In Table 17.1, transitions from State 1 (s1) to State 2 (s2) can be made using both a click on the next element or on the “page 2” element. This affects the recent_comments element through the innerHTML action.

### 17.3 Modeling AJAX Web sites

As opposed to traditional Web, an AJAX Web site contains both static and dynamic content. Furthermore, each page contains hyperlinks to other web pages as shown in Figure 17.3.

![Figure 17.3: Model of an AJAX Web Site: AJAX Pages, hyperlinks and AJAX states.](image)

The difference to the traditional Web is that the user may trigger events in the same page (such as next and prev) which generate new application states. In order to complete the comparison, the transitions caused by the events may be called AJAX links.
As opposed to this, traditional Web Sites are characterized just by a graph of *pages*, connected by hyperlinks. Crawling the AJAX part of a Web site leads to an increase in search quality, but also to an overhead that needs to be addressed in terms of performance. The overhead is caused by high fan-out of the AJAX Web Site model and by the negative ratio event/pages, when compared to a Web site that has been built using traditional models.
Chapter 18

AJAX Crawling

The main contribution of this work is an AJAX Crawler which addresses the issues of crawling events and building the extended AJAX Model. We present a basic algorithm which we improve in order to address duplicates and caching, two particular issues specific to crawling dynamic content and events.

18.1 Basic Crawling Algorithm

The role of the AJAX Crawling algorithm is to build the model of the AJAX Web Site. Since building the hyperlink graph is a solved problem in traditional search engines, we focus on the algorithm which builds the AJAX Page Model. (i.e., for YouTube, indexing all comment pages of a video).

We detail the crawling algorithm for AJAX applications in Algorithm 18.1.1.

The first step of crawling is to read the initial DOM of the document at a given URI (line 2). The next step is AJAX-specific and consists of running the onload event of the body tag in the HTML document (line 3). All Javascript-enabled browsers invoke this function at first. Crawling starts after this initial state has been constructed (line 5). The algorithm performs a breadth-first crawling, i.e., it triggers all events in the page and invokes the corresponding Javascript function. Whenever the DOM changes, a new state is created (line 11) and the corresponding transition is added to the application model (line 16). As mentioned in Section 17, a transition is annotated with the event information: source, trigger, action(s) and modif(s). After a new state has been reached, the crawler reconstitutes the initial DOM in order to be able to invoke the next events in the initial state. After all events have been triggered, the immediately adjacent states in the application graph can be crawled.

Differences from Traditional Crawling. AJAX Crawling is similar to traditional crawling and the same problems arise: duplicate elimination, large number of states, high amount of network calls. For example, special care must be taken in order to avoid regenerating states that have already been crawled (i.e., duplicate elimination). This is a problem also encountered in traditional search engines. However, traditional crawling can most of the time solve this by comparing the URLs of the given pages - a quick operation. AJAX cannot count on that, since all AJAX states have the same URL. This requires the need to define an efficient similarity function between two states. Currently, we compute a hash of the content of the state. Two states with the same hash value will be considered the same and the state will not be duplicated in the application model.
Algorithm 18.1.1 Breadth-First AJAX Crawling Algorithm

1: Function init(url)
2:  dom = readDocument(url)
3:  dom.executeFunction(body.onLoad) {AJAX Specific}
4:  appModel.add(dom) {Add first state to the App. Model}
5:  crawl(dom)
6: end Function

7: Function crawl(State s)
8:  for all Event e ∈ s do
9:    dom.executeFunction(e.function)
10:   if dom.hasChanged() then
11:     State newState = new State(dom)
12:     if appModel.contains(newState) then
13:       newState = appMode.get(newState)
14:     end if
15:     Transition t = new Transition(e.source, e.trigger, e.action*, e.modif*)
16:     appModel.add(t, s, newState)
17:     appModel.rollback(t)
18:   end if
19: end for
20: for all Transition t ∈ (s, s1) do
21:   Crawl s1 {Breadth-first traversal of reachable states}
22: end for
23: end Function

18.2 Addressing Crawling Challenges

The challenge of crawling AJAX is to retrieve as many application states as possible. Each state is a DOM tree to which events are attached. However, especially since events leading to a new state can be so granular as to change only a minimal amount of content from the page, it is important to minimize the number of generated DOM trees. As also shown in the Transition Graph of Figure 17.1, a state might be reached again as a consequence of the invocation of more events (e.g., state 2 can be reached either by clicking the next arrow from state 1 or the previous arrow from state 3. There are a few optimizations which can be applied:

**Infinite state expansion.** If the same events can be invoked indefinitely on the same state, the application model can explode. We solve this by limiting the amount of iterations.

**Infinite loops.** The code running into infinite loops can also cause an explosion in the application model. Still, we do assume this problem does not occur too often and we apply a hard-coded limit on the number of states that we index. Code analysis was not in our scope.

**Identifying identical states.** We do this by applying a technical optimization. Furthermore, we apply an additional heuristic crawling policy which detects and reduces expensive server calls (Hot Nodes), described as follows in Section 19.

**Irrelevant events.** There are many events in an AJAX Application, each causing changes in the application content and structure. Ideally, a totally automatic approach would generate all application states based on any granular event. We can focus just on the most important events (click, doubleclick, mouseover).

**Number of AJAX Calls.** We identified that the biggest challenge in the performance of the AJAX Crawler...
is the high amount of calls to the server, especially since this can lead to the same state. We developed a special technique for solving this problem, mentioned below.

In the following we improve the crawling algorithm in order to overcome the main bottleneck: the number of irrelevant AJAX invocations to the server.
Chapter 19

A Heuristic Crawling Policy for AJAX Applications

Section 18 presented a basic traditional crawling algorithm. In this section we optimize AJAX Crawling algorithm by applying the “Hot Node” approach. The main performance decrease in AJAX Crawling are caused by a large number of network calls and the large number of events leading to the same state (a flavor of the duplicate elimination problem). These problems appear in traditional search, but the same factor that makes AJAX search different that traditional search, i.e., the fact that transitions are caused by function invocations on the client, is taken into account as a way to optimize the crawl. We deal with this problem by identifying the same state but before fetching the content twice.

19.1 Intuition

We base our reasoning on the following statement: Crawler sufficiency criterion. The functionality of an AJAX Crawler is sufficient if all content is fetched from the server, and the model is built correctly.

An AJAX Crawler An AJAX Web Site is made possible because Javascript functions are invoked on the client-side. The same invocations cause the problems presented above. However, not all calls are equally expensive. The functions that invoke content from the server are expensive.

Hot Nodes. The Javascript functions that actually make the server invocation and fetch content from the server.

Lemma. The functionality of an AJAX Crawler is sufficient if all necessary Hot Nodes invocations are done once (each time with the corresponding arguments).

The following steps describe the Hot Node approach, and how hot nodes are identified and how the problem of avoiding duplicate states is reduced to the problem of avoiding duplicate Hot Node invocations.
Chapter 19. A Heuristic Crawling Policy for AJAX Applications

19.2 Javascript Invocation Graph

The heuristic we use is based on the runtime analysis of the Javascript invocation graph. This structure contains a node for each Javascript function in the program and its dependencies (i.e., invoked functions). The invocation Graph for the YouTube Web site is depicted in Figure 19.1.

The nodes in the Javascript invocation graph are Javascript functions. The functionality of an AJAX page is expressed through events. In case of YouTube, these are next, prev, jump to page, as shown in Table 19.1. Functions in the Javascript code can be invoked either directly by event triggers (event invocations) or indirectly by other functions (local invocations). Table 19.2 lists some functions appearing in the YouTube page.

In order to fulfill their functionality in an AJAX application the events call, directly or indirectly, the Hot Nodes. The dependencies in the code are listed in Figure 19.3. In case of YouTube, we have a single Hot Node, i.e., getURLXMLResponseAndFillDiv. In AJAX, the same Hot Node can be invoked from different comment pages with the same argument in order to fetch the same content from the server, as shown in Figure 19.3. For example, both events “Next page (from page 1)” and “Jump to Page 2” lead to the same server invocation, for page 2. In this approach we detect this situation and we avoid invoking the same function twice, as shown below.

<table>
<thead>
<tr>
<th>Event #</th>
<th>Functionality</th>
<th>Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“init page”</td>
<td>onload</td>
</tr>
<tr>
<td>2</td>
<td>“next page” (from page 1)</td>
<td>onclick</td>
</tr>
<tr>
<td>3</td>
<td>“prev page” (from page 2)</td>
<td>onclick</td>
</tr>
<tr>
<td>4</td>
<td>“jump to page 2”</td>
<td>onclick</td>
</tr>
<tr>
<td>5</td>
<td>“jump to page 3”</td>
<td>onclick</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td>onmousedown</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
<td>onmouseover</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 19.1: Events and functionalities in the Javascript Invocation Graph.
19.3 Optimized Crawling Algorithm

We solve the problem of caching in AJAX applications and detecting duplicate states by identifying and reusing the result of server calls. Just as in traditional I/O analysis in databases, we tend to minimize the number of the most expensive operations, i.e., the *Hot Calls*, invocations which generate AJAX calls to the server. The new Crawler with heuristics can be summarized in Algorithm 19.3.1. The main points of the algorithm are:

**Step 1: Identifying Hot Nodes.** The crawler tags the *Hot Nodes*, i.e., the functions that directly contain AJAX calls (line 34). This can be detected by analyzing the Stack trace of the Javascript invocations during the AJAX Crawl, as shown in Figure 19.2) for the case the crawler is on comment page number 2 of a certain video: if the top level fetches AJAX Content on the server, the immediately lower function is an AJAX call, in this case, a single function In the YouTube application, there is one hot node, i.e., the function `getURLXMLResponseAndFillDiv(URL, ID)` marked with A.
Chapter 19. A Heuristic Crawling Policy for AJAX Applications

<table>
<thead>
<tr>
<th>Level #</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SERVER INVOCATION</td>
</tr>
<tr>
<td>1</td>
<td>(A) getURLXMLResponseAndFillDiv (<a href="http://youtube/v=abc">http://youtube/v=abc</a>, 2)</td>
</tr>
<tr>
<td>0</td>
<td>(0) click event “next” (from page 2)</td>
</tr>
</tbody>
</table>

Figure 19.2: Javascript stack trace during AJAX Crawling (grows bottom up).

**Step 2: Building Hot Node Cache.** The crawler builds a table containing all *Hot Node* invocations, the actual parameters used in the call and the results returned by the server (line 34-53). Whenever a Hot Call is intercepted, an entry is made in this table.

**Step 3: Intercepting Hot Node Calls.** The crawler adopts the following policy:

1. Intercept all invocations of Hot Nodes (functions) and actual parameters (line 34).
2. Lookup any function call within the Hot Node Cache (line 37-39).
3. If match is found (hot node with same parameters) do not invoke AJAX call and reuse existing content instead (line 41).

The effect of this optimization on the YouTube application is the following: although the crawler invokes all events, it will avoid invoking twice `next` from page 2 and `previous` from page 3 in order to get to the same state, just because they refer to the same underlying AJAX call (i.e., the function `getURLXMLResponseAndFillDiv` with the same parameters). The events which allow direct jumps to page 2 are also intercepted from any page. The effect is that the number of AJAX calls decreases, as we will show in Section 21.

### 19.4 Complexity Analysis

The complexity of our proposed Approach can be summarized as follows: A Naïve Crawling Algorithm operates in an exponential complexity in terms of the number of AJAX events per page since all possible graph connections in the application model itself must be considered. The Enhanced Crawling Algorithm reduces the complexity of processing to $O(E \times S)$ where $E$ is the average number of events per page and $S$ is the number of states. The size of the application model, however, is dominated by $O(S)$ where $S$ is the number of states.

<table>
<thead>
<tr>
<th>Hot Node</th>
<th>Parameters</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td><a href="http://youtube/v">http://youtube/v</a> = abc, 2</td>
<td>(commentpage2)</td>
</tr>
<tr>
<td>A</td>
<td><a href="http://youtube/v">http://youtube/v</a> = abc, 3</td>
<td>(commentpage3)</td>
</tr>
<tr>
<td>A</td>
<td><a href="http://youtube/v">http://youtube/v</a> = abc, 4</td>
<td>(commentpage4)</td>
</tr>
<tr>
<td>A</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td><a href="http://youtube/v">http://youtube/v</a> = xyz, 1</td>
<td>(commentpage1)</td>
</tr>
<tr>
<td>A</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 19.4: The Hot Node Cache
Algorithm 19.3.1 Breadth-first Heuristic AJAX Crawling Algorithm

1: Cache hotNodesCache = {}
2: 
3: Function init(url) { . . . } end Function
4: 
5: Function crawl(State s) {
6: for all Event e ∈ s do
7: manageFunction(e.function)
8: if dom.hasChanged() then
9: State newState = new State(dom)
10: if appModel.contains(newState) then
11: newState = appMode.get(newState)
12: end if
13: Transition t = new Transition(e.source, e.trigger, e.action*, e.modif*)
14: appModel.add(t, s, newState)
15: appModel.rollback(t)
16: end if
17: for all Transition t ∈ (s, s1) do
18: Crawl s1 {Breadth-first traversal of reachable states}
19: end for
20: end Function
21: 
22: Function invokeFunction(Function f) {
23: Statement[] statements = getStatements(e)
24: for all Statement stmt ∈ statements do
25: if stmt is function then
26: callStack.push(stmt, e.args)
27: result = manageFunction(stmt)
28: else
29: Execute Statement e
30: end if
31: end for
32: end Function
33: 
34: Function manageFunction(Function f) {
35: Statement[] statements = getStatements(f)
36: for all Statement s ∈ statements do
37: if s is AJAXCall then
38: Function topEntry = callStack.top()
39: if not hotNodeCache.contains(topEntry.function, topEntry.arguments) then
40: hotNodes = hotNodes U topEntry.function
41: result = callAJAX(entry.function)
42: insert (topEntry.function, topEntry.arguments, result)
43: into hotNodeCache
44: else
45: result = hotNodeCache.lookup(topEntry.function, topEntry.arguments)
46: end if
47: end if
48: if s is function then
49: result = invokeFunction(e)
50: else
51: for all Function f ∈ s do
52: s.f = manageFunction(f, f.args)
53: end for
54: Execute statement s
55: end if
56: end Function
19.5 Simplifying Assumptions

As being a first step in the direction of AJAX Crawling, in the proposed algorithm and model we made the following assumptions that we mention below.

- **Snapshot Isolation**: we assume that an application does not change during crawling. This is realistic since crawling must anyway be done regularly, but not continuously, even for the most updated sites. In YouTube, it is not relevant to crawl comments every second.

- **No Forms**: a lot of AJAX applications (such as Google Suggest [71]) use forms to infer actions from the user, dynamically. We do not deal with AJAX parts that require user inputting data in forms.

- **No update events**: We explicitly avoid triggering update events, such as Delete buttons. In case of crawling an authenticated user’s Yahoo! Mail or GMail (well-known) AJAX clients, this could mean deleting E-mails from the user’s Inbox.

- **State explosion**: Google Maps for example has an infinite amount of states (i.e., as many as there are pixels on the map). Still, an automated crawling is viable, by limiting the amount of automatically indexed states, and this is also the approach that we take. We predict that in the future, AJAX Web Sites will provide a robots.txt file with information on the possible granularity of search on their pages.

- **No Image-based retrieval**: The states in applications such as Google Maps [68] are not text-based, but image-based. We limit ourselves to text-based retrieval.
Chapter 20

An AJAX Search Engine

The purpose of this work is to crawl AJAX content with the goal of improved search results. In order to be able to provide this functionality, the crawler is integrated in a complete architecture similar to that of a traditional search engine, as shown in Figure 20.1. The components are described below.

20.1 AJAX Crawler

The main contribution of this work, the AJAX Crawler builds the AJAX model of an AJAX Web Site and AJAX pages. In order to do this, it implements the crawling Algorithm of Sections 18 and 19. The AJAX Models are then used to build the index.

20.2 Indexing

Indexing is an operation which starts from the model of the AJAX Site and builds the physical inverted file. In traditional information retrieval, an index is an inverted file[16] containing information about the documents in which the keywords occur. The result of the Indexing will be used during query processing, in order to return results. As opposed to traditional index processing, in our case a result is an URI and a state. As an example, the inverted file for the YouTube Application is presented in Table 20.1. The enhanced inverted file contains a link to the Web page containing the words (in this case, the URLs are two videos of Morcheeba), and to the state (i.e., the comment page) containing the word. The score is computed based on the number of occurrences of the word in the state.

![Figure 20.1: Architecture of an AJAX Search Engine.](image)
20.3 Query Processing

As presented in the Introduction, when searching an AJAX application, a user is interested in obtaining the states in which a certain keyword appears. Furthermore, the user might be interested in the DOM element in which the desired text resides. We present the evaluation of simple keyword queries and of conjunction queries.

20.3.1 Processing Simple Keyword Queries

The index constructed in Section 20.2 can be used to extract this information as shown in Table 20.2. It shows the results of the query: *morcheeba*. Each query returns the URI and the state(s) which contain the keywords. In this case, the first state of the second Morcheeba video is ranked higher, since the score in the index was two (i.e., the keyword appears twice in the state).

<table>
<thead>
<tr>
<th>Query</th>
<th>URI</th>
<th>State</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>morcheeba</td>
<td><a href="http://www.youtube.com/watch?v=Iv5JxME0js">www.youtube.com/watch?v=Iv5JxME0js</a></td>
<td>s1</td>
<td>0.9</td>
</tr>
<tr>
<td>morcheeba</td>
<td><a href="http://www.youtube.com/watch?v=w16JILSySWQ">www.youtube.com/watch?v=w16JILSySWQ</a></td>
<td>s1</td>
<td>0.7</td>
</tr>
<tr>
<td>morcheeba</td>
<td><a href="http://www.youtube.com/watch?v=w16JILSySWQ">www.youtube.com/watch?v=w16JILSySWQ</a></td>
<td>s2</td>
<td>0.7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 20.2: Query Processing on the AJAX news application.

20.3.2 Processing Conjunctions

A query composed of multiple keywords returns all states and elements where all keywords occur. Conjunctions are computed as a merge between the individual posting lists of the corresponding keywords, sorted on URL and state. First, entries are compatible if the URLs are compatible, then if the States are identical.

As an example, shown in Figure 20.2 let’s take the query Q3 from the Introduction: *Morcheeba singer*. This will result in the conjunction between two posting lists. The posting lists of keyword *Morcheeba*...
(already shown in Table 20.2) and of keyword *singer* are presented and merged in the first row of Figure 20.2. The second row indicates the second phase of Processing Conjunctions, and shows how incompatible URIs, as well as incompatible states under the same URI are eliminated from the result. The result of Q3 is the tuple \((URL_1, s_2)\), corresponding to the second page of comments of the video presented in Section 16.

### 20.3.3 Ranking

This work focused on boolean retrieval, with the purpose of increasing Recall. Therefore, ranking is not the main focus of this work. We mention briefly that we implemented a ranking algorithm based on both traditional ranking mechanisms (i.e., *tf/idf*), positional information, and application based information (i.e., distance in the transition graph). The ranking mechanisms are presented in Part IV.

### 20.4 Result Aggregation

The purpose of the Result Aggregation phase is to present results to the user. In both traditional and AJAX Search, results are links to application states. As opposed to traditional search however, a link in a function from the application to the application states. Just as traditional IR returns the original Web pages, states must be reconstructed. In order to present to the user the state which contains that value (and not only a link to it), the following algorithm is used:

1. Extract from the page model the path from the initial state to the desired state.
2. Construct the DOM of the initial state.
3. Invoke all annotated events to the desired state and construct the DOM of the generated state.
4. Present the generated DOM in a browser.

This process allows the browser to continue processing the page starting from the desired state, since the state is also preserved (e.g., the Javascript variables).

### Phase 1: Merge.

<table>
<thead>
<tr>
<th>List : &quot;Morcheeba&quot;</th>
<th>List 2: &quot;singer&quot;</th>
<th>Result: &quot;Morcheeba&quot; ∧ &quot;singer&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL1, s1</td>
<td>URL1, s1</td>
<td>URL1, s1</td>
</tr>
<tr>
<td>URL1, s2</td>
<td>URL1, (s1, s2)</td>
<td>(URL1, URL2)</td>
</tr>
</tbody>
</table>

### Phase 2: Eliminate Incompatibilities.

Figure 20.2: Processing Conjunctions.
20.5 Parallelization

Because crawling AJAX faces the difficulty of not being able to really cache dynamic Web content, except for the heuristics discussed in Section 19, network connections must continuously be created. This drastically increases the crawling time, as we will also show in Section 21. The situation can however be improved through parallelization. Figure 20.3 shows the parallel architecture of the AJAX Search Engine.

A precrawler is used to build the traditional, linked-based Web site structure (as presented in Section 17). This results, for example, in a list of videos to crawl and the references between them. The total list of URLs of AJAX Web pages (i.e., videos) is then partitioned and supplied to a set of parallel crawlers. The parallelization is complete since crawling an AJAX Web page is completely independent of crawling another AJAX Web page. Each crawler applies the crawling algorithm of Sections 18 and 19, and builds for each crawled page the AJAX Model (i.e., the transition graph). More indexes are then built from the disjunct sets of AJAX Models. Query processing is then performed by query shipping, computing the results from each Index, as explained in Section 20.3, and then performing a merge of the individual results from each index, returning the final result list to the client.
Chapter 21

Experimental Results

This chapter presented a model and implementation of an AJAX Crawler. We implemented a prototype version of the AJAX Crawler and we applied it to a subset of YouTube. YouTube uses AJAX in order to display the comment pages for each video, so that the video is not refreshed. Crawling video comments allows to retrieve videos based on other keywords than just those from the video title, therefore increasing search quality. We crawled the AJAX part, and compared AJAX crawling and search with traditional crawling, which ignores this content. The goals of the experiments were to:

1. Evaluate the search result quality when AJAX content is crawled.
2. Evaluate the performance overhead of AJAX crawl over Traditional crawl.
3. Determine the good threshold between gain in search result and performance decrease.

21.1 Experimental setup

We used real data set for evaluating the impact of AJAX Search and two algorithm flavors.

21.1.1 YouTube Datasets.

The experiments were performed on YouTube [117], and in particular on a subset of 10000-page videos of YouTube (called YouTube10000). The videos have been chosen by starting the crawling on the video of Section 16 and continuing on the related videos until the desired number has been reached. Each video and its comments have been crawled using AJAX. For performance reasons, we restricted the number of comment pages that we retrieved to ten (e.g., ten comment pages per video). The characteristics of the data are displayed in Table 21.1.

AJAX Crawling means determining the overhead of crawling the additional AJAX content in applications. In case of YouTube, this reflects in a variable number of comment pages belonging to the main video page. From the 10000 crawled videos, we extracted statistics related to the number of videos, comments, and number of pages per video.
Table 21.1: You Tube Statistics.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of pages</td>
<td>10000</td>
</tr>
<tr>
<td>No. of states</td>
<td>41572</td>
</tr>
<tr>
<td>No. of events</td>
<td>187980</td>
</tr>
<tr>
<td>No. of events per page</td>
<td>avg. 4</td>
</tr>
</tbody>
</table>

Figure 21.1: Distribution of YouTube videos based on number of comment pages.

Figure 21.1 shows the distribution of videos with a given number of comment pages (i.e., AJAX states). Most videos have, indeed, just a single page of comments. However, there are enough videos with a lot more than one page, and crawling them leads to better search results, as we will show below. This is also what motivated this work in the first place.

The number of additional pages themselves shows just one dimension on which the amount of crawled content increases. The processing time of any crawler will be influenced by the number of actual events that must be invoked on the page in order to fetch all the AJAX content, and which result in network connection times. Figure 21.2 shows that especially the number of events grows at least polynomially, and mostly affects the overhead that needs to be handled by an AJAX Crawler in YouTube.

Figure 21.2: Number of Pages, States and Events in YouTube.

In any YouTube page, there is an average of four events that can be invoked by a user (next, prev and direct “jumps” to the immediately few previous and next pages), as it was shown in Figure 16.1. This leads to a lot more events than pages, including duplicate events. They will be avoided using the Hot Node policy as we will show below.
21.1.2 Algorithms.

In order to evaluate the impact of AJAX Search, more flavors of crawling have been used.

1. Traditional Crawling. We configured the AJAXSearch to read just the first state of each YouTube video. In case of YouTube, this means the first comment page (i.e., 10 comments). Traditional crawling reads the same content that is obtained when JavaScript is disabled in a user’s browser.

2. AJAX Non-Cached. This is the basic AJAX crawler which reads the AJAX content of a Web Page, and uses the algorithms proposed in Section 18, but without heuristics. Therefore, this is a non-optimized AJAX Search engine, but with full capabilities of client-side code and triggers events from the page in order to build the AJAX Page Model of Section 17.

3. AJAX Cached. This is the full AJAX crawler. It uses the optimization from Section 19. This is the full-fledged, optimized, AJAX Crawler.

4. AJAX Parallel. We parallelized the AJAX Crawler on four nodes, as shown in Section 20.5. While all previous configurations ran in sequential mode (Figure 20.1), this version of the crawler runs in parallel mode (Figure 20.3) and uses the Hot Node approach for better performance.

When search capabilities are used, we implemented the Indexing and Query Processing capabilities of Section 20. The Crawling process(es) stored the AJAX model and disk. The Index was constructed incrementally [29] from the application model, and is fully maintained in memory. We ran the experiments on a Intel Xeon 3050 2.13GHz, 2 MB L2 Cache, Dualcore with 4 GB RAM (with ECC, DDR-2 533 Mhz), 1 x 250 GB S-ATA 7200 rpm hard disk, 1 x 500 GB S-ATA 7200 rpm hard disk running Windows 2003 Server. More capabilities of the framework, which go beyond the scope of this work, are described in [54]. Javascript code was analyzed using the COBRA Toolkit [40] toolkit.

21.2 Crawling Time

This section addresses the crawling time induced by crawling AJAX content in YouTube, as opposed to just traditional crawling. We also present cumulative results based on the overhead and gains induced by different methods.

<table>
<thead>
<tr>
<th></th>
<th>Trad.</th>
<th>AJAX Non-cached</th>
<th>AJAX Cached</th>
<th>AJAX Parallel</th>
<th>AJAX Cached/Trad.</th>
<th>AJAX Non-Cached/AJAX Cached</th>
<th>AJAX Cached/AJAX Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (secs.)</td>
<td>24556.43</td>
<td>319693.08</td>
<td>244016.36</td>
<td>64633.00</td>
<td>x9.94</td>
<td>x1.31</td>
<td>x3.78</td>
</tr>
<tr>
<td>Network time (secs.)</td>
<td>18708.90</td>
<td>159361.91</td>
<td>48959.16</td>
<td>15092.58</td>
<td>x8.52</td>
<td>x3.25</td>
<td>x3.24</td>
</tr>
</tbody>
</table>

Table 21.2: Crawling Time (secs.) for Trad. and AJAX Crawling.

<table>
<thead>
<tr>
<th></th>
<th>Trad.</th>
<th>AJAX Non-cached</th>
<th>AJAX Cached</th>
<th>AJAX Parallel</th>
<th>AJAX Cached/Trad.</th>
<th>AJAX Non-Cached/AJAX Cached</th>
<th>AJAX Cached/AJAX Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items/second</td>
<td>0.41</td>
<td>0.13</td>
<td>0.17</td>
<td>0.64</td>
<td>x0.41</td>
<td>x0.76</td>
<td>x0.27</td>
</tr>
</tbody>
</table>

Table 21.3: Crawling Throughput (states or pages/second) for Trad. and AJAX Crawling.

The crawling times obtained by all methods and optimizations are summarized in Table 21.2. The crawling times were obtained for crawling the YouTube subset of 10000 pages, in case of Traditional, AJAX
Non-Cached, AJAX Cached and AJAX Parallel Crawling. In Traditional crawling, the first page of the video was read, and also the first page of comments (loaded by default, without Javascript). In AJAX Non-Cached all Javascript code is enabled, the first page of comments is read, the next and prev events are invoked in order to load the additional comment pages from the server; In AJAX Cached the Hot Node approach is used in order to avoid unnecessary invocations to the server. The AJAX Parallel Crawling parallelized crawling on four machines and used caching. More conclusions can be driven on the following points:

**AJAX Crawling is Expensive.** The Crawling Time for 10000 pages with AJAX is in the range of about 68 hours. Furthermore, Figure 21.3 shows the distribution of YouTube videos based on the crawling time of each page. Most of pages have a reasonable crawling time, but for approximately 1500 and of pages are crawled between 15 and 20 seconds, still a considerable amount at the scale of the Web.

**AJAX Crawl induces an overhead over Traditional Crawl.** AJAX Crawling non-optimized takes about thirteen as much as Traditional Crawling due to the overhead of processing the application model. AJAX Cached however is about nine times slower than Traditional AJAX which just indexes one state for a maximum of ten crawled states per video.

**Network Time is 50%.** Figure 21.4 displays the crawling time for an increasing number of crawled videos and the corresponding network time. This shows that half of the time needed during Crawling is taken by Network Time (50%). Furthermore, as already shown in Table 21.2 just the network of Non-Cached version is a factor of 3.25 higher than in the AJAX Cached version of the Crawler. This also underlines the importance of applying the Hot Node Optimization on this important time of the time slide.

**The Hot Node Heuristics is effective.** The heuristic approach of the Hot Nodes causes a 1.31 factor
of improvement in crawling time as opposed to the Non-Cached Approach. This is due to the reduced number of AJAX Calls per page as opposed to the Non-Cached Approach. In case of YouTube, this results in a lower number of clicks on next or prev links, if a direct click on a link which leads to the same page number has already been done in the crawler. Figure 21.5 displays the number of AJAX events from the crawled pages for which a network call was needed for an increasing number of crawled videos (one to one hundred) and shows that using the Hot Node approach causes a reduction with a factor of four of the needed network calls.

Figure 21.5: Number of AJAX Events resulting in AJAX Calls with and without caching policy.

Parallelization is Possible and Effective. The parallel version of the crawler is an additional optimization we applied, as explained in Section 20.5. With four processes running on different partitions of the URL list, the running time decreases almost perfectly by 25%, as opposed to the AJAX Non-Parallel version. As explained in Section 20.5, parallelization in case of AJAX can be achieved because crawling each AJAX page and its corresponding AJAX states is independent of the other AJAX pages.

Crawling Throughput. It is interesting to compute the crawling throughput in pages per second and states per second. In particular, it is important to mention the overhead of AJAX Crawling in terms of number of indexed pages. Table 21.3 shows the page and state throughput for all crawling methods. It can be seen that the throughput of AJAX Crawling of Youtube10000 is a factor of 10 slower than the throughput of Traditional Crawling. It translates in a time of 2.45 seconds per page for AJAX, as opposed to 0.25 seconds per traditional page. The basic crawling time in both cases is caused by the parsing libraries we used, and can be optimized.

21.3 Size of Application Model

As shown in Section 17, a contribution of this work is the new model of AJAX Web Sites. The model is an annotated automaton, useful for describing and restoring transitions in the AJAX Web site. Crawling AJAX Applications is therefore influenced by the maintenance and size of the application model. Table 21.4 shows the size of the application model in case of traditional crawling and of the AJAX Crawling, after is has been serialized on disk. Of course, this shows a factor of four overhead in terms of space as opposed to traditional AJAX Sites. The Size of the AJAX Model is mostly influenced by the size of the DOM which is 1505MB in case of AJAX Crawl for 10000 web sites.

The model just needs to be maintained during crawling, is used after query processing in order to restore a certain state. Since the model is not updated after crawling, but needs to be quickly read, compression

![Number of AJAX Events resulting in network requests](image-url)
Chapter 21. Experimental Results

Table 21.4: Application Model Size (MB) for Trad. and AJAX Crawling.

<table>
<thead>
<tr>
<th></th>
<th>Trad.</th>
<th>AJAX (all methods)</th>
<th>AJAX/Trad.</th>
</tr>
</thead>
<tbody>
<tr>
<td>App. Model Size</td>
<td>800</td>
<td>3192</td>
<td>x3.99</td>
</tr>
<tr>
<td>(MB)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOM Size (MB)</td>
<td>641</td>
<td>1505</td>
<td>x2.34</td>
</tr>
</tbody>
</table>

Techniques are left as an avenue for future work.

21.4 Query Processing Time

One goal of this work was also to motivate the investment in Crawling AJAX by proving the improvement in quality of results. We built the Index as shown in Section 20.2 and we performed queries on it, as in Section 20.3.

21.4.1 Queries

The queries we performed are taken from the most popular YouTube queries, retrieved from [118]. A snippet of these queries is shown in Table 21.5. There are 100 queries in total. The Table shows a sample of nine queries in order of cardinality. We specify the number of videos where the query appear in the first comment page, and the total number of comments where the keywords appear. ¹.

<table>
<thead>
<tr>
<th>ID</th>
<th>Query</th>
<th>Occurrences First Page</th>
<th>Occurrences All Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>youtube</td>
<td>500</td>
<td>2405</td>
</tr>
<tr>
<td>Q2</td>
<td>with you</td>
<td>496</td>
<td>2401</td>
</tr>
<tr>
<td>Q3</td>
<td>wow</td>
<td>56</td>
<td>321</td>
</tr>
<tr>
<td>Q4</td>
<td>funny</td>
<td>56</td>
<td>269</td>
</tr>
<tr>
<td>Q5</td>
<td>dance</td>
<td>40</td>
<td>244</td>
</tr>
<tr>
<td>Q6</td>
<td>kiss</td>
<td>16</td>
<td>120</td>
</tr>
<tr>
<td>Q7</td>
<td>low</td>
<td>16</td>
<td>91</td>
</tr>
<tr>
<td>Q8</td>
<td>fight</td>
<td>20</td>
<td>78</td>
</tr>
<tr>
<td>Q9</td>
<td>akon</td>
<td>6</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 21.5: YouTube Queries.

21.4.2 Query Processing Time

We ran the queries on the 500 page-index. The query processing times for the individual queries of Table 21.5 on the index containing 500 pages are shown in Table 21.6. Query processing times in case of AJAX Search are obviously larger than in the case of traditional search which just returns a result if the keywords are found in the first page.

¹for the numbers we used a subset of the first 500 videos
21.5. Precision

The throughput in terms of pages/second in case of traditional and states/second in case of AJAX Crawl, varies much for individual queries. Generally traditional search offers slightly better page throughput, although for a much smaller number of results.

<table>
<thead>
<tr>
<th>ID</th>
<th>Query</th>
<th>Query Time Trad.(msecs)</th>
<th>Query Time AJAX (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>youtube</td>
<td>1984.3</td>
<td>69054.9</td>
</tr>
<tr>
<td>Q2</td>
<td>with you</td>
<td>1009.4</td>
<td>38196.1</td>
</tr>
<tr>
<td>Q3</td>
<td>wow</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Q4</td>
<td>funny</td>
<td>1.5</td>
<td>14.1</td>
</tr>
<tr>
<td>Q5</td>
<td>dance</td>
<td>1.6</td>
<td>39</td>
</tr>
<tr>
<td>Q6</td>
<td>kiss</td>
<td>0</td>
<td>6.2</td>
</tr>
<tr>
<td>Q7</td>
<td>low</td>
<td>1.5</td>
<td>3.2</td>
</tr>
<tr>
<td>Q8</td>
<td>fight</td>
<td>0</td>
<td>1.6</td>
</tr>
<tr>
<td>Q9</td>
<td>akon</td>
<td>0</td>
<td>7.8</td>
</tr>
</tbody>
</table>

Table 21.6: Query Processing Times (msecs) on YouTube.

The goal of AJAX Crawling is to increase search result quality by crawling more than traditional search content. We have not performed an exhaustive evaluation of the precision increase, but we include here anecdotal evidence for the value of crawling AJAX content. Table 21.7 shows more queries which only return the correct video if the content of the comments is included, in addition to the original queries from Section 16. Hidden information can therefore be retrieved, such as a the soundtrack of a certain movie (P1), a video based on the shooting location (P2), a song in another language (P3), the new lead singer in the new video of a band (P4).

<table>
<thead>
<tr>
<th>ID</th>
<th>Query</th>
<th>Video</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>prime start song</td>
<td>uKJeLG8-M5I</td>
<td>RJD2 “Ghostwriter” (First song from the movie “Prime”)</td>
</tr>
<tr>
<td>P2</td>
<td>corrs singapore video</td>
<td>dJR20hstdS4</td>
<td>The Corrs “Dreams” (Video shot in Singapore)</td>
</tr>
<tr>
<td>P3</td>
<td>maria tanase longing song</td>
<td>sq_pqQini6U</td>
<td>Maria Tanase “Doina din Maramures” (Song in another language)</td>
</tr>
<tr>
<td>P4</td>
<td>morcheeba mysterious video</td>
<td>w16JjLSySWQ</td>
<td>Morcheeba “Enjoy the Ride” (A particular video of the band “Morcheeba”)</td>
</tr>
<tr>
<td>P5</td>
<td>morcheeba enjoy the ride singer</td>
<td>w16JjLSySWQ</td>
<td>Morcheeba “Enjoy the Ride” (Comment page number two)</td>
</tr>
</tbody>
</table>

Table 21.7: Queries reflecting the increase in precision with AJAX Search.

---

\[^2\]The complete URL of the song can be obtained is by appending the prefix `http : //www.youtube.com/watch?v=` to the text in the Video column.
21.6 Disadvantages of a full non-AJAX Approach

An alternative to using AJAX in a Web site is to expose all AJAX states together as content, but we can evaluate here the quality of this solution in terms of precision. For example, YouTube offers a link in which it includes all user comments together. We evaluate how the Precision and Recall vary if this other extreme is used (e.g., all data is exposed to the user). Precision decreases of course compared to the approach where only the correct comments pages are displayed.

<table>
<thead>
<tr>
<th></th>
<th>AJAX</th>
<th>Condensed AJAX Pages (all methods)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision (Videos)</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>Recall (Videos)</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>Precision (States)</td>
<td>1</td>
<td>0.65</td>
</tr>
<tr>
<td>Recall (States)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 21.8: Recall for Trad. and AJAX Crawling.

21.7 Recall

The effect of quality in AJAX Crawling can be expressed using Recall. For each query we evaluated the number of videos returned by just using the traditional approach, as opposed to the total number of videos returned in the AJAX Crawl approach, when also comment pages are taken into account during searches. Table 21.9 shows these results.

<table>
<thead>
<tr>
<th></th>
<th>Trad.</th>
<th>AJAX (all methods)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall</td>
<td>0.14</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 21.9: Recall for Trad. and AJAX Crawling.

21.8 Partial Crawling. Setting Crawling Threshold.

Crawling AJAX brings better results, with the price of performance. We aim to establish a tradeoff between the crawling performance and the result quality, and indicate the possible points based on which this tradeoff can be achieved.

For the purpose of establishing this tradeoff, we executed a partial crawling of the YouTube Website, by keeping the number of pages constant (i.e., 10000), and varied the maximum amount of crawled AJAX states per page from one to ten.

Recall. Partial Crawling has the effect that the more AJAX States are retrieved (i.e., comment pages in YouTube), the worse the computed recall of traditional search gets. Another hint on setting the crawling threshold is to crawl as many states per AJAX page until the recall difference between traditional and AJAX Crawl reaches a “good enough” level. We consider all results returned by AJAX Search as the
relevant results (i.e., the desired ones) and we assume boolean retrieval. Figure 21.6 shows the recall of traditional search and AJAX search (which is constant, equal to one). Also in this case, crawling between a maximum of four to five states is already enough to gain a 0.7 increase in Recall over Traditional Search. This result is of most importance for current search engine implementors, since it shows that it is enough to crawl a relative small amount of AJAX content in order to get an important increase in Recall.

![Recall for Trad. and AJAX Crawling](image1)

**Figure 21.6:** Recall for Traditional and AJAX Search in case of Partial Crawling.

**Application Model Size.** Although not the most important factor of overhead, but specific to the AJAX indexing problem, is the growth of the size of the AJAX application model size compared to traditional model of an AJAX Web site.

![Application Model Size and DOM Size(MB) for increasing number of crawled states](image2)

**Figure 21.7:** Application Model Size(MB) in Case of Partial Crawling.

Figure 21.7 displays the size of the application model size and of the DOM in case of partial crawling. The values for one (1) correspond to traditional crawling. Setting the threshold to 4 crawled states per video ensure a size of 2GB for YouTube10000 and a growth factor of 2.5 over traditional search which is reasonable and conforms to a good threshold in case of query throughput or recall.
Chapter 22

Discussion: Integrating AJAX Crawl into Existing Search Engines

Currently search engines do not use AJAX Crawling because naïve crawling is expensive. We have shown that AJAX Crawling is possible although it does induce a real overhead over traditional search. There are however three points which encourage existing search engines to incorporate AJAX Search: First, the AJAX model extends current Web model. The existing model of the Web is maintained “as is” and is just enhanced in AJAX Crawl. The extensions can also be dropped, reducing the model to that of traditional Web. Second, indexing is based on traditional methods. The index structures that we used are also extensions of traditional inverted files. This makes the implementation into existing search engines straightforward. Third, a Performance/Quality benefit can be obtained. The stronger assumptions of commercial search engines is that crawling AJAX is expensive. We have shown that the Hot Node approach improves the naïve approach through the process of memoization [63]. Furthermore, we have experimentally shown that a very good performance/quality benefit can be achieved by just partially Crawling the AJAX Content. This tradeoff, along with using parallelization, successfully addresses the obvious demand for scalability. Finally, the techniques used for AJAX can also be applied for other types of Rich Internet Application such as Flash, minimally addressed by commercial search engines.
Chapter 23

Demonstration

As shown in Chapter 16, AJAX and Rich Internet Applications are increasingly frequent on the Web (in YouTube, Amazon, GMail, Yahoo!Mail) and on mobile devices, offering a high degree of interactivity to the user. Current search engines however do not index the AJAX content for two reasons: (i) there are no corresponding tools, and (ii) naïve crawling is expensive.

Among others, the challenges, as opposed to traditional search engines, are: automatically identifying states by triggering events, efficiently crawling application states, avoiding the invocation of potentially very numerous events, scalability in the number of events, duplicate elimination of states, result presentation and aggregation, ranking.

The demo shows that this AJAX Search engine can be built and presents it. Just as a traditional search engine, it contains a crawler, indexer and query processor, as shown in Figure 20.1, but the components are are adapted to AJAX. Our first implementation focuses on AJAX sites without user input, thus avoiding the already studied domain of the “hidden Web”. We apply the demo on a real application and showcases challenges and solutions. Furthermore, we show how this automatic crawling and search can be further improved with user-defined rules on the data (such as in robots.txt), for even more accurate search results.

23.1 The AJAX Applications

In our demo we apply the crawler and show the use cases on the two applications described below.

Application 1: AJAXNews. The news reader AJAXNews is a custom-built application residing at [5]. It has several pieces of news loaded from the server and changed through two buttons (next and previous) or through a side-menu as shown in Figure 23.1. This application can be used for testing (since it can be customized), and for showing the quality of ranking. Furthermore it illustrates how grouping strategies for the result can be used in order to avoid generating a large number of superfluous states.

Application 2: Yahoo! Mail. We chose Yahoo!Mail [116] as the first test on a real AJAX Application. Yahoo! Mail is an E-mail client. Since it uses text, it is easier to approach than, for example, Google Maps. The E-mail client also displays the calendar of the user and news along with the messages, using AJAX. Yahoo! Mail has a custom search engine but only on messages. Crawling this application is, how-
23. Demonstration

Figure 23.1: AJAXNews: What does a traditional engine see?

ever, cumbersome because of frequent application changes, and because of heavy use of Javascript, requiring us to extend the openly available [40] framework that we used for implementation. Yahoo! Mail will be used in order to show the behavior of an AJAXSearch on a complex application. The possibility for the application developer to enhance search result quality using personalized rules will also be demonstrated.

23.2 Demo Scenarios

We showcase AJAXSearch on the two AJAX applications.

Scenario 1: Crawling.

We crawl the AJAXNews Application. During crawling, the user can visualize how the different states are retrieved. This is shown in Figure 23.2. On the left, the DOM tree of the current application state can be inspected. On top, the individual Javascript events can also be invoked manually by the user to generate states. The Javascript variables can also be seen at each state. At the end, the Transition Graph is displayed in a textual format and can be stored on disk for later retrieval. During crawling, similar states are detecting using a hash on the content of the DOM. An additional point is that the application model can grow indefinitely if events can always be invoked. However, the AJAXNews application always ends. Otherwise, a maximum limit on the number of states can be set.

Scenario 2: Indexing.

We generate the index by reading the application model and building an inverted file. The statistics of the index are displayed in the status bar of the application. The Index can now be queried, as shown in the next scenario.
23.2. Demo Scenarios

Figure 23.2: Crawling an AJAX Application. Displaying the DOM.

Scenario 3: AJAXSearch.

We use either AJAXNews or Yahoo!Mail. The user can run a set of queries on the AJAXNews and the Yahoo!Mail application.

In this scenario, the user can see how results are returned in terms of tuples \((URL, state, element)\) and not just in terms of \((URL)\) as in traditional search. If two keywords occur in the same state but in different elements, the Least Common Ancestor (LCA) of the two elements is used. The application in action is shown in Figure 23.3. The user has several options:

- Read the paragraph abstract.
- Get the cached version of the state.
- Click on the link. Reconstitute the application state.

The first two steps are also common in traditional web search. However, the last one is specific to AJAXSearch. An AJAX application has a context defined by Javascript variables. Our framework is able to recreate the state by traversing the application model built during the crawling phase, as shown in Figure 17.1. However, the time needed to fetch the result increases for each of the above options. Reconstituting the application state is the most expensive option, but the advantage for the user is that she can continue the navigation from that point.

Scenario 4: Traditional Search vs. AJAXSearch.

For this scenario we use the AJAXNews Application. The purpose is to compare AJAXSearch with Google (a non-AJAX-aware search engine). There are two levels of interactivity for the application which the user can choose:

- **No Javascript-aware.** We let the application be indexed by Google.
- **AJAXSearch.** Full AJAX capabilities; correct results are returned.
We run a query on the AJAXNews application using Google:

```
newsreadersite : www.giannifrey.com
```

In case of the news application, the non Javascript-aware version (or Google) causes a surprise to the user since only the following content is indexed:

```
<h1>Newsreader</h1>
<table ...>...loading data ...</table>
```

The reason is that the first page is also loaded using Javascript. **Google does not index the AJAXNews application correctly**, but AJAXSearch does.

**Scenario 5: Result Aggregation.**

We can run this scenario on either the News application or on the Yahoo!Mail application. Search results can be very granular, since there are many events in the application, and second because we also return locations of an element in the page. There are therefore several grouping strategies, specific to AJAX, that we can apply. The user can select one the following options:

- **No group.** This is the default option. All occurrences of \(\langle URL, state, element, position\rangle\) are returned.
- **Group by state.** More occurrences of the keywords can appear in the same state, in different parts of the page structure. Returning the elements is relevant because some parts may be more or less important in the applications. For usability however, they can be aggregated and only the state is returned.
- **Group by location.** In case of occurrences in many states but in the same visual location, it is advisable to return an aggregated result based on this common location. For example, the news header appears in all states. Returning all states may be daunting for the user. This allows the user to see which part of the page contains most occurrences.

**Scenario 6: State Explosion.**

We show an example when crawling states automatically leads to an explosion in the number of states. In applications such as Yahoo!Mail, several independent events may be invoked, such as those for displaying E-mails and those for showing the user’s calendar. Both are displayed using independent next/previous events. We can also illustrate this on the AJAX news application shown in Figure 23.4, enhanced with calendar events (application at [4]). Yahoo! Mail is shown in Figure 23.5. The total number of states is then a cartesian product, as in Figure 23.6.

Searching these applications brings the following results: each matching news item appears as many times as there are events in the calendar. We can partially solve this problem at runtime during grouping, but this does not avoid generating a large application model, which causes poor crawling and indexing times.

**Scenario 7: Personalization.**

We can prevent indexing many states at crawling by letting the application developer intervene. By knowing the application, she can influence the number of indexed states by writing rules.
23.2. Demo Scenarios

Figure 23.3: AJAXSearch in action. Results are links to application states.

Figure 23.4: Application with two sets of independent events.

**Example: User Defined Rule.** A rule specifies a correspondence between the pieces of structure in the model which cause a join in the states. For example, the following rule:

```
//message/date < - > //calendar//event//date
```

Figure 23.5: Personalization: Correlations in the application model increase search precision and crawling performance.
specifies that the date of the E-mail message is the same as the date of the event in the calendar. The effect in functionality is that the user can search for messages on a given date (e.g., the messages during “VLDB”). This is also shown in Figure 23.5. In this scenario, the user can crawl, index and search the extended AJAXNews application and/or Yahoo!Mail. The index will be smaller and only the matching states are returned at query processing time.

At the data level, this is an **outer join** between the states, based on the given condition, i.e., all states which contain the two pieces of information are considered as one if they share the same value, or discarded from the final application model and from index, if the values are not the same. All states not containing the value are still joined with the other states. Advantages: simplicity, minimal intervention from the application provider and good quality results.
Chapter 24

Related Work

Our work is new as what concerns crawling AJAX Web Applications. Nevertheless, there are parallels with work form the database, IR and DB-IR communities.

**AJAX Crawling.** We are the first work to list an automatic AJAX Crawling Algorithm and optimization. A state machine was used for modeling AJAX in [91] and constructed using a diff-based approach, but manual modeling was used by the authors in extensions such as [89]. Software engineering studies the extreme problem of a Web site being merged into a single AJAX page [92]. Giving hints to our AJAX crawler is also possible, as shown in [54]. The Hot Node approach is a memoization technique, also encountered in the context of XQuery in [63].

**Duplicate elimination.** There are two kinds of duplicate cases during crawling: syntactic and semantic duplicates. The syntactic duplicates are more than one page linking to the same page with the same URL. Traditional crawlers extract and store all static URLs in a look-up table used to avoids repeated access to the same URL. For the semantic duplicates, several algorithms exist to detect changes in tree structures: [36] or near-duplicate Web pages: [77], [88], Broder et al.’s shingling algorithm[27] and Charikar’s [35], [88] random projection. DustBuster (Different URLs with Similar Text) [17] tries to detect duplicate URLs using mining. This is not possible for AJAX application crawling. Traditional Web Crawler has been addressed in works such as [23]. Shah [101] crawls YouTube but for Data Mining purposes.

**Javascript Analysis.** The work of [86] analyzes Javascript and constructs a control flow, for testing Javascript applications. However, this model cannot help the crawler identify duplicate states in AJAX applications. Most Javascript works, such as [37], deal with detecting common spam redirects on Web pages.

**Application Models.** Colorful XML [78] encodes multiple views over the data in the same data structure; we are similar in the approach (i.e., multiple states in a single-page application), but their model lacks the notion of transition. The Transition Graph bears similarities with ActiveXML [1], a dynamic XML structure enhanced with dynamic calls. Active XML views are however always evolving, and offer just snapshots of current states as opposed to AJAX Crawling.

**Deep Web.** Because AJAX applications interact with the server using user input, crawling AJAX is related to Hidden Web [44], [98], [108], [113], [114]. AJAX Crawl does more than a hidden Web crawler, since it focuses on very granular interactions with the application and since it escapes the page paradigm commonly used in search.
Chapter 25

Summary

Crawling AJAX is a difficult problem, avoided by current search engines. The benefits reflect in improved search results. This work addresses AJAX Crawling pragmatically. and proposed an AJAX Crawler. The most difficult issues in crawling AJAX, duplicate detection and caching, have been addressed by the crawling algorithm and its optimizations based on Hot Nodes. The experiments on YouTube proved the benefit of crawling AJAX content and offered an insight on the performance/result quality threshold. Usually, a large number of states already leads to large performance overhead.

There are several avenues for future work, most address ways to decrease the number of AJAX content that needs to be crawled. The first one is crawling more current AJAX applications, such as Google Maps. A second one is to address forms in AJAX applications. Most AJAX applications allow user input. Combining AJAX Search and work on Deep Web can provide insight on which content is relevant for crawling. Furthermore, explicit rules provided by implementors of AJAX search engines could be seen as a modern version of robots.txt files that describe how to interpret the data found by the crawler. An approach as the one presented in Chapter 12 in the context of indexing Enterprise Application Data, which actually describes data interactions using rules, is complementary to this approach. The next one is crawling and personalization, i.e., focusing on a specific user’s or group’s interaction with the server. This would mean that only relevant states and events are crawled, improving both quality and performance. Finally, crawling AJAX can also be seen as a repetitive process, which can reduce the number of crawled events, by ignoring events which did not cause large changes in previous crawling sessions. These approaches are explained in Part IV.
Part III

Improving Hidden Web Crawling and Coverage using AJAX Suggestions
Chapter 26

Problem Statement

Chapter II has shown how an AJAX Crawler can be built in order to construct the enhanced model for AJAX Websites. AJAX web Sites poses several challenges to a crawler, both in performance and in functionality. In terms of performance, crawling challenges are the very high number of events and also by the possibility for different events to leading to the same states. The latter problem (i.e., duplicate elimination) has been addressed in Chapter II, but the former has been not. In terms of functionality, AJAX Crawling has been considered only for Web Sites without user input. Most AJAX Websites existing today are benefiting from user input, and actually use AJAX capabilities in order to enhance the interactivity of the Web sites.

This part of the work addresses AJAX Web Sites with user input. Crawling such a Website has been the purpose of Hidden Web Crawlers [98]. There are two important issues in Hidden Web Crawling:

- Hidden Web Crawling is difficult, since a search engine must “guess” what a user would insert in certain input boxes and precision and coverage of these systems are limited.
- Hidden Web is important since it is very large: i.e., estimates suggest that Hidden Web is 500 times bigger than traditional Web.

In this part we aim to improve hidden Web Crawling, by taking into account the AJAX capabilities of the crawled Web sites.

26.1 Motivating Example (Amazon)

Amazon [7] is an Electronic Store on which books, CDs and other merchandise can be bought online. Amazon offers a search interface through which a user can enter search keywords and find required products. Crawling the Amazon Web site in order to retrieve all possible products is a desired functionality, so that a commercial search engine can display a product page in its search results. As mentioned in Section 26, this is the purpose of hidden Web crawlers which need to submit queries into the search interface, in order to access content from the database underlying the Website. This approach is usually error prone since a search engine would use a dictionary in order to submit queries. [98]. The goal is to emulate user input accurately and broad as possible.
Chapter 26. Problem Statement

Figure 26.1: Amazon’s Suggest Functionality.

Amazon has recently integrated AJAX features in their Web site. One of them is the suggest functionality: when the user types a keyword, words are suggested from the database, as shown in Figure 26.1. We will use this feature into crawling, in order to improve the quality of a traditional Hidden Web Crawler on Amazon, crawler which just uses blindly a dictionary in order to access the underlying database.

26.2 Problem Description

With the advent of AJAX Web Sites, the problem of Hidden Web Crawling receives additional help, i.e., suggestions which already reflect content from the database. It is relevant to study the benefits of these suggestions in terms of: (i) coverage, (ii) quality, (iii) return on investment when compared to traditional AJAX Crawling and give values to the corresponding metrics.

26.3 Plan of Attack

In this part we take a further step in crawling the Hidden Web. We use an approach which crawls a Database-Driven AJAX Web Site that accepts user input, which is furthermore enhanced with AJAX capabilities, called Suggest. AJAX Suggest assists a usual user in typing relevant information to the website, even during typing, by displaying data taken directly from the underlying database of the system. An AJAX Crawler can use this information which reflects a part of the underlying Database of the Website, in order to:
(i) **Improve Crawling Coverage.** More information can be retrieved from the actual database underlying the Web site.

(ii) **Improve Crawling Quality.** Relevant information can be retrieved from the actual database, since the suggest functionality returns relevant information *for free.*

The contribution of this work is an AJAX Suggest Crawler and an evaluation of the use of AJAX in order to improve crawling. The goal was to show: how big the coverage of the AJAX Crawler is, and also how the Return on Investment (ROI) for AJAX Crawler is, considering that AJAX Suggest functionality does not need to be guessed by a crawler and that it comes for free from the AJAX Web site. We apply the crawler on the real Amazon [7] book search, which supports AJAX Suggest.

This part of the dissertation is structured as follows: Chapter 26 presented an overview of the potential of AJAX in crawling the Hidden Web. Chapter 27 presents the metrics used to evaluate the quality of a hidden Web Crawler: effort, coverage, and return on investment (ROI). Chapter 28 introduces AJAX Suggest crawling method, and shows it in comparison the traditional way to crawl the Hidden Web. Following up, Chapter 29 presents the experimental results and values obtained for the suggested metrics when crawling Amazon’s “books” section, using traditional Hidden Web Crawling, AJAXSuggest Crawling and the combined approach. Chapter 30 presents related work while Chapter 31 concludes the work and presents avenues for future work.
Chapter 27

Qualitative Evaluation of Hidden Web Crawling

In order to correctly estimate the difference in quality between Hidden and AJAX Crawling methods, we define clear methods.

27.1 Metrics for the Quality of Hidden Web Crawling

The goal of a crawler is to extract as much information as possible, with as little effort as possible. In order to do this, we devise the Effort, Quality and Return on Investment (ROI) metrics, as displayed in Table 27.1.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Traditional</th>
<th>AJAXSuggest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort</td>
<td>Number of dictionary words</td>
<td>Number of letter combinations</td>
</tr>
<tr>
<td>Quality</td>
<td>Number of distinct results</td>
<td>Number of distinct results</td>
</tr>
<tr>
<td>ROI</td>
<td>Quality/Effort</td>
<td>Quality/Effort</td>
</tr>
</tbody>
</table>

Table 27.1: Metrics evaluating the Quality of a Hidden Web Search Engine.

27.2 Probabilistic Estimation of the Size of the Crawled Database (Random Sampling)

Previous sections used an absolute approach which estimates the amount of pages and products retrieved by the different crawling methods. What we want to achieve is an estimation of the coverage with regards to the whole size of the Amazon Database. We use a probabilistic approach based random sampling.
Random Sampling Model

Problem Statement. We assume we have a corpus of \( N \) pages. By following a probabilistic law, we sample pages out of the corpus \( n \) times. The total number of distinct pages we obtained is \( m \) with \( k \) collisions. Requirement: Compute \( N \).

\[
N = m + k
\]

![Figure 27.1: Estimating the size \( N \) of a Corpus Using Random Sampling.](image)

Probalistically, we model this problem as in Figure 27.1 in the following way:

The probability to have hit one of the \( m \) distinct pages is \( \frac{m}{N} \). On the other side, this is equal to the probability of falling \( n \) times inside the \( m \) boxes, which is equal to \( 1 - \frac{1}{N} \) \( n \) (i.e., the probability not to fall outside the boxes, for \( n \) times). The equation that we use as the base is:

\[
N(1 - \left(1 - \frac{1}{N}\right)^n) = m
\]

Alternative 1: \( k \ll n \). In this case we estimate \( N \) as being a polynomial:

\[
N = a_2n^2 + a_1n + a_0
\]

By solving this equation with a large \( n \) and zeroing the corresponding terms for the large powers of \( n \), we obtain:

\[
N = \frac{n^2}{2 \cdot k} - \frac{n(3 + 2k)}{6k} - \frac{12 - k}{18}
\]

\[
N = \frac{n^2}{2 \cdot k} + O(n)
\]

The experimental section of this part shows that \( m \ll k \) and this case is not applicable.

Alternative 2: \( m \ll n \). In this case, we solve the probabilistic equation above and make the following notations:

\[
m = \alpha n
\]

\[
N = \beta n
\]

The equation above can be rewritten as:
27.2. Probabilistic Estimation of the Size of the Crawled Database (Random Sampling)

\[ 1 - e^{-\frac{1}{\beta}} = \frac{\alpha}{\beta} \]

The solution of this equation in \( \gamma \) (\( \alpha \) is given) is:

\[ \beta = \frac{1}{\text{LambertW}(-\frac{1}{\alpha}e^{-\frac{1}{\alpha}}) + \frac{1}{\alpha}} \]

where \( \text{LambertW} \) (the Lambert function) is the function \( w \) which is the result of the equation \( w(x)e^{w(x)} = x \).

We will use this result in the Experimental part of this work, Chapter 29.
Chapter 28

AJAX Suggest Crawling

We devise two crawling methods for the hidden Web: Traditional, AJAXSuggest. Furthermore, we devise a combined one which uses the benefits of all three methods. The three methods are represented in Figure 28.1 and described below.

28.1 Traditional Hidden Web Crawling

We use a basic hidden Web crawling method, which aims to access hidden content by "guessing" what a user would enter in a search box. The method works based on the following steps:

- prepare an exhaustive list of English dictionary words without a specific topic (using available dictionaries on the Internet)
- submit the words to Amazon
- retrieve the results

28.2 AJAXSuggest Crawling Model

We also adopt an enhanced crawling model for AJAX Web Sites. We tend to minimize the effort in terms of used words, by using the following steps:
• generate just letter combinations of one, two, three, etc. letters, etc.
• retrieve the corresponding Amazon suggestions (in case of Amazon these are usually partial phrases, titles, authors, etc.)
• submit the suggestions to Amazon, and
• retrieve the results.

28.3 Combined Crawling Model

The third way to crawl AJAX is the mixed model, i.e., use first the traditional crawling method, and then use the AJAXSuggest as an ‘addition’ to the initial model, for better coverage. It is important in all three cases to find a model which retrieves the benefits of the enhanced method. We will evaluate all these three methods in terms of the metrics defined in Chapter 27.1, on the example of Amazon’s Book search. We will show that each method brings its own benefits, and results not retrieved by the other methods, and will recommend the use of a combined method in crawling the AJAX Hidden Web.
Chapter 29

Experimental Section

Estimating coverage and quality of Hidden Web AJAX Crawler was done on Amazon. We have crawled the Hidden Amazon Web page by using the input text box for book search. The two approaches above have been studied, their combination, and the two metrics Effort, Quality and Return on Investment (ROI) as shown in Table 27.1. We estimate Effort by the number of queries that need to be submitted to the application (which increases Network traffic) and Quality by the actual coverage of the results obtained through the method, and ROI as Quality/Effort.

29.1 Experimental Setup

We crawled the Amazon website, focusing on the Book category, and extracted the results obtained after submitted dictionary queries (Traditional approach) and on the other side combinations of letters, which lead to Suggestions that are again submitted to the Amazon, since they are known to be leading to results from the database. Queries were submitted either through the Amazon Web Services API [100] or programmatically through an actual Web Browser component embedded in the crawler. Web Application Testing APIs Watij [109] was used, as well as the browser plug-in [59].

Traditional Approach. The Traditional approach is dictionary based and emulates a traditional Hidden Web crawler, without knowledge on the content or semantics of the underlying database. We used a complete lists of 109'581 English words which are a combination of several dictionaries. The list is available at [46].

The dictionary-based approach is characterized by the following two features:

- broadness: dictionary words have a general meaning (they cover a large range of topics).
- redundancy: many groups of dictionary words have similar meanings since they have for example the same stem.

This causes a possibly broad, but also not focused search. This only influences Precision of search though. In order to optimize the test execution time and maintain a compatibility to the AJAXSuggest approach, which uses letter combinations, we have submitted just words starting either with “a” or “b”.

125
AJAXSuggestion Approach. The new approach we propose for crawling the Hidden Web aims to minimize effort in submitting queries, and maximizing the amount of information retrieved from Amazon for free through the suggestion functionality. We therefore generate suggestions by virtually typing letters in the search box. In order to emulate a real user, we input combinations of one, two and three letters of the alphabet. Typing causes Amazon to retrieve relevant suggestions from the database, which complete the letter already in the input box. The final step is to submit now the actual suggestions to Amazon, known to contain relevant information, and to retrieve the results. As in the case of traditional crawling, we have just submitted letter combinations of one, two, and three letters starting with “a” or “b”.

The properties of the suggestion-based approach are:

- The effort is limited to the amount of possible letter combinations, less than the total number of dictionaries.
- Suggestions retrieved from Amazon are for sure relevant, since by definition are returned by Amazon from the database.
- Suggestions multiply “for free” the effort of typing text (ten suggestions are retrieved for entry).
- Suggestions contain not only words, but also expressions, name of books or authors.
- Results retrieved by submitting the suggestions to the search engines have a higher coverage, but are more specialized, focused, meaning.

Combined Approach. We have taken the results obtained by both results in order to show the clear overall benefit of using both the traditional and the AJAXSuggest approach for crawling the Hidden Web.

Technical Limitations

In performing tests, we have used the Amazon Web Services API [100], and another method which uses a browser programmatically, in order to interact with the user. The limitations are that the Amazon API returns maximum 4000 results and the browser 1200, which limits the accuracy of the estimated coverage. Another limitation is the limited number of suggestions returned by Amazon (i.e., maximum ten per letter combination), which also decreases the overall factor with which the small effort of the letter combinations is actually increased. Finally an enhanced Hidden Web crawling approach can obviously be used when the Web site offers the suggest functionality.

29.2 Effort

For each of the suggested crawling methods we compute the amount of queries needed for obtaining the final results from Amazon. The idea is that AJAX suggestions need a few queries which are then boosted by the suggestions, as opposed to the dictionary approach.

It can be noticed from Figure 29.1 that, indeed, AJAXSuggest that submitted just letter combinations and suggestions, is a very promising method. The dictionary itself submits all dictionary words, meaning a lot more words than in case of AJAX Suggest.
29.3 Quality of Crawling: Relative Coverage

We estimate for each of the crawling methods the amount of results returned. We mention again that the goal of hidden Web crawling is to increase coverage of the whole hidden database.

Figure 29.2: Coverage of the Database in case of Traditional, AJAXSuggest and Combined Hidden Web Crawling.

We show in Figure 29.2 the results obtained by each of the methods Traditional, AJAX Suggest and Combined. The figure shows a relative strong coverage of both first methods and a clear improvement in case of the combination of the methods. The number of results returned by Traditional is anyway very high since the number of submitted words is high.

It is worth mentioning that there is both big overlapping in the set of returned results by the two crawling methods and also a large number of disjunct results, on both sides. The amounts are displayed in Figure 29.3. The overlapping is of a factor of 24% of the total retrieved results and the benefit obtained by the AJAXSuggest method is big enough not to be ignored: 45% of the results returned by AJAXSuggest are not found by the dictionary. The conclusion of this experiment is the fact that the combined method Traditional and AJAXSuggest gives the best coverage, and furthermore, a significantly better coverage than the traditional approach, with less additional effort (as shown in Figure 29.1).
Chapter 29. Experimental Section

Figure 29.3: Distribution of results returned by Traditional, AJAXSuggest and Combined Hidden Web Crawling.

29.4 Probabilistic Estimation of the Absolute Coverage.

As explained in Section 27.2, we can estimate the coverage of the entire Amazon using the probabilistic sampling method. For this we have used the Amazon Book Search engine using random queries from the following sources:

- The initial generic dictionary.
- Britcaps [24] list of proper names, places, abbreviations, company names.
- Author names [112].
- Idioms. List of English expressions.

From each category we randomly picked half of the words, which we submitted to Amazon and computed the union of the total pages. The statistics are summarized in Table 29.1:

Based on the formulas in Section 27.2, for the case $m \ll n$, we obtain the following results:

$$\alpha = \frac{m}{n} = \frac{10}{15} = \frac{2}{3}$$

$$\gamma = \frac{N}{n} = 1.143880143$$

Based on these results we conclude that:

<table>
<thead>
<tr>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of submitted queries</td>
<td>$n = 13,000$</td>
</tr>
<tr>
<td>Number of results from random sampling</td>
<td>$n = 15\text{Million}$</td>
</tr>
<tr>
<td>Total number of distinct pages</td>
<td>$m = 5\text{Million}$</td>
</tr>
<tr>
<td>Total number of collisions</td>
<td>$k = 10\text{Million}$</td>
</tr>
</tbody>
</table>

Table 29.1: Results from random sampling of Amazon’s book Website.
\[ N = 17158202 = 17.16 \cdot 10^6 \]

After having estimated the size of \( N \) (i.e., the total number of Amazon book pages), we can estimate the absolute coverage of each method that we have in Table 29.2. This shows that adding AJAXSuggest to the traditional hidden Web Crawling causes a 7% increase in coverage and is worth from the quality point of view.

### 29.5 ROI: Return on Investment.

We estimate the Return on Investment of each method, by evaluating the cost versus gain for each method. The gain is the Coverage, the cost (i.e., investment) is the number of submitted queries to Amazon.

![Return on Investment](image)

**Figure 29.4:** Return on Investment for Traditional, AJAXSuggest and Combined Hidden Web Crawling.

We display the Return on Investment of each crawling method in Figure 29.4. It is clear that the dictionary based approach has a high return on investment, because of the high coverage of the results. The effort is however a lot bigger than for the AJAXSuggestion approach, which only submits a few letter combinations and the suggestions obtained from Amazon, with a lower result coverage though. The AJAXSuggest method returns a higher ROI but the combined method returns of course the best ROI, with less additional effort as opposed to traditional crawling.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Traditional</th>
<th>AJAXSuggest</th>
<th>Combined Crawling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Coverage</td>
<td>24.72%</td>
<td>13.58%</td>
<td>30.86%</td>
</tr>
</tbody>
</table>

*Table 29.2: Metrics evaluating the Quality of a Hidden Web Search Engine.*
Chapter 30

Related Work

Crawling is a studied problem and just as mentioned in Chapter II, work on crawling includes references to commercial search engines such Google [67] or Yahoo [115]. Hidden Web [111] has been studied in works such as [76], [43] and [98]. The challenge of finding query strategies in order to optimize Deep Web Crawling has been studied in [94] and the issue of coverage was explicitly addressed in [43]. The Performance and quality of search engine is a matter of classical Information Retrieval [16]. Most interesting is the notion of search engine coverage, and especially its estimation based on actual data. Other works also tried to compute the coverage of Amazon, which is not publicly available. No work has considered AJAX as an actual source for increasing the data quality.
Chapter 31

Summary

The suggestions approach brings a novelty in the exiting methods of crawling the hidden Web. Crawling is not performed blindly anymore but guided by these suggestions. Since the suggestions have a more specialized meaning than the words from a dictionary, the AJAXSuggest approach gets a smaller coverage. But words for dictionary are redundant as meaning and bring a lot of redundant results for high effort. If the suggestions are extracted without any strategy or restrictions, the redundancy is not as high as in the case of the dictionary, and with obvious less effort. For best coverage, a combined result is however recommended.

Future work involves using suggestions not only for obtaining coverage, but also for achieving precision. This results intuitively from the fact that when a user looks for a specific topic, the suggestions from the Web site already guide the user to a relevant result from the database (based on text similarity). In order to better measure the quality of results, the set of words from dictionary can be extended. Furthermore, the AJAXSuggest method, other letter combinations could be tried and also the length of letter combinations should be increased. Also a good idea would be to find strategies for selecting the suggestions that bring large number of results or more relevant results. A user study can be performed in order to accurately estimate the quality of results and more elaborate probabilistic estimations of corpus sizes can be used, based on random walker models [25].
Part IV

Ranking in AJAX Applications
Chapter 32

Problem Statement

As shown in Chapter II, in AJAX and in Rich Internet Applications (RIAs) in general, the traditional paradigm where a page is identified by a unique URL breaks. For example, a map application can show multiple city locations under the same URL since the logic resides on the client. For the same reason, an AJAX News Application displays each AJAX News Item apparently separated, but still under the same URL. The difference is shown in Figure 32.1. This is opposed to traditional Web Sites, where a Web Site consists of Web pages linked through internal and external hyperlinks. This new model induces also new entities in the way the results of search are presented and computed. As opposed to traditional search, where search results are URLs which uniquely identify pages, in this case results are application states all residing under the same URI, and the states themselves are linked through transitions, caused by user events. These new dimensions have not been explored before. This part explores and divides new techniques for ranking based on the application logic in the new AJAX applications, such as the implicit semantics given by order and DOM structure.

32.1 Motivating Example

As also shown in Part II, we take the simple example of an AJAX news application, shown in Figure 32.2. The application displays news items ordered based on time, dynamically loaded from the server as the user invokes next and previous Javascript events on the interface. This simple version of an AJAX Application, partially inspired by the dynamic loading of E-mail message in an AJAX E-mail client such

![Diagram of Traditional and AJAX Web Sites](image)

Figure 32.1: The transition to AJAX changes the focus from HTML pages and links to AJAX States, Events and Application Logic.

137
Chapter 32. Problem Statement

as Yahoo!Mail and by the click-based interactivity of an application such as Google Maps. As opposed to a traditional Web application, all news items are displayed under the same URL. Therefore, traditional search would not differentiate between different states in the application.

The importance of state order. Keyword search in application data produces results expressed in terms of \{URL, state\}. For example, searching for millionaire produces the results:

(1)\{www.fakenews.com, s1\} (2)\{www.fakenews.com, s2\}.

It is important to know that usually the order of states in the application logic is not random and it already reflects an importance of the state. We aim to include this into search and make sure that the rank of result including s1 is higher than the rank of results including s2. In case of news items, calendar items, etc. the most recent items are reflected in the order in which the states are displayed in the application. Also, the first state in an AJAX application has a higher rank than all others, since it is the only one can access directly using the URL itself.

The importance of page structure. Complex applications such as GMail [65] display at the same time E-Mails, news, calendar information, etc. The news application above is just a sub-case of the whole application. The following question arise: (i) which parts of the application page are most important in terms of data that flows through them, and (ii) which triggers (event generators) contribute to generating most of the data? The situation is illustrated in Figure 32.3.

The importance of content. Content is of course also important in AJAX search. The other two dimensions of search: states and structure are however new and specific to AJAX applications.

32.2 Problem Description

Dimensions of Ranking. The application logic of an application is of course a ubiquitous type of semantic information. However, these semantics remain generally unexplored in search. A crawler simulating all possible events can build the model of the application logic, consisting of states and transitions. This automaton materializes the implicit semantic information on states, transitions and events described by the application logic. Search on Rich Internet Applications should return states, or, even more specifically, states and structural information from the application (e.g., main text area or title) containing the desired keywords.

Ranking AJAX application addresses the following issues, that we address in this work: (i) how to rank application states according to their importance in the application logic and (ii) how to rank application (page) structure according to its participation in the application logic and (iii) how to rank the AJAX application as a whole. The three dimensions based on which we base our approach are: application logic (transitions, events), content (the actual text) and page structure (DOM), as shown in Figure 32.4.

Figure 32.2: AJAX News Application and two news items.
32.3 Plan of Attack.

As mentioned in the Introduction, the semantics given by application logic is an unexplored source of opportunities. This paper brings the following contributions by proposing ranking functions which take into account all of the following:

1. **States.** States are not static Web pages, but are generating during user interaction. States are generated using events and both their connectivity in the transition graph and their content influences their importance.

2. **Exploring Click distance.** When searching news items, it is interesting to attach a higher rank to more recent news items. From the application logic point of view, most recent news items are closer to the first news item. This is a dimension not explored by current search engines, not even for traditional Web applications. For this purpose, we adopt a ranking scheme which enhances PageRank.

3. **Exploring Relevant Page Parts.** Another ranking dimension is that of the page structure (the application structure). For example, Yahoo Mail displays e-mail messages and a calendar in the same page, but in two different panels. These two separate parts of the application are independent and result in a number of states which is polynomial in the number of E-mail messages and the number of calendar events. Searching for content appearing in both parts of the application can be done, but content should be ranked differently: i.e., most likely, the content part which can get most clicks from the user (i.e., the E-mail messages) should have an added weight.

4. **Detecting Irrelevant Page Parts.** If some parts of the application do not participate in the application logic, this should be explored in terms of factoring out these parts and indexing them only once, instead of in each state. This saves index size. In an extreme case, they should not be indexed at all if there is an indication they do no serve search (e.g., copyright information).
5. Smart Crawling. Indexing AJAX applications is an expensive process compared to traditional search: a lot of events must be generated, there where traditional search only indexes one page.

This part of the dissertation is organized as follows: Chapter 33 shows how crawling and indexing are performed in order to extract application logic, and how search is done on the application logic. Chapter 34 presents AJAX Rank, the algorithm which adapts PageRank in order to rank application states, application events and application structure based on the application logic. In Chapter 35 we propose AJAX HITS, an algorithm which assigns importance to page parts, with the purpose of achieving a smart crawling of AJAX Web Sites. Subsequently, Chapter 36 shows experimentally the capabilities of the ranking function we propose for application logic, Chapter 37 discusses the possibilities to integrate the new methods into existing search engines. Finally, Chapter 38 presents related work while Chapter 39 summarizes and presents conclusions and future work.
Chapter 33

Capturing Application Logic.

We re-iterate the steps presented in Chapter II applied to the news application example.

**Crawling.** For the purpose of indexing states, we build the application model, e.g. a graph (an automaton) with nodes being states and edges being the transitions. A part of the application model of the AJAX News Application is presented in Figure 17.1.

![Figure 33.1: Entities describing an event in an AJAX Application.](image)

From a practical perspective, this graph is annotated with information on which elements triggered the event (i.e., source), the actual type of event (e.g., click, move), the elements affected by the events (i.e., target) and what changed (i.e., the action). An example is given in Figure 33.1.

**Indexing.** After crawling, an inverted file is built containing for each keyword the application it appears in, the state each keyword belongs to. In order to exploit at maximum the application logic, the DOM element where the keyword appears is also mentioned. A score is additionally computed as the number of occurrences of the word in the given state. Table 33.1 presents the inverted file for the AJAX News Application of Figure 32.2.

<table>
<thead>
<tr>
<th>Word</th>
<th>URI</th>
<th>State</th>
<th>Element</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>millionaire</td>
<td>s1</td>
<td><a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsTitle</td>
<td>1</td>
</tr>
<tr>
<td>millionaire</td>
<td>s1</td>
<td><a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsContent</td>
<td>1</td>
</tr>
<tr>
<td>millionaire</td>
<td>s2</td>
<td><a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsContent</td>
<td>0.8</td>
</tr>
<tr>
<td>millionaire</td>
<td>s2</td>
<td><a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsTitle</td>
<td>0.8</td>
</tr>
<tr>
<td>country</td>
<td>s1</td>
<td><a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsContent</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 33.1: Inverted File for AJAX news application (Figure 32.3).

**Search.** The main purpose of capturing application logic is of course to search relevant states in an AJAX application (which traditional search engines do not do). We can use the index in order to retrieve
Chapter 33. Capturing Application Logic.

the corresponding states. Table 33.2 shows the results of two queries: *country*, returning one state and *millionaire*, returning two states and corresponding elements. Each query returns the URI and the state containing the keywords. Additionally, a rank is computed as it will be shown in Section 34. Our goal is to give importance to the first state since it is more recent.

<table>
<thead>
<tr>
<th>Query</th>
<th>URI</th>
<th>State</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td>s1 <a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsContent</td>
<td>0.8</td>
</tr>
<tr>
<td>millionaire</td>
<td>s2 <a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsContent</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>s2 <a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsTitle</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>s1 <a href="http://www.fakenews.com">www.fakenews.com</a></td>
<td>newsContent</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 33.2: Query Processing on the AJAX news application.

In case of conjunctions, the result of search is a tuple containing the state and the Least Common Ancestor (LCA) element of the DOM elements containing the given keywords as shown in Figure 33.2. The rank in this case is computed as an aggregated value of the individual rankings.

Figure 33.2: Computing Least Common Ancestor during Query Processing.
Chapter 34

AJAX Rank: Ranking States, Events and Application Structure

Ranking an AJAX Application raises both challenges and opportunities. Instead of relying on a static, flat page model as in traditional IR, a more dynamic ranking method, based more on XML-IR approaches, can be defined. The challenges arise from the need to assign a weight not only to the page, but also to the application state and to possibly avoid recrawling the same states in the page. There are three dimensions of ranking, as explained in Figure 32.4

34.1 Ranking Requirements

We can summarize the special observations which influence ranking in an AJAX application, as follows:

1. Ranking affects application states and DOM elements.
2. The importance of a state transition (of an event) influences the rank of a state. (e.g., a move action is ranked lower than \( innerHTML = ... \)) influencing the destination state.
3. The participation of an element to changes in the whole application increases its rank.

As a consequence, we defined \textbf{AJAXRank}, a ranking function which fulfills the above requirements. Its functionality and quality depends on the application. AJAXRank has the following building blocks:

1. Ranking application states.
2. Ranking application events.
3. Ranking the application Structure.

We detail the building blocks of AJAXRank as follows.

34.2 Ranking Application States using PageRank

Ranking Application States is the most important building block of AJAXRank. It encompasses the fact that an AJAX application is composed of states connected through transitions which correspond to
events. We define this by adapting PageRank [22] to the specific requirements of an AJAX Application. Furthermore, we also show that PageRank itself does not apply well to an AJAX Application.

PageRank is a well-known, objective, ranking function for Web pages. It is described by the following formula, applied in a converging iterative algorithm:

$$ PR(p_i) = \frac{1-d}{N} + d \cdot \sum_{p_j \in H^{-1}(p_i)} \frac{PR(p_j)}{|H(p_j)|} $$

In the formula, $p_1, p_2, \ldots, p_N$ are the pages under consideration, $H^{-1}(p_i)$ is the set of pages that link to $p_i$, $|H(p_j)|$ is the number of outbound links on page $p_j$ and $N$ is the total number of pages.

The intuition behind PageRank [22] is:

1. A user can jump to any Web page randomly (with probability $(1-d)/N$).
2. A user can choose to jump to any page linked from the current page (with probability $d$).
3. The importance of a page is boosted by the importance of the linked pages.
4. The importance of a page boosts the rank of the linked page.

The naïve way to define a ranking function for AJAX is to consider all events as links, consider states as being pages, and apply PageRank. We argue as follows that this approach fails, and the PageRank algorithm is not suitable for AJAX. In particular the following hold in AJAX but not in PageRank:

- **A user cannot jump to any state randomly.** This is only valid to the first state. The first state can be reached either directly through the URL which points to that state, or from any other state of the AJAX application.
- **“Three-click-rule”.** Given a chain of states linked by events, the user is most likely to browse just through some of them (in an extreme case just three) as it gets further from the initial state. The name “three-click-rule” is frequent in Web usability studies; we adopt it here and adapt it to the requirements of an AJAX Web Application and of search.

We adapt the PageRank algorithm so that it applies these characteristics. We define **AJAXRank (AR)** as follows:

$$ AR(s_1) = (1-d) + d \cdot \sum_{s_j \in H^{-1}(s_1)} (w(s_j, s_1) \cdot AR(s_j)) $$

$$ AR(s_i)_{i \geq 2} = d \cdot \sum_{s_j \in H^{-1}(s_i)} (w(s_j, s_i) \cdot AR(s_j)) $$

where $s_1, s_2, \ldots, s_N$ are the states under consideration, $s_1$ being the first state, $H^{-1}(s_1)$ is the set of states that have transitions to $s_1$, $w(s_i, s_j)$ is the weight assigned to the transition from state $s_i$ to state $s_j$ where $w(s_i, s_j)$ is zero if $s_i$ does not have a transition to $s_j$. Note that $\sum_{s_k \in H(s_j)} w(s_j, s_k) = 1$. As opposed to PageRank, the first state is the only state to which a random jump is possible, that is why the $(1-d)$ factor appears just in that case.

Similarly to PageRank, AJAXRank has a result of vector of values computed using an iterative algorithm. The iteration defines a convergent series, the convergence limit being the vector with the final values of the AJAXRank. Section 36 will show a comparison between AJAXRank and PageRank and how the parameter $d$ value can be chosen. It is worth mentioning that we settled to the value $d = 0.75$.

### 34.3 Ranking Application Events

There are different types of events that can appear in the document: click, double click, move, scroll, etc. As shown in Section 33, the events are characterized by a type of an event triggered on a source
34.3. Ranking Application Events

element, and on one or more targets to which actions are applied. Each state is derived from another state based on a single event. We use this information for assigning a different weight to edges than the weight based only on the number of outgoing edges. In particular, we use the type of event and the actions applied on the target elements in order to determine the importance of an event (an edge in the graph).

Each edge between two states is annotated during crawling with the event information as shown in Figure 17.1. Each transition \( t \) between two states \( s_i \) and \( s_j \) is annotated with a source \( src \), one event type \( e \), more targets \( t_i \) and corresponding actions \( a_i \), as shown in Figure 34.1. Each target \( t_i \) has a single corresponding action \( \text{action}_i \) (such as \text{innerHTML}, \text{move}, etc.).

![Figure 34.1: Application Model annotated with Source, Events, Targets and Actions.](image)

The actions and events have a relative importance which influences the weight of the transition edge. In particular, we defined the importance of actions and events as scores, as shown in Table 34.2a for events and Table 34.2b for scores. The scores are empirically defined based on the observed frequency of these events and on the importance of the modification caused by the actions.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>click, doubleclick</td>
<td>1</td>
</tr>
<tr>
<td>mouseover</td>
<td>0.5</td>
</tr>
<tr>
<td>rest</td>
<td>0.4</td>
</tr>
</tbody>
</table>

(a)Scores for Events

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>innerHTML, outerHTML</td>
<td>1</td>
</tr>
<tr>
<td>visibility change</td>
<td>0.75</td>
</tr>
<tr>
<td>document.write</td>
<td>0.5</td>
</tr>
<tr>
<td>element movement</td>
<td>0.25</td>
</tr>
<tr>
<td>color change</td>
<td>0.25</td>
</tr>
</tbody>
</table>

(b)Scores for Actions

The intuition is that click events (\text{onClick}, \text{onDblClick}, etc.) rank higher than \text{mouseover} events. The rest of the events are classified equally, as shown in Table 34.2a. The actions applied to the targets also have a differentiated importance. Table 34.2b shows that the greater importance is given to actual content change (innerHTML, outerHTML), visibility change (using properties of the stylesheet), append at the end of the document (document.write) and then other move and change events. Consequently, after defining the importance of actions and events, we distribute these weights on the edges. The weight of each edge is proportional to the importance of the corresponding actions and events.

The weight of an edge \( s_i, s_j \), generated by the event \( e \) on source element \( s_i \), applied to the targets \( t_i \) through the actions \( a_i \) is:

\[
w(s_i, s_j) = \frac{1}{\sum_{s_k \in \text{H}(s_i)} w(s_i, s_k)} \cdot \text{score}(e) \cdot \sum_{a_i \in A(s_i, s_j)} \text{score}(a_i),
\]

where \( s_k \) are all states reachable from \( s_i \) and \( A(s_i, s_j) \) are all actions that the event applied on the targets.
This formula ensures that the sum of all weights of the outgoing edges is 1. The new weights influence AJAXRank and will be used accordingly in Section 36.

### 34.4 Ranking Application Structure

The goal of this work is to correctly identify, index, and rank the relevant parts of AJAX applications. Search is one goal for doing this, but this information can also be used for actually development purposes or even for building a focused crawler based on the most important parts of an application. We define the term structure as the page DOM of the the application. There is usually a fixed skeleton structure on which all events occur.

As an example, we detail the application structure (DOM) of the AJAX News application of the Example in Section 32 in Figure 34.3 intuitively and 34.4 (as a DOM).

![Figure 34.3: Structure of the AJAX news application (Figure 32.2).](image)

In order to compute the importance of the application parts of an AJAX application, we used the following intuition: if an element is a frequent target of changes, it is more important. More information on the actual AJAX application can be therefore derived.

We can decide the importance of an element based on another view of the application model: the Annotated Application Structure. We model it as a DOM annotated with information on the elements whose content changes, or which act as source or target of an event in the Transition Graph. The importance is computed based on the element’s participation to roles (source or target), events and actions.

An example of the annotated Application Structure is shown in Figure 34.5. Each node is annotated with the event, role and operation (event type, e.g, click or action, e.g., innerHTML) to which it participated, based on Table 34.1.

This information at hand, the ranking of an element can be relatively easy computed as the average of its contributions throughout the application. We base the result on the already defined scores of the events (Table 34.2)a and actions (Table 34.2)b. Furthermore, we define a score on whether an element
34.4. Ranking Application Structure

Figure 34.5: Annotated Application Structure with (Event, Role, Operation) for the AJAX news application (Figure 32.2)

<table>
<thead>
<tr>
<th>Element</th>
<th>Event</th>
<th>Role</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>newsTitle</td>
<td>e1</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsTitle</td>
<td>e2</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsTitle</td>
<td>e3</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsTitle</td>
<td>e4</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsContent</td>
<td>e1</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsContent</td>
<td>e2</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsContent</td>
<td>e3</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>newsContent</td>
<td>e4</td>
<td>target</td>
<td>innerHTML</td>
</tr>
<tr>
<td>next</td>
<td>e1</td>
<td>source</td>
<td>click</td>
</tr>
<tr>
<td>next</td>
<td>e2</td>
<td>source</td>
<td>click</td>
</tr>
<tr>
<td>prev</td>
<td>e3</td>
<td>source</td>
<td>click</td>
</tr>
<tr>
<td>prev</td>
<td>e4</td>
<td>source</td>
<td>click</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 34.1: Annotation for the Application Structure of the AJAX news application (Figure 32.2).

acts as a source (less important) or a target (very important since the content changes), as shown in Table 34.2.

<table>
<thead>
<tr>
<th>Role</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>0.2</td>
</tr>
<tr>
<td>target</td>
<td>0.8</td>
</tr>
<tr>
<td>rest</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 34.2: Scores for Roles

In order to compute the $ER$ (ElementRank), we consider an element $E$ which participated to $N$ events. We encode each event with $event_i$, the role with $r_i$ (either “source” or “target”) and the operation with $o_i$. If $E$ was a source (less important), then $o_i$ encodes the event type $e_i$ which triggered the transition (e.g., click); if $E$ was a target (more important), then the $o_i$ encodes the action $a_i$ applied on it to change the
content. The ER is computed using the following formula:

\[
ER(E) = \frac{\sum_{i=1,\ldots,n} \text{score}(r_i) \cdot \text{score}(o_i)}{N}
\]

The astute reader may notice that it is not always possible to generate a stable DOM, valid throughout the application, simply because the page might change completely during the transitions. However, this is possible if the structure is stable, which is the case of the simple news application but also Google Maps, GMail or Yahoo! Mail. We leave the most complex case as future work.

### 34.5 Cumulative AJAXRank.

A result of a Search for a query \( Q \) is a triple \( \langle URL, state, element \rangle \). We define the rank of the result as a weighted sum of keyword importance (\( tf*idf \) ranking), of \( AR(state) \) (the AJAXRank of a given state) and the \( ER(element) \) (the rank of DOM element), as given in the following formula:

\[
R(res, Q) = w_1 \cdot \sum_{k \in Q} tf(k, state) \cdot idf(state) + w_2 \cdot AR(state) + w_3 \cdot ER(element).
\]

The AR of a state has already been defined. The \( tf \) and \( idf \) are defined as follows, and can be computed at indexing time, based on the following formulas:

\[
tf(k, s) = \frac{\text{count}(k, s)}{\sum_{t_k \in s} \text{count}(t_k, s)}
\]

\[
idf(k) = \log \frac{|s_j|}{\{s_j | k \in s_j\}}
\]

The \( tf \) of term \( k \) in state \( s \) is the term frequency of term \( k \) in a state \( s \), normalized with report to the count of all terms in state \( s \). The \( idf \) of a term \( t \) is the log of the number of total states divided by the total number of states the keywords appears in. Therefore, the semantics of the formulas follows the well-known semantics in IR [16], with states acting as documents. Of course, the \( tf \) and \( idf \) values are computed at indexing time and just read at query time.

### 34.6 Rank Aggregation.

During query processing several index entries must be merged in order to retrieve a single result containing more keywords. In case of conjunction queries, the LCA of two elements is computed during Processing of Conjunctions (Section 33). For example, as shown in Figure 34.6, the LCA of “newsTitle” and “newsContent” is “news”.

![Figure 34.6: Propagating Ranking to ancestor elements.](image)

**Ranking LCA.** The rank of the LCA is computed at the same time with query processing, as the average of the ranks of the subelements which are considered in the LCA, as shown in the next formula:

\[
\text{rank}(LCA(e_1, \ldots, e_n)) = \sum_{i=1,\ldots,n} \text{rank}(e_i).
\]
One can draw similarities between this and XRank [73]. Generally in an XML structure, the rank of the top elements influences the rank of the low element and vice-versa. In case of applications, we only propagate the rank to the top of the DOM, from very specific elements (e.g., dataContent), to less specific elements (e.g., body), higher in the hierarchy (but maybe more important visually). We will present the capabilities and potential of AJAX Rank in experimentally.
Chapter 35

AjaxHITS: Smart Crawling

The goal of indexing AJAX Sites is to increase precision by invoking Javascript events and crawling additional content generated on the client. The overload however may not be bearable at the scale of the Web especially if the events are too numerous. One idea to reduce the number of crawled events is to just crawl the events which generate most content and the parts whose content is mostly changed by the relevant events.

![Diagram of an application model for AjaxHITS](image)

Figure 35.1: Application Model for AjaxHITS

The trigger elements and the target elements of an event can be represented as an additional edge on the application DOM tree, as shown in Figure 35.1. Each edge is annotated with the number of events which have the same source and target, and we write $i \overset{N_{events}}{\rightarrow} j$. This information is retrieved from Table 34.1.

The problem of finding important elements translates into the following: find in the graph of the application structure, the most important nodes that point to other nodes, and the most important nodes that are pointed at. This is the same problem as HITS Algorithm [83], used in IR in order to retrieve the most relevant sites that contributed to a query on the Web, and the most important results.

35.1 The AjaxHITS Algorithm

We propose AjaxHITS, an adapted HITS algorithm for AJAX Web sites which works on a graph structure and retrieves most important trigger and target elements in the DOM of the same AJAX Applications.
Algorithm 35.1.1 describes AjaxHITS. In traditional HITS algorithms, each edge represents a different link from one page to another. In case of AjaxHITS, links are event dependencies. In our case, the idea is: important trigger nodes affect important target nodes of the page and important target nodes are affected by trigger nodes. As opposed to classical HITS, we add a single edge between the same source and target, but assign as a weight $w$ to the edge, equal the number of time the event that connects the source and target was invoked, written $i \xrightarrow{w} j$. This reduces the complexity of the original HITS which has more edges to process. Lines 9 and 10 from Algorithm 35.1.1 contain the part which differs from the original HITS algorithm and consider the weight of the edge in the formula for the algorithm and contribute to the complexity reduction.

**Algorithm 35.1.1 AJAX HITS Algorithm**

1: Input: N: number of nodes; Events $i \xrightarrow{w} j$ {i: trigger, j: target, $i \rightarrow j$ triggered $w$ times during crawling}
2: Output: $hubs_i$, $auth_j$, $i = 0 \ldots N - 1$
3: Function computeAjaxHITS()
4: for $i = 0$ to $N - 1$
5: $auth_i := 1$
6: $hubs_i := 1$
7: end for
8: while (!converge($hubs$)) AND (!converge($auth$)) do
9: $hubs_i := \sum_j auth_j \cdot w$
10: $auth_j := \sum_i hubs_i \cdot w$
11: end while
12: end Function

The result of the AjaxHITS algorithm is a rank associated to all triggers and targets in the page. Based on this result, a second crawl of a page can decide to trigger events just on the most important 'hubs' and to index just the most important 'authorities'. A microbenchmark is shown in Section 36 and shows that AjaxHITS offers a better strategy on choosing triggers than the strategy which does not take the amount of data in consideration. For indexing though, all target elements containing text should always be considered, independently of ranking - they just contribute for proper ranking of target elements.
Chapter 36

Experiments

The goal of the experiments is to emphasize the capabilities, performance and the added value of using AJAXRank for dynamic Web applications. We implemented the crawler which builds the application model based on the states, events and transitions. We evaluated the quality of the engine by running queries, conducting a restricted user study and by objective measures such as state coverage of the crawler.

36.1 Experimental setup

For this purpose, we consider the following AJAX applications:

1. **Simple news application with links.** A five-news-item application, corresponding to the application model in Figure 36.1. It is still an almost linear news application which has the states A, B, C, X, D, where A is the first state (i.e., the first piece of news) and D, the last one.

   ![Figure 36.1: Application Model of News Application with five states.](image)

   In order to have a more realistic scenario, we assume the application also has a link from the main page to a special piece of news (the favorite), which we choose to be X. By adding this link, the application model is not symmetrical anymore and is more representative for the ranking function. Figure 36.1 already assigns weights to the edges, equally distributed between the outgoing edges.

2. **Complex news application.** The test AJAX News Application was at the base of the running example in this work, and it is found at this URL[5]. The text consists of real news, in German. The simple body consists of a side menu with title news, a *news title* section and a *news content* section. News are changed by using two buttons (*next* and *previous* arrows) or the side menu. Therefore, the model is more complex: each piece of news is linked to all other pieces of news through AJAX events because of the side menu, and all news have next previous links to the next piece of news (as Shown in Figure

...
Each click causes an AJAX Call to be sent to the server to fetch the news, and the interface is changed through `innerHTML` events. The statistical information on number of states, events, actions, and number of keywords of this application is listed in Table 36.1.

<table>
<thead>
<tr>
<th>States</th>
<th>#Transitions (AJAXCalls)</th>
<th>#Source Elements</th>
<th>#Target Elements</th>
<th>#Actions (innerHTML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>80</td>
<td>10</td>
<td>4</td>
<td>452</td>
</tr>
<tr>
<td>16</td>
<td>288</td>
<td>18</td>
<td>4</td>
<td>1684</td>
</tr>
<tr>
<td>24</td>
<td>624</td>
<td>26</td>
<td>4</td>
<td>3684</td>
</tr>
<tr>
<td>32</td>
<td>1088</td>
<td>32</td>
<td>4</td>
<td>6452</td>
</tr>
<tr>
<td>40</td>
<td>1680</td>
<td>42</td>
<td>4</td>
<td>9988</td>
</tr>
<tr>
<td>48</td>
<td>2400</td>
<td>50</td>
<td>4</td>
<td>14292</td>
</tr>
</tbody>
</table>

Table 36.1: Number of AJAX Entities in the sample News Application.

The number of targets is fixed (four elements ever change) through `innerHTML` invocations (actions), and each event. The number of transactions is quadratic (because each piece of news is connected to each other piece of news not only to the neighbors.

3. **News Application with Calendar.** This example is inspired by Yahoo! Mail [116], which includes two independent parts in the interface: an E-mail reader for reading and browsing events, and a calendar displaying events. This application uses the complex mail reader without menu and the calendar for emulating the Yahoo! Mail application. The purpose of this example is to show how application structure can be extracted from the application. The interface of the application was presented in Chapter 23 and shown again in Figure 36.3.

We present the measurements in quality and performance on the most representative application from the ones below using **microbenchmarks**.
36.2 AJAXRank vs. PageRank

We applied PageRank and AJAXRank to the example in Figure 36.1 and it converges after 24 iterations. This improves with the number of states as shown in Section 36. We concluded that the most accurate value for the d parameter is lower than in case of PageRank, and we chose $d=0.75$. Intuitively, this is lower than the usual value for $d$ in PageRank (e.g., 0.85) because in case of AJAX Application the probability to jump to the first state $(1-d)$ is a lot higher than in case of PageRank, applied to the whole Web. The results of the ranking for PageRank and AJAXRank, and the order of the states, are presented in Figure 36.4.

![Figure 36.4: AJAXRank vs. PageRank in the application model of Figure 36.1.](image)

Obviously, PageRank ranks better the states which are highly linked to (i.e., X as the mostly linked, then C, B, D, as moderately linked, and only then the start state A). Therefore, PageRank ignores the importance of the starting state - a fair result for the Web. On the contrary, in an AJAX application, the starting state can be reached from all other states, or just by typing the URL of the application. Also the importance of the other states decreases as they are further away from the states which are directly reachable (A, X) (“Three-click-rule”). This is not reflected by PageRank since the first state is poorly ranked and this is an argument which makes PageRank not suitable for AJAX application. On the other side, AJAXRank favors the first state by insuring that there is a possibility of a random walk to it from all other states with probability $(1-d)$ where $d=0.75$. Naturally, state X is also highly ranked since it is easily reachable. The other states, B, C, D are ranked increasingly lower, conformant to the “three-click-rule”. We further detail this point as follows.

36.3 Influence of the Click-Distance

The advantage of AJAXRank vs. PageRank in order to favor states which are closer to the states that are directly reachable is better illustrated if the model is simplified, such as the one presented in Figure 36.5, corresponding to a double-linked list of states (e.g., news items accessible from the main page by clicking on the next or previous arrow, and state X is not favored through an additional link).

![Figure 36.5: Influence of click-distance on AJAXRank and PageRank.](image)
As also shown in Figure 36.5 (right side), PageRank favors the middle states, states A and D being ignored as they are in the extremities. AJAXRank acts according to both semantics of the first state and to that of the distance semantics. When additional links are present, AJAXRank also boosts middle states as has been shown before. The states which are further away from A rank low. They will rank however higher in the final result based on the importance of their content.

### 36.4 AJAXRank Iterations.

The computation of AJAXRank (in particular the iterative part of ranking states) is an integral part of crawling and building the index. Table 36.2 shows the number of iterations needed for the algorithm to converge in case of the Complex news application (2) for varying number of news items.

<table>
<thead>
<tr>
<th>States</th>
<th># AJAX Rank Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>14</td>
</tr>
<tr>
<td>40</td>
<td>14</td>
</tr>
<tr>
<td>48</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 36.2: Number of Iterations for AJAXRank for the AJAX news Application.

In case of the AJAX Crawling and in particular of the AJAX News example, the crawling time of states is dependent on the number of transitions (which is, at its turn, quadratic in the number of states). However, very importantly, AJAXRank needs a constant number of iterations until it converges, which is around 14 iterations, as shown in Table 36.2.

### 36.5 Choosing the AJAXRank Parameter.

We evaluate the choice of the parameter $d$ for AJAXRank. We had already shown that on the small example of the “linear” news application the ranking of states was made to reflect the two rules, i.e., first rule is important and the three-click rule. Figure 36.6 shows the actual ranking order of the application states using the (1) Simple news application (without additional link) with different parameters, from 0.65 to 0.95. The value of $d = 0.75$ is the most appropriate. Smaller values, cause the first states not to be ranked high enough, while too high values, including $d = 0.85$, the consacrated value used in PageRank [22] cause the initial state to be too highly ranked with report to the other application states.
As shown in Section 35 the importance of elements in an AJAX Web page can be computed during crawling. A crawler can select just the most relevant elements to trigger events on and thus optimize the crawling. We perform AJAXHits on the complex news application (news items and calendar), the application structure of which is displayed in Figure 36.7. The application model of the complex news application has two areas, each one with two AJAX Events (prevNews, nextNews, prevCal, nextCal) that change the newsTitle, newsContents and calItem text parts. The events triggered in this application are displayed in Table 36.3.

<table>
<thead>
<tr>
<th>Element</th>
<th>Triggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>newsTitle</td>
<td>News - 1, News - 1</td>
</tr>
<tr>
<td>newsContents</td>
<td>News - 1, News - 1</td>
</tr>
<tr>
<td>calItem</td>
<td>News - 1, News - 1, News - 1, News - 1</td>
</tr>
</tbody>
</table>

Intuitively, we want to extract the most important events in this application. The outgoing and ingoing ranks of the trigger elements (the arrows) and of the event targets (the text boxes) is displayed in Table 36.3.

This table could be used as a source for smart crawling, however, it is independent of the amount of data in the application. For example, more news items means that the nextNews and prevNews triggers, which display the items, are more important. On the other side, the importance of target elements is dependent on the number of events influencing the different parts. For example, newsTitle and newsContents are influenced by two triggers, nextNews and prevNews, while calendarItem by four triggers. This can cause an element to be considered less important, despite its contribution to show more items of data. AjaxHITS can be used to reflect and adjust these considerations.

We performed three flavors of AjaxHITS, with the purpose of studying the influence of data size on the importance of elements:
### Table 36.3: Hubs and authorities ranks for DOM elements, computed using ingoing and outgoing links.

<table>
<thead>
<tr>
<th>ID</th>
<th>Hubs</th>
<th>Auth</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevNews</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>nextNews</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>prevCal</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nextCal</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>newsTitle</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>newsContent</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>callItem</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Running AjaxHITS correctly reflects the ranking of the four arrows and text boxes, dependent on the amount of data loaded in the application. **In case 1**, with the results presented in Table 36.4, a large number of news items causes the importance of events triggered by `prevNews` and `nextNews` to grow and also that of `newsTitle` and `newsContent`. On the contrary `prevCal`, `nextCal` are not ranked highly, neither is `newsContent`, despite the four events leading to it. **In case 2**, with the results in Table 36.5, a large number of calendar items raises the importance of the events triggered by `nextCal` and `prevCal` and that of the textbox `calItem`. On the contrary, the `nextNews`, `prevNews`, `newsTitle` and `newsContent` is decreased. **In case 3**, a similar number of news Items and calendar events leads to a similar ranking of all parts in the application DOM, as shown in 36.6. Target elements are however more influenced about the connectivity of the elements that from the amount of data affected by the element, which boosts.

**Smart Ranking Strategy** AjaxHITS can be used to correctly rank `trigger` elements based on the amount of data they load from the server. The top ranked trigger elements should be chosen in order (e.g, for Case 1, `prevNews` and `nextNews`). The target elements containing text are also ranked, but since text and search in AJAX application is the main driver of this work, the best strategy is to always crawl the target elements, otherwise this leads to an imbalance in the elements that ranked for content (e.g, `calendarItem` in Case 3).
### 36.6. Ranking Application Structure with AjaxHITS

Table 36.4: Hubs and authorities ranks for DOM elements with AJAXRank, for $N_{\text{news}} \gg N_{\text{events}}$ ($N_{\text{news}}=100, N_{\text{events}}=25$)

<table>
<thead>
<tr>
<th>ID</th>
<th>Hubs</th>
<th>Auth</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevNews</td>
<td>0.7</td>
<td>0</td>
</tr>
<tr>
<td>nextNews</td>
<td>0.7</td>
<td>0</td>
</tr>
<tr>
<td>prevCal</td>
<td>0.02</td>
<td>0</td>
</tr>
<tr>
<td>nextCal</td>
<td>0.02</td>
<td>0</td>
</tr>
<tr>
<td>newsTitle</td>
<td>0</td>
<td>0.69</td>
</tr>
<tr>
<td>newsContent</td>
<td>0</td>
<td>0.69</td>
</tr>
<tr>
<td>calItem</td>
<td>0</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 36.5: Hubs and authorities ranks for DOM elements with AJAXRank, for $N_{\text{news}} \ll N_{\text{events}}$ ($N_{\text{news}}=25, N_{\text{events}}=100$)

<table>
<thead>
<tr>
<th>ID</th>
<th>Hubs</th>
<th>Auth</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevNews</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>nextNews</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>prevCal</td>
<td>0.68</td>
<td>0</td>
</tr>
<tr>
<td>nextCal</td>
<td>0.68</td>
<td>0</td>
</tr>
<tr>
<td>newsTitle</td>
<td>0</td>
<td>0.06</td>
</tr>
<tr>
<td>newsContent</td>
<td>0</td>
<td>0.06</td>
</tr>
<tr>
<td>calItem</td>
<td>0</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 36.6: Hubs and authorities ranks for DOM elements with AJAXRank, for $N_{\text{news}} = N_{\text{events}}$ ($N_{\text{news}}=30, N_{\text{events}}=20$)

<table>
<thead>
<tr>
<th>ID</th>
<th>Hubs</th>
<th>Auth</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevNews</td>
<td>0.09</td>
<td>0</td>
</tr>
<tr>
<td>nextNews</td>
<td>0.09</td>
<td>0</td>
</tr>
<tr>
<td>prevCal</td>
<td>0.14</td>
<td>0</td>
</tr>
<tr>
<td>nextCal</td>
<td>0.14</td>
<td>0</td>
</tr>
<tr>
<td>newsTitle</td>
<td>0</td>
<td>0.62</td>
</tr>
<tr>
<td>newsContent</td>
<td>0</td>
<td>0.62</td>
</tr>
<tr>
<td>calItem</td>
<td>0</td>
<td>0.45</td>
</tr>
</tbody>
</table>
Chapter 37

Discussion: Integrating AJAXRank and HITS into existing Search engines

The proposed ranking schemes for AJAXRank, which take into account the application logic have been algorithmically described and experimentally proved on a small AJAX application. The ranking schemes take into account the links between the states, the importance of events between states and the actual data flow inside the application, when a crawler automatically (i.e., blindly) crawls an AJAX Web Site. Integrating the ranking schemes in a real application would require a further analysis of the actual usage of the application by the users. A full-fledged user study was out of scope of this work. However, it would be useful to study which events are actually most used by the users, such as by studying search engine logs [103], technique which would furthermore reflect in a different weight of the edges in the application structure and would most likely affect the final results returned by AJAXRank and HITS.
Chapter 38

Related Work

Ranking algorithms have different applications, including ranking Web pages, XML or desktop documents. On the other side, there exists work on the performance of the ranking algorithms. Ranking algorithms are divided into more categories, the most prominent being link-based ranking algorithms, content-based ranking algorithms, and mixed algorithms. PageRank [22] is a linked-based algorithm for Web Pages, as well as HITS [83]. AjaxRank adapts PageRank [22] and ranks states instead of pages. The model of an AJAX Web site differs from that of the Web since one cannot randomly jump to any state unless linked by events. Thus, AjaxRank favors the first state of an AJAX Application as opposed to PageRank. In AJAX Applications it is relevant to study the importance of different parts of the application structure, in order to perform smart-crawling. Although the original HITS is query-dependent, we adapt it in AjaxHITS and only use the link structure in the application for ranking and smart crawling. Other works such as [38] take a similar approach to AjaxHITS and try to determine, for traditional Web Sites, clusters of connected pages. Web log analysis [103] is a technique which can be used to assign weights based on user interactions. AJAXRank and AJAXHITS are not affected by SPAM, however PageRank Spam-detection algorithms such as SPAMRank [18] or RobustPageRank [11] are Structure-based algorithms such as XRank [73], developed for XML, adopt a similar strategy to ranking application structure and combine PageRank and content-based ranking. Content-based algorithms such as [57] rank topics but in a single AJAXSite this is not very relevant, and we used tf*idf content-based-ranking schemes instead, and integrated them into AjaxRank.
Chapter 39

Summary

This chapter presented an approach to retrieving and ranking search results in a Rich Internet Application. The approach is based on an extensive crawling of the application and on giving importance to states, events and transitions. The approach was compared to PageRank and HITS and it can be used to automatically infer information on the application structure, useful for example in enterprise search.

There are of course more avenues for future work. An enhanced analysis of the content and of the quality of the transitions is necessary: not all states are equally important and not all application parts are relevant for search. Current ranking methods apply to an automatic crawler. As mentioned in section 37, one way to further improve AJAXRank and HITS is to actually take into account actual user actions (log analysis) in order to better estimate the weight of different parts of the DOM in the whole application. Another way to partially solve this issue is through minimal collaboration of the application developer, which can create a more accurate and focused application model without blind crawling. Another point is that our the framework assumes there is no connection to the server side. Especially in case of Enterprise Search, where internal data (such as E-mails or entire databases) is accessible for crawling and indexing, this knowledge can be combined with the client-side search engine in order to obtain fast access to data and better understand all interactions. Constructing the scoring scheme can also be done lazily by using Top-k Techniques. Exploring the application structure is by itself a viable and promising direction. For example, the information generated by research or commercial Web Development Tools [52], [110], Frameworks such as JSF [79], Struts [105], or APIs such as the Google Web Toolkit [74] can be used to create an application model automatically. AJAX applications also start applying design patterns in the spirit of GoF [64]. Special search strategies relying on extracting relevant entities in the page (Information Extraction) can be applied in order to return even more specific results to the user. Last but not least, indexing more complex applications such as Google Maps [68] or using in the process the model supplied by the application developer or produced by the developing environment with which the AJAX application was created. The goal is: better ranked search results with minimal intervention.

A very relevant avenue for future work is the use of AJAXRank and AJAXHits in a real search engines and for real Web sites, where query log information is used for assigning correct and realistic weights to the edges of the modeled application structure. This results in an accurate estimation of Web site usage, and in a good estimation of the application structure's contribution to the whole application - like this, smart crawling can be realistically applied with the crucial purpose of efficiently crawl the increasing amount of AJAX content in applications.
Part V

Conclusions
39.1 Summary of the Work

This work presented an innovative concept, i.e. that of indexing application data. The challenges were to model the application data, index the content correctly, and create links inside the content of the application data. We addressed these challenges through first by providing an enhanced model of the data, either explicit (using declarative rules written by the application provider) or implicit (transition graph for AJAX Web Sites). Predicate-based indexing is a novel method of indexing application data, by enhancing traditional inverted files with predicate information, a format which applies to all patterns identified in application data. The format is extensible also applicable to, for example, JSP data.

Normalization avoids the need to instantiate potentially exponential application views and eliminates redundancies by encoding both data and rule information together linearly. Memoization is used to achieve the same effect on AJAX Web applications and to solve the challenge of duplicate elimination in case of multiple and granular events. Enhanced query processing is a polynomial operation which defines joins on the enhanced tuples in the inverted files. Results are links to the application data which can be used to “restore” the instance or the state. The operation of Result Aggregation is all the more difficult in case of AJAX Applications, where client-side logic and especially the state (i.e., variables, etc) must be reconstructed. The limitations we achieve are for now bordered by the capabilities of the server-side, which might maintain a state of its own, case that we are not capable to take into account.

All over this work, the tradeoff between performance and quality of results has been studied and the performance shown to be considerably better than any naïve approach while the overhead also minimal compared to Traditional Approach. The advantages are very clear for versioned data (e.g. Wikipedia). In the cases where the overhead cannot be minimized (e.g. YouTube’s User Comments), partial indexing has been studied as a method to improve quality and still maintain a reasonable overhead. Final considerations have been also given to a correct ranking model of instances, which takes full advantage of the new model of the data by adapting existing techniques such as $tf*idf$, PageRank and HITS into the AJAXRank and AJAXHITS algorithms. AJAX, and the enhanced Web models and capabilities available in Web pages which include application logic have been studied as a way to improve Hidden Web Crawl- ing, a well-known challenge of traditional Web Search, with relevant results. Using application logic not only as a purpose, but also as a mean to improve quality has been addressed by using AJAX Suggest capabilities of Web Sites in order to improve Hidden Web Search.

Overall, the work is at the border between databases and information retrieval, i.e., between the structured approach and a relaxed, content-described model. Explicit and implicit descriptions of the models are both present, and techniques from both worlds are used (join techniques as well as ranking techniques). Optimizations have been applied both for performance and scalability (a database and systems concept) as well as for result quality (an information retrieval) and the trade-off between them has been studied.

This work provides and defines for the first time not only the concepts for being dealing with application data, but also provides valid pointers for future work.
39.2 Future Work

After this work has provided pointers for being able to grasp these new research directions, there are several and interesting avenues for future work.

**Predicate-based Indexing.** This part of the work studied normalization as a way to reduce redundancies in the application data, making use of XML in the process. A formal comparison with XML Normal Forms of . As a step further to Partial Indexing technique used to study the trade-off between the amount of data to be indexed as opposed to the quality of results, the technique of Fuzzy Indexing can be used. This could mean that for the indexed data, several or all index entries are merged with the purpose of reducing index size and detail of the data description, by still maintaining good enough results. In this case, results may become imprecise, if, for example, two versions are always seen as one entity, but this situation is very much acceptable in many cases. Quantifying this trade-off is a promising research avenue. Generally, embedding entity and data mining extraction in the data search might also provide additional “application logic” information into the data, and therefore improving result quality. Finally, extending the description language in order to making it be able to describe more patterns or the same patterns in different data format is another practice-driven desiderata.

**AJAX Search.** AJAX Web Sites and Rich Internet Applications are in growing number in Internet, with advantages and drawbacks. In the same idea of studying trade-offs as presented in this work, a careful study of the two extremes: traditional (i.e, no AJAX) and extreme, AJAX-only, for a single Web Site. This would allow to study a performance trade-off in terms of the optimal amount of data to be “AJAXed” or the amount of parts from the application structure to be transformed using AJAX. If this type of study also includes metrics such as user satisfaction and user-interactivity, a better and more formal understanding of the current Web trends and their needs could be achieved.

**Ranking and Coverage.** AJAXRank and AJAXHITS algorithms suffer from the drawback that they do not yet incorporate the knowledge of actual user events in real applications, and provide, already more than their predecessors PageRank and HITS in traditional Web Search, . This important piece of information might be used for weighing the edges of application model and the application structure graph so that transitions are ranked not based on their connectivity or data transfer they are capable of transferring, but in terms of the amount of data that they are transferring in a real application scenario. Using the actual existing capabilities of AJAX Applications themselves, such as AJAX Suggest is a way to improve Hidden Web Search. Applying this technique in order to improve focused crawling (e.g., crawling with the purpose of highly precise results in a certain topic), and overall, in order to further increase coverage I consider as very important and promising avenues for future work.

Finally, this work has provided the toolkit for addressing, using, searching and researching application data.
List of Tables

5.1 Query Language on Application Data ........................................... 15
11.1 Problems with the Accuracy of State-of-the-art Desktop Search .................. 42
11.2 Experimental Data Collections ................................................. 42
11.3 Index Size(MB) ......................................................................... 44
11.4 Index Size With Positional Information(MB) ..................................... 44
11.5 Index Creation Time(seconds) .................................................... 45
11.6 Index Creation Time With Positioning Information (seconds) ..................... 45
11.7 Size of Normalized View and Materialized Instances ............................. 45
11.8 Query Processing Time(ms) without Positional Information ...................... 46
11.9 Query Processing Time(ms) with Positional Information ......................... 46

17.1 Annotation for the Transition Graph of an AJAX application .................... 74
19.1 Events and functionalities in the Javascript Invocation Graph .................... 82
19.2 Functions in the Javascript Invocation Graph on YouTube page ................ 83
19.3 Functions corresponding to events in the Javascript Invocation Graph on YouTube page . 83
19.4 The Hot Node Cache .................................................................... 84

20.1 Inverted File for YouTube AJAX Web Site. ........................................ 88
20.2 Query Processing on the AJAX news application. ................................ 88

21.1 YouTube Statistics. ..................................................................... 92
21.2 Crawling Time (secs.) for Trad. and AJAX Crawling. ............................ 93
21.3 Crawling Throughput (states or pages/second) for Trad. and AJAX Crawling. . 93
21.4 Application Model Size (MB) for Trad. and AJAX Crawling .................... 96
21.5 YouTube Queries. ........................................................................ 96
21.6 Query Processing Times (msecs) on YouTube. .................................... 97
### List of Tables

21.7 Queries reflecting the increase in precision with AJAX Search. ......................................................... 97
21.8 Recall for Trad. and AJAX Crawling. ........................................................................................................ 98
21.9 Recall for Trad. and AJAX Crawling. ........................................................................................................ 98
27.1 Metrics evaluating the Quality of a Hidden Web Search Engine. ............................................................. 119
29.1 Results from random sampling of Amazon’s book Website. .................................................................... 128
29.2 Metrics evaluating the Quality of a Hidden Web Search Engine. ............................................................. 129
33.1 Inverted File for AJAX news application (Figure 32.3). ........................................................................... 141
33.2 Query Processing on the AJAX news application. ................................................................................... 142
34.1 Annotation for the Application Structure of the AJAX news application (Figure 32.2). ......................... 147
34.2 Scores for Roles ..................................................................................................................................... 147
36.1 Number of AJAX Entities in the sample News Application. ................................................................. 154
36.2 Number of Iterations for AJAXRank for the AJAX news Application. .................................................... 156
36.3 Hubs and authorities ranks for DOM elements, computed using ingoing and outgoing links. ............... 158
36.4 Hubs and authorities ranks for DOM elements with AJAXRank, for \( N_{\text{news}} \gg N_{\text{events}} \) (\( N_{\text{news}}=100, N_{\text{events}}=25 \)) ................................................................................................. 159
36.5 Hubs and authorities ranks for DOM elements with AJAXRank, for \( N_{\text{news}} \ll N_{\text{events}} \) (\( N_{\text{news}}=25, N_{\text{events}}=100 \)) ................................................................................................. 159
36.6 Hubs and authorities ranks for DOM elements with AJAXRank, for \( N_{\text{news}} = N_{\text{events}} \) (\( N_{\text{news}}=30, N_{\text{events}}=20 \)) ................................................................................................. 159
List of Figures

1.1 The Search Engine View differs from the Application View of the user .................. xv
3.1 Versions function on a group_by and order_by scenario. ................................. 4
3.2 Cross-Application Search functions on top of a join semantics. ......................... 5
3.3 Enhanced Desktop Search ......................................................... 6

4.1 The Rule Language ........................................................................ 7
4.2 Excluded Example. ...................................................................... 8
4.3 Comment Example. .................................................................... 9
4.4 Alternative Example. ................................................................. 10
4.5 Version Example. .................................................................. 11
4.6 Placeholder Example. ............................................................... 12
4.7 Field Example. ..................................................................... 13

6.1 Normalized View - Alternatives .................................................. 18
6.2 Normalized View - Versions ..................................................... 20
6.3 Tagging Table (Alternatives Example). ......................................... 20
6.4 Tagging Table (Versions Example). ........................................... 20

7.1 Inverted File - Alternative (Figure 4.4). ........................................ 23
7.2 Inverted File for Versions (Figure 4.5). ........................................ 24

8.1 Dewey IDs for the versioned document in Figure 4.5 ....................... 28

9.1 Introducing Skip Spaces and Positions in Versioned Data (Figure 4.5) ............ 32
9.2 Inverted File - “Skip Entries” for Versioned Data (Figure 4.5). ....................... 32

10.1 Left deep tree for joining three keywords ...................................... 36
10.2 Processing Conjunctions with Multiple Rules. .................................... 37
### List of Figures

10.3 Basic Keyword Search in Versioned Data returns no consecutive positions ........................................... 38  
10.4 Performing Phrase Search with Skip Spaces in Versioned Data ......................................................... 39  
10.5 Left deep tree with SKIP_MERGE ...................................................................................................... 39  

11.1 Experimental Queries ........................................................................................................................... 43  
11.2 Precision and Recall for Experimental Data Sets .................................................................................. 43  
11.3 Indexing Time (secs) in case of Partial Indexing. ................................................................................... 47  
11.4 Index Size (MB) in case of Partial Indexing. .......................................................................................... 48  
11.5 Size (MB) of Normalized View and Materialized Instances in case of Partial Indexing. ......................... 48  
11.6 Query Processing Time (secs) in case of Partial Indexing. ................................................................... 48  
11.7 Recall (Returned Instances) in case of Partial Indexing. ..................................................................... 49  

12.1 Typical Structure of a Web Page in an Enterprise Web Application .................................................... 52  
12.2 The “Output” pattern ............................................................................................................................. 54  

13.1 Example of the PetStore Application ...................................................................................................... 57  
13.2 Indexing the Pet Store Web Application ............................................................................................... 58  
13.3 Example Query Result .......................................................................................................................... 60  

16.1 YouTube: Comments load using AJAX. ................................................................................................. 69  

17.1 Model of an AJAX Web page. .................................................................................................................. 73  
17.2 Event Structure in Javascript. ................................................................................................................ 73  
17.3 Model of an AJAX Web Site: AJAX Pages, hyperlinks and AJAX states. ................................................. 74  

19.1 Javascript Invocation Graph. .................................................................................................................. 82  
19.2 Javascript stack trace during AJAX Crawling (grows bottom up). ....................................................... 84  

20.1 Architecture of an AJAX Search Engine. ............................................................................................... 87  
20.2 Processing Conjunctions. ....................................................................................................................... 89  
20.3 Parallel AJAX Search architecture. ....................................................................................................... 90  

21.1 Distribution of YouTube videos based on number of comment pages. .................................................. 92  
21.2 Number of Pages, States and Events in YouTube. ............................................................................... 92  
21.3 Distribution of YouTube videos based on crawling times. ................................................................... 94  
21.4 Crawling and Network Time for increasing number of crawled videos. ............................................... 94  
21.5 Number of AJAX Events resulting in AJAX Calls with and without caching policy. ........................... 95
List of Figures

21.6 Recall for Traditional and AJAX Search in case of Partial Crawling. .......................... 99
21.7 Application Model Size(MB) in Case of Partial Crawling. ........................................... 99

23.1 AJAXNews: What does a traditional engine see? ....................................................... 104
23.2 Crawling an AJAX Application. Displaying the DOM. ............................................... 105
23.3 AJAXSearch in action. Results are links to application states. .................................. 107
23.4 Application with two sets of independent events. ....................................................... 107
23.5 Personalization: Correlations in the application model increase search precision and crawling performance. .................................................. 107
23.6 Independent events cause a state explosion. .............................................................. 108

26.1 Amazon's Suggest Functionality. .................................................................................. 116

27.1 Estimating the size N of a Corpus Using Random Sampling. ....................................... 120

28.1 Traditional Hidden Web Crawling and AJAXSuggest Crawling. ............................... 123

29.1 Effort (Number of submitted queries) in case of Traditional, AJAXSuggest and Combined Hidden Web Crawling. .................................................. 127
29.2 Coverage of the Database in case of Traditional, AJAXSuggest and Combined Hidden Web Crawling. .................................................. 127
29.3 Distribution of results returned by Traditional, AJAXSuggest and Combined Hidden Web Crawling. .................................................. 128
29.4 Return on Investment for Traditional, AJAXSuggest and Combined Hidden Web Crawling. .................................................. 129

32.1 The transition to AJAX changes the focus from HTML pages and links to AJAX States, Events and Application Logic. .................................................. 137
32.2 AJAX News Application and two news items. ............................................................ 138
32.3 The different triggers and targets in the Gmail application. ....................................... 139
32.4 Dimensions of Ranking in case of AJAX Applications. .............................................. 139

33.1 Entities describing an event in an AJAX Application. .................................................. 141
33.2 Computing Least Common Ancestor during Query Processing. .................................. 142

34.1 Application Model annotated with Source, Events, Targets and Actions. ...................... 145
34.2 Scores for Events and Actions. ...................................................................................... 145
34.3 Structure of the AJAX news application (Figure 32.2). .............................................. 146
34.4 DOM of the AJAX news application (Figure 32.2). .................................................... 146
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.5</td>
<td>Annotated Application Structure with (Event, Role, Operation) for the AJAX news application (Figure 32.2)</td>
<td>147</td>
</tr>
<tr>
<td>34.6</td>
<td>Propagating Ranking to ancestor elements.</td>
<td>148</td>
</tr>
<tr>
<td>35.1</td>
<td>Application Model for AjaxHITS</td>
<td>151</td>
</tr>
<tr>
<td>36.1</td>
<td>Application Model of News Application with five states.</td>
<td>153</td>
</tr>
<tr>
<td>36.2</td>
<td>Model of the experimental AJAX News Application.</td>
<td>154</td>
</tr>
<tr>
<td>36.3</td>
<td>Application with two sets of independent events.</td>
<td>154</td>
</tr>
<tr>
<td>36.4</td>
<td>AJAXRank vs. PageRank in the application model of Figure 36.1.</td>
<td>155</td>
</tr>
<tr>
<td>36.5</td>
<td>Influence of click-distance on AJAXRank and PageRank.</td>
<td>155</td>
</tr>
<tr>
<td>36.6</td>
<td>AJAXRank with different values of $d$, for the Application in Figure 36.1.</td>
<td>157</td>
</tr>
<tr>
<td>36.7</td>
<td>Application Structure of the extended News Application with calendar, with trigger-target dependencies and cardinalities.</td>
<td>157</td>
</tr>
</tbody>
</table>
Bibliography


[64] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, 1994.


[75] Philip J. Harding, Quanzhong Li, and Bongki Moon. XISS/R: XML Indexing and Storage System using RDBMS. In VLDB, 2003.


Cristian Ioan Duda

Personal Details
- **Nationality:** Romanian
- **Date of birth:** July 17th, 1980
- **Place of birth:** Aiud (Alba), Romania
- **Marital status:** Not married

Education

**ETH Zurich, Switzerland** 
PhD Candidate Computer Science 
*Coordinator:* Prof. Dr. Donald Kossmann 
*Thesis Topic:* Searching Application Data (Information Retrieval); 
*Other Projects:* Lazy /Adaptive XML Storage; XL (Web Service engine)

2002 – 2003 
**“Babes-Bolyai” University** 
M.Sc Computer Science 
*Major:* Databases in Internet and Electronic Commerce 
*Grade Point Average:* 10.0/10.0 
*Awards:* "Performance Scholarship" awarded by the University.

1998 – 2002 
**“Babes-Bolyai” University** 
Bachelor Degree 
*Major:* Computer Science 
*Grade Point Average:* 9.76 / 10 

Feb. 2001-Sept.2001 
**University of Pisa** 
Exchange Student In Computer Science 
*Note:* Erasmus exchange program. Communication - exclusively in Italian.

1994 – 1998 
**“Grupul Scolar Industrial Metalurgic”** 
BAC Degree (High school) 
*Major:* Computer Science 

Professional activity

June 2003 – August 2003 
**iQuest International** 
Internship (Software Development) 
*Cluj-Napoca, Romania*

2000 – 2002: 
**“Babes-Bolyai” University** 
Volunteer activity at the Communication Centre 
*Cluj-Napoca, Romania*

1995-1998 (High School) 
**Romania** 
Participation and awards to the Physics Olympics and Mathematics Olympics 
*Awards:* first prize (regional level), distinctions (national level) (Romania)

Languages
- English, German, French, Italian, French, Romanian (native)

Publications

2009: 
Cristian Duda, Donald Kossmann, Gianni Frey, Reto Matter, Chong Zhou, **AJAX Crawl: Making AJAX Applications Searchable**, ICDE 2009, Shanghai, China

2007: Cristian Duda, David Graf, Donald Kossmann, **Predicate-based Indexing of Enterprise Web Applications**, Demo Paper, CIDR 2007, Asilomar, California

Jens Dittrich, Cristian Duda, Björn Jarisch, Donald Kossmann, Marcos Vaz Salles, **Bringing Precision to Desktop Search: A Predicate-Based Desktop Search Architecture**, ICDE 2007, Istanbul, Turkey

2005: Cristian Duda, Donald Kossmann, **Adaptive XML Storage or the Importance of Being Lazy** (XIME-P 2005 Poster)  
Second International Workshop on XQuery Implementation, Experience, Perspectives, June 16-17, 2005, Baltimore, Maryland, USA.

2003: Cristian Duda, **UBBInfo Search: A First Step towards the Paperless Office**  
Second RoEduNet Conference, June 5-6, 2003, Iasi, Romania

---

**Research Projects (PhD)**

**Keyword Search in Application Data**  
**ETH Zurich, Switzerland**

**Description:** Current search engines are only syntactic. They do not understand the semantics embedded in the indexed data. (e.g., Annotations, Versions, Alternatives, etc.). This project aims at improving the results of current search engines by identifying these patterns, indexing and querying them efficiently.

**Implementation:** .Net Framework

**XML Store/XML Persistence Service**  
**ETH Zurich, Switzerland**

**Description:** A full-index approach for XML data is inefficient because of variations in the schema and in the application workload. This XML store adapts to the application workload and dynamically achieves optimization for reads or updates using partial, cache-like indexes.

**Implementation:** Java

**XL (part of team)**  
**ETH Zurich, Switzerland**

**Description:** Web service composition and integration language and platform, which uses XQuery as its native data model. XL is a main use case for the XML Persistence Service.

**Implementation:** Java, XQuery; XML, SOAP, WSDL, UDDI

---

**Theses**

**A Bitmap Index for XML and Path Expressions (Master Thesis, 2003)**  
**Coordinators:** Prof. Dr. Donald Kossmann (Technical University Munich), Prof. Dr. Leon Timbulea (“Babes-Bolyai” University, Cluj-Napoca Romania)

**Thesis download**(PDF):  

Containment-based identifier schemes for XML help define a bitmap index structure on XML documents. The index is probed using efficient ‘magic’ bit-wise operations.

**J Web Spacer (Bachelor Thesis, 2003)**  
**Technologies:** HTML, Java Servlets, JINI and JavaSpaces


**Thesis download**(PDF):  

The thesis describes a JavaSpaces-based extension of Java-servlet-compatible web servers. The result is a distributed Model-View-Controller architecture and provides natural scalability.