AJAX Crawl:
Making AJAX Applications Searchable

Reto Matter

Master Thesis
July 31, 2008

Supervised by:
Prof. Dr. Donald Kossmann
Cristian Duda
Databases and Information Systems Group
Abstract

Current search engines such as Google and Yahoo! are prevalent for searching the Web. Search on dynamic client-side web pages is, however, either inexistent or far from perfect, and not addressed by existing work, for example on Deep Web. This is a real impediment since AJAX and Rich Internet Applications are already very common in the Web. AJAX applications are composed of states which can be seen by the user, but not by the search engine, and changed by the user using client-side events. Current search engines either ignore AJAX applications or produce false negatives. The reason is that crawling client-side code is a difficult problem that cannot be solved naively by invoking user events. The challenges are: lack of caching, duplicate states detection, very granular events, reducing the number of AJAX calls and infinite event invocation. This thesis sets the stage for this new search challenge and proposes a solution: it shows how an AJAX web application can be crawled in the granularity of the application states. A model of AJAX Web Sites is presented. An AJAX Crawler and optimizations for caching and duplicate elimination are defined, and finally, the gain in search result quality and corresponding performance price are evaluated on YouTube, a real AJAX application.
The text of this thesis is based on a paper submission which I am a co-author of [17]. The paper is a submission for the ICDE 2009 conference. Also, this work is an extension of the achievements of Gianni Frey’s Master Thesis [20], which was finished in November of the year 2007. It therefore follows that there exist some redundancies between my thesis and the mentioned work.
Danksagung

Ich möchte es nicht versäumen, einigen Personen, die mich auf meinem Weg begleitet haben, zu danken. Zum Beispiel möchte ich Herrn Prof. Dr. Donald Kossmann herzlich dafür danken, dass er sich bereitwillig zur Verfügung gestellt hat, mein Mentor zu sein, und dass er mir ermöglicht hat, ja, mich ermuntert hat, meine Masterarbeit bei ihm in der Databases and Information Systems Group zu schreiben.

Die engste Zusammenarbeit während der vergangenen sechs Monate fand mit Cristian Duda statt; wir trafen uns oft, um gemeinsam die Fortschritte oder Probleme zu besprechen. Er ist immer auf meine Vorschläge, Ideen oder Bedenken eingegangen, hat sich viel Zeit für mich genommen und ist mir jederzeit mit Rat und Tat zur Seite gestanden. Mir werden diese Treffen wie auch die gesamte Zusammenarbeit mit ihm fachlich wie menschlich in bester Erinnerung bleiben.

Bis zu seiner Heimkehr nach China war auch Chong Zhou an diesen Treffen zugegen. Auch ihm möchte ich für seine Unterstützung und seine Ideen, die er eingebracht hat, danken.


# Contents

Abstract 3
Note 5
Danksagung 7

1 Introduction 13
  1.1 Motivating Example: YouTube Comments 15
    1.1.1 Traditional Search 15
    1.1.2 AJAX Search 15
  1.2 Contributions 16

2 Modeling AJAX 17
  2.1 Event Model 17
  2.2 AJAX Page Model 17
  2.3 Modeling AJAX Web sites 19

3 AJAX Crawling 21
  3.1 Crawling Algorithm 21
  3.2 Addressing Crawling Challenges 23

4 A Heuristic Crawling Policy for AJAX Applications 25
  4.1 Javascript Invocation Graph 26
  4.2 Optimized Crawling Algorithm 27
  4.3 Simplifying Assumptions 30
  4.4 Implementation 31
    4.4.1 Detecting new Hot Nodes 31
    4.4.2 Detecting already cached Hot Nodes 34

5 An AJAX Search Engine 37
  5.1 AJAX Crawler 37
Contents

5.2 Indexing ........................................... 37
5.3 Query Processing .................................. 38
  5.3.1 Processing Simple Keyword Queries ................. 38
  5.3.2 Processing Conjunctions ............................ 39
  5.3.3 Ranking ......................................... 39
5.4 Result Aggregation .................................. 41
5.5 Parallelization ..................................... 42

6 Parallelization of the AJAX Crawler 43
  6.1 Basic Idea And Architectural Considerations ......... 44
  6.2 The Precrawling Phase .............................. 44
    6.2.1 The Precrawler class ........................... 46
    6.2.2 The URLPartitioner class ....................... 46
  6.3 The Crawling Phase ................................ 46
    6.3.1 The MPAjaxCrawler class ........................ 47
    6.3.2 The SimpleAjaxCrawler class .................... 49
  6.4 The Indexing Phase ................................ 49
  6.5 The Query Processing Phase ......................... 50
    6.5.1 Merge Phase ................................... 50
    6.5.2 Ranking ....................................... 51
  6.6 Result Aggregation ................................ 54

7 Experimental Results 55
  7.1 Experimental setup .................................. 55
    7.1.1 YouTube Datasets ............................... 55
    7.1.2 Algorithms ..................................... 58
  7.2 Crawling Performance ............................... 58
    7.2.1 Overhead of AJAX Crawling ....................... 58
    7.2.2 Distribution of Crawling Times .................... 59
    7.2.3 Influence of Number of States .................... 59
  7.3 Effects of Caching ................................ 60
  7.4 Parallelization .................................... 62
    7.4.1 Performance of Parallelized Crawling ............. 64
    7.4.2 Parallel vs. Non-parallel ....................... 64
  7.5 Query Processing .................................. 65
    7.5.1 Queries ........................................ 65
    7.5.2 Query Processing Time ........................... 66
  7.6 Setting Crawling Threshold ......................... 67
Contents

7.7 Improvement in Search Quality ............................................ 68

8 Infrastructural Setup ......................................................... 71

8.1 Precrawling Phase ......................................................... 71
8.1.1 [org.ajax.crawler.]Precrawler .................................. 71
8.1.2 [org.ajax.crawler.]URLPartitioner ............................. 72

8.2 Crawling Phase ............................................................. 74

8.3 Indexing and Query Processing ......................................... 76

8.3.1 Building a New Index ............................................... 76
8.3.2 Loading an Index .................................................... 77
8.3.3 Query Processing .................................................... 77

9 Related Work .................................................................. 79

10 Conclusions and Future Work .......................................... 81

11 Bibliography .................................................................. 83
Chapter 1

Introduction

Google Mail[22], Yahoo! Mail[37], Google Maps[23] are well-known AJAX (“Asynchronous JavaScript and XML”) applications. One of their goals is to enhance the user experience by running client code in the browser instead of always refreshing the web page. Another goal is minimizing server traffic. To achieve this, an AJAX application is able to refresh only parts of the web page by fetching new data asynchronously from the server, without changing the URL of the web page. Traditional Web Applications such as Amazon[2] 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 an URL.

Current search engines ignore AJAX content since crawling AJAX is a difficult problem, due to the following challenges:

1. **No Caching/pre-crawl.** Current search engines pre-cache the Web and crawl locally. Events, however cannot be cached.

2. **Duplicate states.** Several events can lead to the same state, because the same underlying Javascript function is used to render the content. This is crucial in optimizing and reducing the size of the application view.

3. **Very granular events.** This can lead to a large set of very similar states.

4. **Infinite event invocation.** Since AJAX applications consist of events, the application view may never lead to a closure.
Chapter 1 Introduction

We will address these issues as we will show further. Because of these challenges, current search engines do not crawl AJAX and provide – among others – following workarounds:

- **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 Search.

However, these workarounds are not generic, and second, not accurate enough or may require the collaboration of the application provider. Furthermore, all approaches of the state-of-the-art are insufficiently precise, as shown in the following example:
1.1 Motivating Example: YouTube Comments

YouTube\textsuperscript{38} is one website 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 1.1 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, 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, \texttt{next} and \texttt{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 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.

1.1.1 Traditional Search

We focus on boolean retrieval. YouTube users can submit the boolean query \texttt{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.

1.1.2 AJAX Search

In order to find the title of the new video, an unprepared user should be able to search for the boolean query \texttt{Q2: "Morcheeba mysterious video"} and get this video as a result. This only works if \textbf{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 which start including AJAX content in their traditional web pages.

1.2 Contributions

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

- **Modeling AJAX.** We propose a the 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 a YouTube subset.

The rest of the thesis is organized as follows: Chapter 2 models AJAX Web Sites, pages and events. Chapter 3 and chapter 4 are the main contribution of the thesis and describe the crawling algorithm and a solution to the problem of detecting duplicates and caching in AJAX crawling. Chapter 5 describes the overall architecture of a search engine, while chapter 6 concentrates on the aspect of the parallelization of the crawler. Experimental results are shown in chapter 7, while chapter 9 and chapter 10 contain conclusions and future work.
Chapter 2

Modeling AJAX

As mentioned before, this thesis 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.

2.1 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 2.1. The `onClick` event triggered on the `<div id="nextArrow">` HTML element `source`, applied to the `doc.comment` element (target). The content of the targets changes using the action: `doc.comment.innerHTML=...`. We will use these notations throughout this work.

```
<div id="nextArrow" onClick="doc.comment.innerHTML=new_comment_page"/>
```

Figure 2.1: Event Structure in Javascript.

2.2 AJAX Page Model

Section 1 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 comment pages). In particular it is an automaton, a \textbf{Transition Graph}. The Transition Graph contains all application entities (states, events, transitions) as annotations. It is defined by:

- **Nodes.** The Nodes are application states. An application state is 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 2.2, which models the \textit{next} and \textit{previous} events invoked on the corresponding buttons of the YouTube application. Traversals of this graph can be entered in a table constructed as Table 2.1. For each transition between Start State and End State, the Source, Event, Target(s) and Action(s) are entered in the table. In Table 2.1, 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.
2.3 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 2.3.

Figure 2.3: Model of an AJAX Web Site: AJAX Pages, hyperlinks and AJAX states.

The difference to the traditional Web is that the user may trigger events in the same page (such as \texttt{next} and \texttt{prev}) which generate new application states. In order to complete the comparison, the transitions caused by the events which lead to some other state without changing the URL may be called AJAX links.

As opposed to this, traditional Web Sites are characterized just by a graph of \textit{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.
Chapter 3

AJAX Crawling

The contribution of this thesis 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.

3.1 Crawling Algorithm

The role of the AJAX Crawling algorithm is to build the model of the AJAX Web Site. An example of an AJAX Web Site is the web page at the URL http://www.youtube.com/-watch?v=w16JlLSySWQ. 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 the YouTube example mentioned above, indexing all comment pages of a video).

We detail the crawling algorithm for AJAX applications in Algorithm 3.1.1.

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 built (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 12). As mentioned in Section 2, a transition is annotated with the event information: source, trigger, action(s) and modif(s). After a new state has been reached,
Algorithm 3.1.1 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 = appModel.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
3.2 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 2.2, a state can be repeated 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:

- **Differences from Traditional Crawling.** 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.

- **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 chapter 4.
Chapter 3 AJAX Crawling

- **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 4

A Heuristic Crawling Policy for AJAX Applications

Chapter 3 presented a basic traditional crawling algorithm. A problem that occurs in crawling is the network time needed to fetch pages. In case of AJAX Crawling, multiple individual events per page lead to fetching network content. The problem of connections is even bigger. Traditional search engines deal with this problem by pre-caching the Web and crawling locally. This procedure is just partly possible in case of AJAX since dynamic content is constantly fetched from the server. The problem of caching is, however, also not trivial. In the traditional case, two pages can be checked to be identical using a single URL, while in the case of AJAX the URL is unchanged.

This chapter addresses this problem and uses the following observation. In the crawled AJAX applications, the structure of the application is usually stable, and contains for example a menu, present in all states, and a dynamic part. The YouTube Page for example contains for each page several links to the next/previous page or direct jump to one of the immediately consecutive pages. This leads to a frequent situation: using the menu items reload the same content from the server, and leads to a state that has already been reached. The basic AJAX Crawler from chapter 3 is only able to decide if it just reached a state that it had already reached before by comparing the current state with all the states it has crawled so far. But because it’s a very expensive task for the crawler to always open a network connection and retrieve – in this particular case known and because of this useless – data from the server, we try to avoid network communication as much as possible. In the ideal case new data which leads to a new state is only fetched once. The optimized AJAX Crawler should be able to somehow detect in advance that some AJAX event leads to a state that it has already reached and intercept that event in order to prevent itself from doing the network communication.
In the following sections we show a solution to this problem by identifying the same state without fetching the respective content from the network.

### 4.1 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 4.1.

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 4.1. Functions in the Javascript code can be invoked either directly by event triggers (event invocations) or indirectly by other functions (local invocations). Table 4.2 lists some functions appearing in the YouTube page. In order to fulfill their functionality in an AJAX application the events call, directly or indirectly, other functions which fetch content from the server. The dependencies in the code are listed in Table 4.3. We call the functions that fetch content from the server **Hot Nodes** and a call to a hot node **Hot Call**. A single function fetches content from the server, i.e., `getURLXMLResponseAndFillDiv(URL, ID)`. In AJAX, the same function...
4.2 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 4.2.1. The main points of the algorithm are:

<table>
<thead>
<tr>
<th>Event #</th>
<th>Functionality</th>
<th>Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;init page&quot;</td>
<td>onload</td>
</tr>
<tr>
<td>2</td>
<td>&quot;next page&quot; (from page 1)</td>
<td>onclick</td>
</tr>
<tr>
<td>3</td>
<td>&quot;prev page&quot; (from page 2)</td>
<td>onclick</td>
</tr>
<tr>
<td>4</td>
<td>&quot;jump to page 2&quot;</td>
<td>onclick</td>
</tr>
<tr>
<td>5</td>
<td>&quot;jump to page 3&quot;</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 4.1: Events and functionalities in the Javascript Invocation Graph.

<table>
<thead>
<tr>
<th>ID</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>showLoading(ID)</td>
</tr>
<tr>
<td>b</td>
<td>togglePanel(ID)</td>
</tr>
<tr>
<td>c</td>
<td>urchinTracker(ID)</td>
</tr>
<tr>
<td>d</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>getURLXMLResponseAndFillDiv(URL, ID)</td>
</tr>
</tbody>
</table>

Table 4.2: Functions in the Javascript Invocation Graph on YouTube page.

can be invoked in order to fetch the same content from the server from different comment pages, as shown in Table 4.3. 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.
Chapter 4 A Heuristic Crawling Policy for AJAX Applications

<table>
<thead>
<tr>
<th>Event #</th>
<th>Functionality</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;init page&quot;</td>
<td>init()</td>
</tr>
<tr>
<td>2</td>
<td>&quot;next page&quot; (from page 1)</td>
<td>showLoading('recent_comments')</td>
</tr>
<tr>
<td>2</td>
<td>&quot;next page&quot; (from page 1)</td>
<td>getURLXMLResponseAndFillDiv (...'action_get_comments=1&amp;p=2'...)</td>
</tr>
<tr>
<td>2</td>
<td>&quot;next page&quot; (from page 1)</td>
<td>urchinTracker(...)</td>
</tr>
<tr>
<td>3</td>
<td>&quot;prev page&quot; (from page 2)</td>
<td>showLoading('recent_comments')</td>
</tr>
<tr>
<td>3</td>
<td>&quot;prev page&quot; (from page 2)</td>
<td>getURLXMLResponseAndFillDiv (...'action_get_comments=1&amp;p=1'...)</td>
</tr>
<tr>
<td>3</td>
<td>&quot;prev page&quot; (from page 2)</td>
<td>urchinTracker(...)</td>
</tr>
<tr>
<td>4</td>
<td>&quot;jump to page 2&quot;</td>
<td>showLoading('recent_comments')</td>
</tr>
<tr>
<td>4</td>
<td>&quot;jump to page 2&quot;</td>
<td>getURLXMLResponseAndFillDiv (...'action_get_comments=1&amp;p=2'...)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.3: Functions corresponding to events in the Javascript Invocation Graph on YouTube page.

- **Step 1: Identifying Hot Nodes.** The crawler tags the Hot Nodes, i.e., the functions that directly contain AJAX calls. In the YouTube application, there is one Hot Node, i.e., the function `getURLXMLResponseAndFillDiv(URL, ID)`. (line 34)

- **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). This step uses the current runtime stack trace.

- **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`...
Algorithm 4.2.1 Breadth-first Heuristic AJAX Crawling Algorithm

1: Cache hotNodesCache = {}
2: Function init(url) {...} end Function
3: Function crawl(State s)
4: for all Event e ∈ s do
5: manageFunction(e.function)
6: if dom.hasChanged() then
7: State newState = new State(dom)
8: if appModel.contains(newState) then
9: newState = appModel.get(newState)
10: end if
11: Transition t = new Transition(e.source, e.trigger, e.action*, e.modif*)
12: appModel.add(t, s, newState)
13: appModel.rollback(t)
14: end if
15: end for
16: for all Transition t ∈ (s, s1) do
17: Crawl s1 {Breadth-first traversal of reachable states}
18: end for
19: end Function

20: Function invokeFunction(Function f)
21: Statement[] statements = getStatements(e)
22: for all Statement stmt ∈ statements do
23: if stmt is function then
24: callStack.push(stmt, e.args)
25: result = manageFunction(stmt)
26: else
27: Execute Statement e
28: end if
29: end for
30: end Function

31: Function manageFunction(Function f)
32: Statement[] statements = getStatements(f)
33: for all Statement s ∈ statements do
34: if s is AJAXCall then
35: Function topEntry = callStack.top()
36: if not hotNodeCache.contains(topEntry.function, topEntry.arguments) then
37: hotNodes = hotNodes U topEntry.function
38: result = callAJAX(entry.function)
39: insert (topEntry.function, topEntry.arguments, result) into hotNodeCache
40: else
41: result = hotNodeCache.lookup(topEntry.function, topEntry.arguments)
42: end if
43: return result
44: end if
45: return
46: result = invokeFunction(e)
47: else
48: for all Function f ∈ s do
49: s.f = manageFunction(f, f.args)
50: end for
51: Execute statement s
52: end if
53: end if
54: end for
55: end Function
Chapter 4 A Heuristic Crawling Policy for AJAX Applications

<table>
<thead>
<tr>
<th>Hot Node</th>
<th>Parameters</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$P_1$</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>$P_2$</td>
<td>...</td>
</tr>
<tr>
<td>B</td>
<td>$P_1$</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.4: The Hot Node Cache

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 $\text{getURLXMLResponseAndFillDiv}(\text{URL, ID})$ 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 chapter 7.

4.3 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.

- **Statelessness Of The Server.** We assume that two server calls of the same Hot Node with the same parameters return always the same result, i.e, the Javascript function $\text{getURLXMLResponseAndFillDiv}(\text{url1, id1})$ returns always the same result.

- **No Forms.** A lot of AJAX applications (such as Google Suggest [24]) 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 [23] are not text-based, but image-based. We limit ourselves to text-based retrieval.

### 4.4 Implementation

In this section we present some details about the implementation of the Hot Node approach introduced above. As mentioned in chapter 1, we adopted the basic AJAX Crawler from [20]. We therefore won’t repeat all implementation details of the whole crawler here, but only those that are necessary to understand how the Hot Node approach was implemented.

For the handling of the HTML and the execution of Javascript, our AJAX Crawler makes use of the COBRA Toolkit [16] and the Rhino framework [31]. The work for coping with the Hot Nodes is divided into two parts. One is the detection and the memorizing of new Hot Nodes, a duty that is taken care of by the *AJAXDocument* class, and the other is to notice that the current Javascript call conforms to a already known Hot Node. Section 4.4.1 deals with the first, section 4.4.2 explains the second part.

#### 4.4.1 Detecting new Hot Nodes

The “main” class of the basic AJAX Crawler, the *AJAXDocument* class, does the bookkeeping of the Hot Nodes. It therefore holds an internal cache of all the Hot Nodes that already happened and their respective results fetched from the server. It also points to a reference of an instance of type *HTMLDocumentImpl* which represents the current DOM tree of the AJAX Web Site. The *XMLHttpRequest* class – the Java equivalent of the Javascript object with the same name – also holds a reference to the same *HTMLDocumentImpl* object instance. Whenever an AJAX call occurs in the Javascript code of the current DOM, the *XMLHttpRequest.open()* method is called to open a connection to a given URL. Using the well-known Observer pattern ([21]) with the *HTMLDocumentImpl* class as the Subject and the *AJAXDocument* class as the Observer,
we intercept the execution of this `open()` method and inform the `AJAXDocument` object. Now the `AJAXDocument` knows that the execution of a Hot Call is being made, i.e. execution has reached a Hot Node, and can appropriately react to that situation:

![UML Sequence Diagram](image)

**Figure 4.2: UML Sequence Diagram explaining the mechanism of detecting new Hot Nodes.**

1. An instance of the `StackInfo` class is created. The method `getHotnodeInfo()` of that object requests the current Javascript Call Stack from the `Interpreter` class, determines the topmost, i.e., currently executing, function and returns the name and the actual parameters of it in a special format as `String`.

2. With that Hot Node information, the internal Hot Node cache can be searched. If the search returns nothing, then we know that we have encountered a new Hot Node. Otherwise, this Hot Call has already been executed and nothing else has to be done by the `AJAXDocument` object.

The UML Sequence Diagram of Figure 4.2 shows schematically how the Hot Node detection mechanism works.

It follows an example to make further clear how the Hot Node information is determined by the `StackInfo` class: Consider the following `onClick` event within an AJAX Web
The corresponding excerpt from the Javascript code file of YouTube looks as follows:

```javascript
<script type="text/javascript">
...
function showLoadingIcon(div_id) { ... }

function getUrlXMLResponseAndFillDiv(url, div_id) {
    getUrl(url, true);
}

function getUrl(url, async) {
    ...
    var xmlHttpReq=getXmlHttpRequest();
    xmlHttpReq.open("GET",url,async);
    xmlHttpReq.send(null);
    ...
}

function urchinTracker (a) { ... }
...
</script>
```

The `showLoading()` and the `urchinTracker()` functions are not interesting in terms of the Hot Node detection mechanism, but the `getUrlXMLResponseAndFillDiv()` function is. This function calls `getUrl()` which eventually opens a network connection and, therefore, is a Hot Node. The schematic representation of the current Call Stack during the execution of `getUrl()` looks as in Figure 4.3.

The `StackInfo.getHotNodeInfo()` method extracts the function name and the actual parameter values of the topmost function in the call stack, in our example of the `getUrl()` function.
4.4.2 Detecting already cached Hot Nodes

In order to be able to detect a Hot Node that is already cached by the application, one requires sort of a debugger functionality, i.e., we have to know at every moment during the execution of a Javascript script what function is being currently executed, and what are the actual parameter values of that respective call. Fortunately, the Rhino environment allows the seamless integration of a Javascript debugger. For this to take place we had to come up with our own implementation of the Debugger4 and the DebugFrame5 interfaces. The resulting classes are called JSDebugger6 and DebugFrameImpl7.

For the detection of Hot Nodes the most important class is DebugFrameImpl. It provides implementations for the following method signatures:

- onEnter(Context c, ..., Object[] args);
- onLineChange(Context c, int lineNumber);
- onExceptionThrown(Context c, Throwable ex);
- onExit(Context c, boolean byThrow, Object resOrEx);

A JSDebugger instance can be attached to the current Context. From this point the DebugFrameImpl gets informed by the Interpreter8 whenever Javascript execution enters or exits a function, proceeds to the next code line or throws an exception by calling the corresponding methods. We are mainly interested in the onEnter() method, for that’s the point where we know the name and the actual parameter values of the currently executed Javascript function. All we have to do then is gather this information and look it up in the Hot Node cache of the AJAXDocument instance. If the search is successful we are to execute a Hot Node and instead of the following XMLHttpRequest.open()
4.4 Implementation

and XMLHttpRequest.send() we deliver the cached result. If the search returns false, the AJAX call is actually made. Figure 4.4 shows how the existing Hot Node detection mechanism works.

![UML Sequence Diagram explaining the detection of existing Hot Nodes.](image)

Figure 4.4: UML Sequence Diagram explaining the detection of existing Hot Nodes.

This chapter focused on the Hot Node detection mechanism, an optimization of the AJAX Crawler that reduces network calls. Together with the parallelized architecture presented in chapter 6, Hot Node detection improves the overall performance of the crawling process. We showed how this impacts the basic Crawling algorithm 3.1.1 from chapter 3 and presented the way how we implemented this mechanism in our AJAX Crawler. In the following we will embed the AJAX Crawler in an AJAX Search Engine.
Chapter 4  A Heuristic Crawling Policy for AJAX Applications
Chapter 5  

An AJAX Search Engine

The purpose of this thesis is to crawl AJAX content with the purpose 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 5.1. The components are described below.

5.1 AJAX Crawler

The main contribution of this thesis, 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 3 and 4. The AJAX Models are then used to build the index.

5.2 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[4].

Figure 5.1: Architecture of an AJAX Search Engine.
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 5.1. 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.

<table>
<thead>
<tr>
<th>Word</th>
<th>URI</th>
<th>State</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>morcheeba</td>
<td><a href="http://www.youtube.com/watch?v=w16JlLSySWQ">www.youtube.com/watch?v=w16JlLSySWQ</a></td>
<td>s1</td>
<td>1</td>
</tr>
<tr>
<td>morcheeba</td>
<td><a href="http://www.youtube.com/watch?v=w16JlLSySWQ">www.youtube.com/watch?v=w16JlLSySWQ</a></td>
<td>s2</td>
<td>1</td>
</tr>
<tr>
<td>morcheeba</td>
<td><a href="http://www.youtube.com/watch?v=Iv5JXxME0js">www.youtube.com/watch?v=Iv5JXxME0js</a></td>
<td>s1</td>
<td>2</td>
</tr>
<tr>
<td>mysterious</td>
<td><a href="http://www.youtube.com/watch?v=w16JlLSySWQ">www.youtube.com/watch?v=w16JlLSySWQ</a></td>
<td>s1</td>
<td>1</td>
</tr>
<tr>
<td>singer</td>
<td><a href="http://www.youtube.com/watch?v=w16JlLSySWQ">www.youtube.com/watch?v=w16JlLSySWQ</a></td>
<td>s2</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 5.1: Inverted File for YouTube AJAX Web Site.

5.3 Query Processing

As presented in the Introduction, when searching an AJAX application, a user is interested in obtaining the states into 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 conjunctions queries.

5.3.1 Processing Simple Keyword Queries

The index constructed in Section 5.2 can be used to extract this information as shown in Table 5.2. 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).
5.3.2 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 5.2 let’s take the query \( Q_3 \) 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 5.2) and of keyword singer are presented and merged in the first row of Figure 5.2. 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 \( Q_3 \) is the tuple \( \langle URL_1, s_2 \rangle \), corresponding to the second page of comments of the video presented in chapter 1.

5.3.3 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). More information can be found in [18].

Right now we work with the weighted sum of four coefficients that make up the total ranking value of a result:

1. PageRank. The PageRank is an URL based measure and therefore the same for every state within the same Web page.
2. **AJAXRank.** The AJAXRank [20] is a measurement for the ranking order of the states within one AJAX Web page.

3. **tf/idf.** Term frequency in AJAX applications is defined as follows:

$$tf(k, s) = \frac{|\{ t | t \in s \land t = k \}|}{\sum_{t_k \in s} |\{ t | t \in s \land t = t_k \}|}$$  \hspace{1cm} (5.1)

where \( k \) denotes a keyword, \( s \) denotes a state and \( t \) denotes any term within \( s \) (note that in the formula of \( tf(k, s) \) we assume bag semantics from multiset theory). The term \( tf(k, s) \) therefore only depends on the keyword(s) within one particular state. The **Inverse Document Frequency** is defined as (\( D \) is the set of states):

$$idf(k) = \log \frac{|\{ s | s \in D \}|}{|\{ s | s \in D \land k \in s \}|}$$ \hspace{1cm} (5.2)

Hence the Inverse Document Frequency \( idf(k) \) of keyword \( k \) is the logarithm of the total number of states divided by the number of states that contain \( k \). We want to emphasize that opposed to the traditional definitions of the \( tf/idf \) measure, in our definition states take over the role of documents.

4. **Term proximity.** This measure rewards the proximity of the terms in the query compared to the occurrences of the terms in the document. The highest value is given to documents which contain the query as is. For example, if the sequence of the keywords differ or the terms are more spread in the document, the coefficient gets smaller.

The overall rank \( R_{res,q} \) of a search result \( res \) of a query \( q \) is then the weighted sum of the above coefficients:

$$R_{res,q} = w_1 \cdot P(url(s)) + w_2 \cdot A(s) + w_3 \sum_{k \in q} tf(k, s) \cdot idf(k) + w_4 \cdot T(q, s)$$ \hspace{1cm} (5.3)

with \( P(url(s)) \) being the PageRank of the URL of the state \( s \), \( A(s) \) the AJAXRank of state \( s \), \( T(q, s) \) the term proximity value of query \( q \) in \( s \) and \( w_i, i \in \{1..4\} \) the corresponding weights.
5.4 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 is 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).
5.5 Parallelization

Because crawling AJAX faces the difficulty of not being able to really cache dynamic web content, except for the heuristics discussed in chapter 4, network connections must continuously be created. This drastically increases the crawling time, as we will also show in chapter 7. To target the improvement of the overall crawling performance of the application we decided to introduce some parallelization into the architecture of the AJAX Crawler. The steps that have been taken to do so are described in chapter 6.
Chapter 6

Parallelization of the AJAX Crawler

In this chapter we explain how we parallelized our AJAX Crawler in order to increase crawling throughput.

As we mentioned in chapter 5 and will show in chapter 7, the performance of the AJAX Crawler without any optimizations is in the best case mediocre. We identified two prime reasons why this is the case:

1. **Network communication.** Because we crawl not only the bare web page but also its AJAX content, new network connections have to continuously be made. We partly address this problem by the Hot Node approach described in chapter 4. But still, compared with traditional crawling, AJAX crawling on average needs more network connections per page. Furthermore the instantiation and initialization as well as the request to the server itself are expensive operations, which often lead to a point where the process is doing nothing else than waiting, e.g, for the response from the server.

2. **Maintaining the application model.** The application model for AJAX Web Sites is considerably more complicated than that of traditional web pages. A lot of additional information about DOM elements, states, Javascript variables and events and the Transition graph has to be stored. It lies in the nature of this that the effort to maintain the application model(s) becomes a more and more expensive task, in terms of not only organizing this additional knowledge, but also in terms of memory consumption.

The following sections introduce the parallel architecture of the AJAX Crawler.
6.1 Basic Idea And Architectural Considerations

We base the parallelization of the AJAX Search engine on the following observation: crawling one AJAX Web page is completely independent of crawling another AJAX Web page. That means that no communication is necessary between these activities. This conforms to a loosely coupled SPMD (Single Program Multiple Data) programming paradigm known from the theory of parallel computing and makes it a particularly qualified candidate for parallelization.

The effectiveness of parallelization increases the less communication is taking place between the concurrent processes. However, the hyperlink structure is still the only structure shared by all processes. Therefore, we removed the building of the hyperlink structure from the main crawling process; parallelization would become complete since any communication between different crawling processes is avoided.

We achieved this by inserting an additional phase at the beginning of the whole process of crawling, called Precrawling Phase. During this phase the traditional, linked-based web site structure (as presented in chapter 2) is built. 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 submitted to a set of parallel crawlers.

Figure 6.1 shows the parallel architecture of the AJAX Search Engine. Each crawler applies the crawling algorithm of Sections 3 and 4, and builds for each crawled page the AJAX Model (i.e., the transition graph). More indexes are then built, one from each disjunct set of AJAX Models. Query processing is then performed by Query Shipping, computing the results from each Index, as explained in Section 5.3, and then performing a merge of the individual results from each index, returning the final result list to the client. The following sections detail the different phases.

6.2 The Precrawling Phase

The Precrawling Phase is responsible for building the hyperlink structure between the Web pages to crawl, and the computation of the PageRank values of these pages.

Additionally, the list of URLs to crawl is partitioned in smaller lists of URLs. These partitions serve as input for a set of parallel AJAX Crawlers. We therefore divided the
6.2 The Precrawling Phase

![Diagram of Parallel AJAX Search architecture](image)

Figure 6.1: Parallel AJAX Search architecture.
work of the Precrawling Phase further down into two sub phases handled by two classes: The Precrawler\(^1\) class and the URLPartitioner\(^2\) class.

### 6.2.1 The Precrawler class

The Precrawler class is given a starting point, i.e., a URL from where it has to start and a maximum number of pages to precrawl. When started, it connects to the initial page, parses it and finds the outbound links. These found links are then crawled and so forth. When the maximum number of pages is reached, the Precrawler stops looking further. The outbound link structure is then used to build the PageRank structure and the PageRank values are computed. After the computation the Precrawler stores the structures to disk using Java serialization and exits. From the implementation point of view the outbound link structure is of type `HashMap<String, ArrayList<String>>` with the key as the URL of a page and the value as a list of all the URLs the key points to. The PageRank structure is of type `HashMap<String, Double>` and holds for every page its PageRank value.

### 6.2.2 The URLPartitioner class

The URLPartitioner is given a path to the serialized outbound link structure (of type `HashMap<String, ArrayList<String>>`), a partition size and a path to a root directory. The partition size decides the number of pages one of the parallel AJAX Crawlers have to crawl, for example 20 or 50. The additional root directory path serves as the place where the URLPartitioner stores the partitions. Every partition is made up of a uniquely named directory (under the root directory) which includes a text file with a list of URLs to crawl.

### 6.3 The Crawling Phase

This is where the actual parallelization comes in. The partitions as output from the URLPartitioner are the input of a set of separate AJAX Crawlers. As explained in section 6.1, these AJAX Crawler are completely independent of each other and could be

---

\(^1\)Precrawler of the org.ajax.crawler package

\(^2\)URLPartitioner of the org.ajax.crawler package
run on different machines and, of course, at different times. Only the AJAX Crawlers
must have access to the partitions, something that can be done by sharing them on the
network or by copying them to the corresponding machines.

The controller instance on every machine is a class called MPAjaxCrawler\(^3\) ("Multi
Process AjaxCrawler"). An instance of this class concurrently starts a number of
SimpleAjaxCrawler\(^4\), each in its own Java Virtual Machine process, and attaches each
of this processes to its own URL partition.

As we will show in chapter 7, with this setting of parallelization, crawling performance
is drastically increased compared to the strictly serial version of the AJAX Crawler.

### 6.3.1 The MPAjaxCrawler class

The MPAjaxCrawler class can be configured as follows. The most important constructor
takes two arguments, an int nOfProcLines and an int maxNumOfPartitions.

- **nOfProcLines** This parameter defines the maximum number of concurrent pro-
cesses. This is important to control memory consumption of the AJAX Crawling.
  For example, if we have a machine with 4GB free physical RAM and we are to
  assign a maximum heap size of 500MB to every started JVM, then we could choose
  this parameter to be about \(4000\text{MB}/500\text{MB} = 8\). For all these nOfProcLines
  the MPAjaxCrawler\(_\text{crawl()}\) method starts a separate thread, and each of these
  threads is continuously starting external JVM processes of the SimpleAjaxCrawler
  class (in a serial manner!) until all the partitions are processed. That means, after
  having started an external process, the thread is waiting for this process to exit.
  Then the thread is requesting the next partition ID and starts a new process.
  That’s what we call a "Process Line".

- **maxNumOfPartitions** This parameter defines the number of URL partitions that
  should be processed. The class manages the partition IDs and the getPartitionID()
  method returns the ID of the next partition to process. This method is synchro-
nized for thread safety. If a thread that requests a new partition ID but the number
  of processed partitions exceeds maxNumOfPartitions, the thread will terminate.

Figure 6.2 shows schematically how the MPAjaxCrawler controls the concurrent Simple-
AjaxCrawler processes.

\(^3\)MPAjaxCrawler of the org.ajax.crawler package

\(^4\)SimpleAjaxCrawler of the org.ajax.crawler package
Chapter 6 Parallelization of the AJAX Crawler

Figure 6.2: MPAjaxCrawler parallelization and the concept of Process Lines
6.3.2 The SimpleAjaxCrawler class

The SimpleAjaxCrawler is the building part of the whole crawler and implements the crawling algorithm as described in chapters 3 and 4. Some small modification had to be implemented in order to deal with the consequences that arise from the parallelization setting. For example, the ID of the URL partition to be crawled is given as parameter to the main(String[] argv) method when starting the corresponding JVM process. The ID is determined by the MPAjaxCrawler class as described above. And before the SimpleAjaxCrawler instance starts crawling, it reads the URLs to crawl from the text file in the directory of its partition into an internal data structure.

The extended application model that is generated during the crawling is stored into the directory of the partition using ordinary Java serialization.

6.4 The Indexing Phase

The principal changes according to this parallelized architecture compared to a serial architecture arises from the distribution of the crawling process to several machines. Typically this leads to the application models as well distributed to different machines. The idea to cope with this situation is as follows: The Index, i.e., the Inverted File, is built on every machine for the particular application models residing on that machine.

Let’s look at the build process of one Inverted File on a particular machine. As explained in the preceding sections of this chapter, what we find after crawling has finished is a root directory which includes for every crawled URL partition one subdirectory. This subdirectory contains the application models for all the pages whose URLs are stored in the also contained text file URLsToCrawl.txt. The files containing the application models are the following:

- ajaxapplications.bin
- domelements.bin
- jsvariables.bin
- states.bin
- statetodomelements.bin
- statetojsvariables.bin
The `loadExt()` method of the `DataBase` class reads these files using Java deserialization into memory and afterwards builds the Inverted File. Special care has to be taken to deal with the so called System IDs. Every element has a property called System ID. Similar to a primary key within a relation, they have to be unique within all elements of the same type. But because the different `SimpleAjaxCrawler` processes are held completely independent from each other, every partition uses the same starting points for the System IDs. The `loadExt()` method handles this problem by holding internal variables which store the current value of the System IDs.

### 6.5 The Query Processing Phase

The queries are then sent to every Inverted File on every machine by `Query Shipping`. This is done by a small, central `Search Application`. The results from every index are collected and merged. As we will show in the following subsections, there are consequences that arise from the parallel architecture.

#### 6.5.1 Merge Phase

Because a query is sent to different Inverted Files by `Query Shipping`, we receive a separate result set for every Inverted File we ship the query to. These result sets have to be merged together. Figure 6.3 shows an example: Query $Q$ is sent to $Idx_1$ and $Idx_2$ and the two corresponding result sets $Res_{Idx_1}(Q)$ and $Res_{Idx_2}(Q)$ are returned. Each result in $Res_{Idx_1}(Q)$ is a 3-Tuple $(u, s, r)$, with $u$ as the URL, $s$ as the state and $r$ as the ranking value. $Res_{Idx_1}(Q)$ and $Res_{Idx_2}(Q)$ are then merged together to one result set $Res(Q)$. 

- `transitions.bin`
- `transitiontodomelements.bin`
- `transitiontojsvariables.bin`
6.5 The Query Processing Phase

6.5.2 Ranking

We want to reiterate the fact that the intersection of URLs between some distinct Inverted Lists is empty. Nevertheless, the parallelization architecture has some impact on the Ranking mechanism. It follows a brief discussion on these consequences:

1. PageRank. As indicated above and in Figure 6.1, the PageRank for each page is computed by the PreCrawler. The PageRank structure has to be read into memory before searching. This is accomplished by the DataBase class. For every result, the PageRank value can easily be accessed through this structure (typically in order $O(1)$).

2. AJAXRank. The AJAXRank [20] is a measurement for the ranking order of the states within one AJAX web page. Because of this, the AJAXRanks from one page are independent from the AJAXRanks in other pages. That is why this ranking coefficient can be computed during the index building phase and no special care has to be taken.

3. tf/idf. While the Term frequency $tf$ does not impose a special problem, the $idf$ does. The reason for this is that the $idf$ is a global measure. For convenience we repeat the definition for the $idf$ as we used it in chapter 5:

$$idf(k) = \log \frac{|\{s|s \in D\}|}{|\{s|s \in D \land k \in s\}|} \quad (6.1)$$

The numerator of the formula, $|\{s|s \in D\}|$, denotes the total number of states while the denominator, $|\{s|s \in D \land k \in s\}|$, denotes the number of states that contain keyword $k$. Because an individual Inverted File does not contain this global information, the $idf$ of keyword $k$ cannot be computed during indexing. This problem can be solved by computing the $tf/idf$ value during query processing.
This can be handled as follows: The answers from each Query Shipping return the corresponding numbers, and the \( idf \) is then computed by the query engine. After that step the \( tf/idf \) value can be calculated.

It follows a small example: Consider a query \( q_1 \) composed of only one keyword \( k_1 \). The query is searched for in two different Inverted Files \( Idx_1, Idx_2 \). \( Idx_1 \) contains a total of 10 states while 4 of them contain keyword \( k_1 \), \( Idx_2 \) has a total of 13 states while 6 contain \( k_1 \). The \( idf \) of keyword \( k_1 \) can now be computed as follows:

\[
idf(k_1) = \log \left( \frac{\left| \{s_i | s_i \in Idx_1 \} \right| + \left| \{s_j | s_j \in Idx_2 \} \right|}{\left| \{s_i | s_i \in Idx_1 \land k_1 \in s_i \} \right| + \left| \{s_j | s_j \in Idx_2 \land k_1 \in s_j \} \right|} \right)
\]

\[
= \log \left( \frac{\left| \{s_i | s_i \in Idx_1 \land k_1 \in s_i \} \right| + \left| \{s_j | s_j \in Idx_2 \land k_1 \in s_j \} \right|}{|Idx_1| + |Idx_2|} \right)
\]

\[
= \log \left( \frac{10 + 13}{4 + 6} \right)
\]

\[
= \log \left( \frac{23}{10} \right) = 0.3617
\]

4. **Term Proximity.** Term Proximity is a local measure that involves the keywords that form a query and some state that contains the keywords. Therefore it is not affected by the parallelization of the architecture.

Figure 6.4 shows an example on the ranking problem in the parallelized architecture. It extends the example from subsection 6.5.1 above. All that has changed is that the ranking values of the results are now represented by numbers to show the effect of the changing values. After the result sets \( Res_{Idx_1}(Q) \) and \( Res_{Idx_2}(Q) \) are merged to \( Res(Q) \), the missing \( tf/idf \) part is computed as shown above and has to be added to the current ranking value of each result. This is done in Step 1. For instance the ranking value of the first result changes from 0.8 to 0.9. 0.8 represents the ranking value composed of the term proximity value, the AJAXRank of the according state and the PageRank of the corresponding URL. To compute the missing \( tf/idf \) value we first compute \( idf(k_1) \) (we assume that query \( Q = k_1 \) is a one word query):
6.5 The Query Processing Phase

\[ Res(Q) \quad \rightarrow \quad Res'(Q) \]
\[ \langle url_{2,5}, s_0, 0.8 \rangle \rightarrow \langle url_{2,5}, s_0, 0.9 \rangle \]
\[ \langle url_{1,9}, s_0, 0.2 \rangle \rightarrow \langle url_{1,9}, s_0, 0.5 \rangle \]
\[ \langle url_{2,3}, s_2, 0.3 \rangle \rightarrow \langle url_{2,3}, s_2, 0.4 \rangle \]
\[ \langle url_{1,2}, s_7, 0.1 \rangle \rightarrow \langle url_{1,2}, s_7, 0.3 \rangle \]
\[ \langle url_{1,4}, s_1, 0.3 \rangle \rightarrow \langle url_{1,4}, s_1, 0.6 \rangle \]

Step 1: Add \( tf/idf \) value.

\[ Res'(Q) \quad \rightarrow \quad Res''(Q) \]
\[ \langle url_{2,5}, s_0, 0.9 \rangle \rightarrow \langle url_{2,5}, s_0, 0.9 \rangle \]
\[ \langle url_{1,9}, s_0, 0.5 \rangle \rightarrow \langle url_{1,4}, s_1, 0.6 \rangle \]
\[ \langle url_{2,3}, s_2, 0.4 \rangle \rightarrow \langle url_{1,9}, s_0, 0.5 \rangle \]
\[ \langle url_{1,2}, s_7, 0.3 \rangle \rightarrow \langle url_{2,3}, s_2, 0.4 \rangle \]
\[ \langle url_{1,4}, s_1, 0.6 \rangle \rightarrow \langle url_{1,2}, s_7, 0.3 \rangle \]

Step 2: Sort \( Res'(Q) \).

Figure 6.4: Ranking Phase in the parallelized architecture.

\[ idf(k_1) = \log \left( \frac{|Idx_1| + |Idx_2|}{|\{s_i|s_i \in Idx_1 \land k_1 \in s_i\}| + |\{s_j|s_j \in Idx_2 \land k_1 \in s_j\}|} \right) \]
\[ = \log \left( \frac{|Idx_1| + |Idx_2|}{|Res_{Idx_1}| + |Res_{Idx_2}|} \right) \]
\[ = \log \left( \frac{|Idx_1| + |Idx_2|}{|Res|} \right) \]

With the \( idf(k_1) \) computed, the weighted \( tf/idf \) value can then be evaluated as

\[ w_3 \cdot tf(k_1, s) \cdot idf(k_1) \] (6.2)

where \( w_3 \) is the weight according to formula 5.3. We assume that this expression evaluates to 0.1 for the first result in \( Res(Q) \) and therefore the new ranking value is \( 0.8 + w_3 \cdot tf(k_1, s) \cdot idf(k_1) = 0.8 + 0.1 = 0.9 \). The ranking values of the other results change accordingly. Because all the results have their ranking value changed, \( Res(Q) \) then has to be sorted by rank. This can be seen in Step 2.
6.6 Result Aggregation

Parallelization also induce some slight changes what state reconstruction is concerned. In principle, the same algorithm that is presented in Section 5.4 can be used, but we have to consider that the page models are distributed on different machines. Thus, in order to present the user the state which contains the query, the algorithm slightly changes:

1. Determine the page model (the machine) the result originally comes from.
2. Extract from that page model the path from the initial state to the desired state.
3. Construct the DOM of the initial state.
4. Invoke all annotated events to the desired state and construct the DOM of the generated state.
5. Present the generated DOM in a browser.

This chapter presented a parallelized architecture of the AJAX Crawler. Two main performance bottlenecks of the ordinary AJAX Crawler have been identified. The first bottleneck caused by the high degree of network communication and the resulting waiting time is addressed by running more instances of AJAX Crawlers concurrently. The second bottleneck caused by the increasing effort to maintain a growing application model is addressed by dividing the application model into several smaller partitions. One instance of an AJAX Crawler then only works on one such partition. As we will show in chapter 7, this can improve its performance considerably. We also showed how Inverted Files can be built from several partitions. Then we discussed the consequences for Query Processing and Result Aggregation when applied to several Inverted Files maybe spread on different machines.
Chapter 7

Experimental Results

This thesis 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 includes AJAX content. We crawled the AJAX part, and compared AJAX crawling and search with traditional crawling, which ignores this content. The goals of the experiments were:

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 trade-off between gain in search result and performance decrease.

7.1 Experimental setup

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

7.1.1 YouTube Datasets.

The experiments were performed on YouTube [38], 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 1 and continuing on the related videos until the desired number has been reached. Each page and its comments have been crawled by invoking AJAX events. For performance reasons, we restricted the number of additional
comment pages that we retrieved (excluding the initial) to ten (e.g., eleven comment pages per video). The characteristics of the data are displayed in Table 7.1. The total number of events triggered during crawling is 187980. Without any intervention, every triggered event would result in a network call. However, the real number of events leading to network calls to the server is only 37349. The reason for this 80% decrease is the hot node policy.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Pages</td>
<td>10000</td>
</tr>
<tr>
<td>Total Number of States</td>
<td>41572</td>
</tr>
<tr>
<td>Total Number of Events</td>
<td>187980</td>
</tr>
<tr>
<td>Avg. Number of Events per Page</td>
<td>18.798</td>
</tr>
<tr>
<td>Number of Events leading to Network Communication</td>
<td>37349</td>
</tr>
</tbody>
</table>

Table 7.1: Statistics of the YouTube10000 data set.

In order to showcase the impact of caching on the crawler, we have used a particular subset of YouTube10000, which contain the first 20, 40, 60, 80 and 100, 250 and 500 videos crawled in a breadth-first fashion in YouTube10000.

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.

Figure 7.1 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 7.2 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. 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 1.1. 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.
7.1 Experimental setup

Figure 7.1: Distribution of YouTube videos based on number of comment pages.

Figure 7.2: Number of states and events vs. number of crawled videos in a subset of YouTube10000.
Chapter 7 Experimental Results

7.1.2 Algorithms

In order to evaluate the impact of AJAX Search, two flavors of crawling have been applied.

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 comments 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 Crawling.** This is the AJAX crawler which reads the AJAX content of a Web Page, and uses the algorithms proposed in chapters 3 and 4. In case of YouTube, this means the full-fledged AJAX Search engine, with full capabilities of client-side code and triggers events from the page in order to build the AJAX Page Model of chapter 2. The crawler was run in both sequential (Section 5.1) and parallel mode (Section 6.1) as presented below. When search capabilities are used, we implemented the Indexing and Query Processing capabilities of chapter 5. The Crawling process(es) stored the AJAX model on disk. The Index was constructed incrementally [13] 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 harddisk, 1 x 500 GB S-ATA 7200 rpm harddisk running Windows 2000 Server. More capabilities of the framework, which go beyond the scope of this thesis, are described in [18]. Javascript code was analyzed using the [16] toolkit.

7.2 Crawling Performance

This section addresses the overhead induced by crawling AJAX content in YouTube, as opposed to just traditional crawling.

7.2.1 Overhead of AJAX Crawling

Table 7.2 shows the crawling times needed for the YouTube subset of 10000 pages. 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 crawling, all Javascript
code is enabled, the first page of comments is read, and the next and prev events are invoked in order to load the additional comment pages from the server, using AJAX. All following numbers use the heuristic crawling algorithm which caches the results of hot node invocations.

<table>
<thead>
<tr>
<th></th>
<th>Trad. (ms)</th>
<th>AJAX (ms)</th>
<th>AJAX Crawl./ Trad. Crawl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>17465598</td>
<td>164751693</td>
<td>x9.43</td>
</tr>
<tr>
<td>Mean per page</td>
<td>1746.56</td>
<td>16475.17</td>
<td>x9.43</td>
</tr>
<tr>
<td>Mean per state</td>
<td>1746.56</td>
<td>3963.045</td>
<td>x2.27</td>
</tr>
</tbody>
</table>

Table 7.2: Crawling Times (ms) and Overhead of AJAX Crawling (ms).

It can be seen that the overhead factor of AJAX is about 9.43 times in total. However, the overhead between crawling a single state and a traditional page is 2.27. This is realistic and it is due, first, to the Javascript processing overhead and, second, the incremental maintenance of the model during crawling.

### 7.2.2 Distribution of Crawling Times

Table 7.2 showed that crawling time in case of AJAX page is generally significantly greater than that of traditional pages. However, as displayed in Figure 7.3 the distribution of pages per crawling time ranges shows that most of the pages take less than five seconds to be crawled. Only pages with a larger number of states take times larger than 20 or 30 seconds. This shows that it is feasible for AJAX crawling to be implemented by commercial search engines, in case of sites with a lower number of AJAX states. Another option is that of a focused AJAX crawling, which just performs crawling on content relevant to a more narrow range of users, which is both useful and restricts the number of AJAX states.

### 7.2.3 Influence of Number of States

It is interesting to notice the variation in crawling performance per video, according to the number of crawled states. Figure 7.4 shows that crawling time increases linearly with the number of crawled states. The lower curve shows the crawling time after the network time has been deducted. The main crawling bottleneck is maintaining and updating the page model of a fairly complex YouTube HTML site. An optimization left as a future
work is the differential update of the index, especially in case of YouTube, where just a small part of the page (i.e., the comment page) changes while most of the page remains intact.

### 7.3 Effects of Caching

We implemented both a non-caching and caching version of the AJAX Crawl engine, conforming to chapter 4. The role of Caching and the heuristic policy is to reduce the number of AJAX calls to the server, and allow early detection of duplicate states.
7.3 Effects of Caching

without performing a network call. For this, we crawled a subset of 10, 20, 40, 60, 80
and 100 YouTube videos and studied the effect of caching on the Crawling Performance
and Throughput.

Hot cache is useful in YouTube since each comment page has invocations to four next
pages which may result in the same call (e.g., in page 1, a click on “next page” results in
the same network call as “jump to page 2”, and the same two events still appear in page
2 and 3). When hot node detection is used, a cache is built, which stores any content
fetched from the server. YouTube comment pages have a single hot node (i.e., function)
accessed by all events controlling the comment pages. Invoking the function a second
time is avoided, and the previously stored content is used.

The heuristic policy showed its advantages. Figure 7.5 displays the number of AJAX
events from the crawled pages for which a network call was needed.

In the case without hot node detection crawling, all events result in a network call. However, in case of the caching version the number of network calls decreases significantly
until it reaches 359, as opposed to the total of 1790 for the non-caching approach, which
is a factor of five. This directly influences the time spent on the network time, as
shown in Figure 7.6, which is reduced by a factor of 0.37 and the overall throughput of
states crawled per second is also improved with a factor of 1.6 (as shown in Figure 7.7).
Chapter 7 Experimental Results

Figure 7.5: Number of AJAX Events resulting in AJAX Calls with and without caching policy.

The same factor can be noticed in terms of page and event throughput, but it is not mentioned here. For applications with more than one hot node, we expect even better improvement in performance.

In terms of crawling time, the total time needed to crawl 100 videos was 1292 seconds in the non-caching version, as opposed to 831.7 seconds in the caching version. This difference made it for us impossible to crawl the whole 10000-video dataset without applying the cache policy. However, we mention that for the entire YouTube10000, the total number of events is 187980 and caching reduces network calls to 37349 (i.e., still a factor of five).

7.4 Parallelization

As mentioned in chapter 6, in order to counteract the fact that caching cannot be fully performed on AJAX Web sites, we applied the parallelization technique, and assigned different URL of videos to four different processes running on the same machine. This
7.4 Parallelization

Figure 7.6: Network time with and without hotnode policy.

Figure 7.7: State throughput per second with and without hotnode policy.
can be easily done since the crawling processes of two different videos are completely independent.

### 7.4.1 Performance of Parallelized Crawling.

We executed a Traditional and Enhanced Crawling, as shown in Table 7.3 in case of parallelization.

<table>
<thead>
<tr>
<th></th>
<th>Parallel Trad.(sec)</th>
<th>Parallel AJAX (sec)</th>
<th>Parallel AJAX./ Parallel Trad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>4800</td>
<td>42180</td>
<td>x8.80</td>
</tr>
<tr>
<td>Mean per page</td>
<td>0.480</td>
<td>4.218</td>
<td>x8.80</td>
</tr>
<tr>
<td>Mean per state</td>
<td>0.480</td>
<td>1.015</td>
<td>x2.11</td>
</tr>
</tbody>
</table>

Table 7.3: Parallel Crawling Times (sec) for Traditional and AJAX Crawling.

The Table presents the total crawling time, mean crawling time per page (video), and mean crawling time per state in case of parallel crawling. In case of parallel crawling, the overhead of crawling an AJAX page with comments over crawling it without comments, is of a factor of x8.80. However, it is worth mentioning that a single state is crawled at about 2.11 the time of a traditional page, which means the overhead is comparable. The overhead is still caused by the additional Javascript manipulation and invocation in AJAX Crawling.

### 7.4.2 Parallel vs. Non-parallel.

We compared the gain in crawling time in case of parallel crawling, over traditional crawling, as shown in Figure 7.8. The crawling times in case of parallelization are 27.5% lower than non-parallelized case, for traditional crawling, and 25.6% lower in case of AJAX crawling. Since we used four parallel processes, this is also consistent with the degree of parallelizaton.
7.5 Query Processing

One goal of this thesis was also to motivate the time lost in crawling by showing the quality of results. We built the Index as shown in Section 5.2 and we performed queries on it, as in Section 5.3.

7.5.1 Queries

The queries we performed are taken from the most popular YouTube queries, taken from [39]. A snippet of these queries is shown in Table 7.4. There are 100 queries in total. The Table shows a sample of 9 queries in order of cardinality. We specify the number

Figure 7.8: Effect of Parallelization on Mean Crawling Time Per Video.
Chapter 7 Experimental Results

of videos where the query appear in the first comment page, and the total number of comments where the keyword appears\(^1\).

<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>wow</td>
<td>310</td>
<td>2041</td>
</tr>
<tr>
<td>Q2</td>
<td>dance</td>
<td>274</td>
<td>1665</td>
</tr>
<tr>
<td>Q3</td>
<td>funny</td>
<td>233</td>
<td>1624</td>
</tr>
<tr>
<td>Q4</td>
<td>our song</td>
<td>104</td>
<td>758</td>
</tr>
<tr>
<td>Q5</td>
<td>sexy can i</td>
<td>99</td>
<td>512</td>
</tr>
<tr>
<td>Q6</td>
<td>american idol</td>
<td>93</td>
<td>865</td>
</tr>
<tr>
<td>Q7</td>
<td>kiss</td>
<td>88</td>
<td>603</td>
</tr>
<tr>
<td>Q8</td>
<td>fight</td>
<td>82</td>
<td>541</td>
</tr>
<tr>
<td>Q9</td>
<td>no air</td>
<td>73</td>
<td>503</td>
</tr>
<tr>
<td>Q10</td>
<td>chris brown</td>
<td>51</td>
<td>478</td>
</tr>
<tr>
<td>Q11</td>
<td>low</td>
<td>38</td>
<td>348</td>
</tr>
</tbody>
</table>

Table 7.4: YouTube Queries.

7.5.2 Query Processing Time

We ran the queries on the 2500 page-index. The query processing times for the individual queries of Table 7.4 on the index containing 2500 pages are shown in Table 7.5.

<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>wow</td>
<td>13.75</td>
<td>229.68</td>
</tr>
<tr>
<td>Q2</td>
<td>dance</td>
<td>19.85</td>
<td>703.12</td>
</tr>
<tr>
<td>Q3</td>
<td>funny</td>
<td>12.82</td>
<td>433.11</td>
</tr>
<tr>
<td>Q4</td>
<td>our song</td>
<td>109.53</td>
<td>2505.26</td>
</tr>
<tr>
<td>Q5</td>
<td>sexy can i</td>
<td>16.72</td>
<td>349.68</td>
</tr>
<tr>
<td>Q6</td>
<td>american idol</td>
<td>22.19</td>
<td>1748.09</td>
</tr>
<tr>
<td>Q7</td>
<td>kiss</td>
<td>0.78</td>
<td>32.66</td>
</tr>
<tr>
<td>Q8</td>
<td>fight</td>
<td>0.62</td>
<td>19.53</td>
</tr>
<tr>
<td>Q9</td>
<td>no air</td>
<td>33.28</td>
<td>1032.63</td>
</tr>
<tr>
<td>Q10</td>
<td>chris brown</td>
<td>4.53</td>
<td>321.40</td>
</tr>
<tr>
<td>Q11</td>
<td>low</td>
<td>0.31</td>
<td>19.84</td>
</tr>
</tbody>
</table>

Table 7.5: Query Times on YouTube.

\(^1\)for the numbers we used a subset of the first 2500 videos
The query processing times in case of AJAX Search are obviously larger than in the case of traditional search. The throughput varies however much for individual queries, as shown in Figure 7.9, but generally traditional search offers better throughput, although for a much smaller number of results.

![Throughput Trad. vs. AJAX for most popular YouTube queries](image)

Figure 7.9: Throughput of popular YouTube queries in Traditional and AJAX Search.

### 7.6 Setting Crawling Threshold

Since query processing time decreases, it is useful to be able to set a crawling limit until the overhead is bearable. One way is to use the total result throughput related to the number of crawled states. Figure 7.10 shows how the number of crawled states (and therefore, amount of indexed AJAX Content) influences throughput. For increasing number of states, the relative result throughput of AJAX vs. Traditional decreases significantly. This result is a hint on where implementers of an AJAX crawl engine can decide the best tradeoff between the amount of AJAX content that is indexed and the query performance. If the limit is 0.4, the number of states to crawl can be set to a limit of 5.
Chapter 7 Experimental Results

7.7 Improvement in Search Quality

We conclude the evaluation by showing that our goal is fulfilled: return better search results to the user. We evaluate the quality of results as follows: we construct eleven different 2500-page-indexes. The first includes only the first (initial) state of the application model and represents the traditional index. The second includes two states and represents an enhanced index with one additional state. The third includes the first three states and so on. Then we run the 100 queries in the query set over all these indexes and retrieve the results. We consider all results returned by the AJAX version as correct and we assume boolean retrieval. We compute the relative Recall as the number of results from traditional search divided by the number of results of enhanced search. For example if $R_1(q_i)$ denotes the result set of query $q_i$ processed on the index with 1 state and $R_5(q_i)$ denotes the result set of query $q_i$ processed on the index with 5 states, then

$$\text{RelRecall}_{1,5}^{q_i} = \frac{|R_1(q_i)|}{|R_5(q_i)|}$$

(7.1)

stands for the relative Recall between traditional and enhanced with 5 states with respect to query $q_i$. Figure 7.11 shows that the $1 - \text{RelRecall}$ measure increases with the number of states. But the gradient decreases with the number of states. Qualitatively, this indicates that the advantage you get by crawling an additional state decreases with the number of states.
Therefore, as with the crawling threshold setting in section 7.6, we could also set a threshold to limit the number of states to crawl. For example, if we set the threshold of $1 - \text{RelRecall}$ to 0.7, then only 4 states have to be crawled.

![Graph showing 1 - RelRecall of Traditional Search/AJAX Search.](image)

Figure 7.11: $1 - \text{RelRecall}$ of Traditional Search/AJAX Search.
Chapter 8

Infrastructural Setup

In this chapter we briefly explain how to configure and work with the AJAX Crawler. It should serve as a starting point for people who engage themselves in further development of the AJAX Crawler.

Most global configuration settings are stored in a class called AJAXConfig. The variables are defined as public static, so they can be used by several different classes.

8.1 Precrawling Phase

As described in chapter 6, the Precrawling phase is responsible for the creation of the hyperlink graph as well as the computation of the PageRank values (done by the Precrawler class). Also, as last step of this phase, the list of URLs have to be partitioned (done by the URLPartitioner class). The URLPartitioner class is dependent on the results of the Precrawler class.

Neither of the two classes provides a Graphical User Interface. They can be started in a shell with the java command or within a development environment.

8.1.1 [org.ajax.crawler.]Precrawler

There are four important variables set in the AJAXConfig class:

1. PRECRAWLER_ROOT_DIR of type String. This is the (absolute) path to the directory where the Precrawler class stores its output data, i.e, the hyperlink graph structure, the PageRank structure and a log file.
Chapter 8 Infrastructural Setup

2. PRECRAWLER_START_URI_ID of type String. This represents the starting page for the Precrawler.

3. PAGERANK_FILE_NAME of type String. This defines the name of the PageRank structure.

4. NUM_OF_PAGES_TO_PRECRAWL of type int. This sets the limit on how many pages that should be contained in the hyperlink graph structure.

The Precrawler class contains a main() method and can therefore be started on its own. It builds the hyperlink graph, computes the corresponding PageRank values and stores the results in the root directory as defined above.

8.1.2 [org.ajax.crawler.]URLPartitioner

Two variables have to be set in the AJAXConfig class:

1. PARTITION_SIZE of type int. This is the number of web pages one SimpleAjaxCrawler has to crawl. Obviously, this number has impact on the memory consumption of a SimpleAjaxCrawler instance.

2. URI_PART_FILE_NAME of type String. This represents the name of the file which contains the URLs.

Figure 8.1: An example subdirectory structure, created by the URLPartitioner

When started, the URLPartitioner reads the PageRank structure in the root directory of the precrawler and produces web page partitions of the defined size. For every partition a subdirectory under the root directory of the crawler (see below in section 8.2) is created. In every of these subdirectories, a text file with a list of URLs is stored.
8.1 Precrawling Phase

For example, if we precrawl 107 pages and define a partition size of 20, then 6 directories are created. Every of these directories contains a file named as defined in variable URI_PART_FILE_NAME, e.g., URLsToCrawl.txt. The names of the directories are numbers starting from 1. The directory structure of this example is showed in Figure 8.1.

As an example on how to run the program, assume you want the Precrawler to build the web graph and the PageRank structure for 10000 YouTube videos. The starting video shall be http://www.youtube.com/watch?v=w16JlLSySWQ. Then the following parameters in the AJAXConfig should be set accordingly:

```
PRECRAWLER_ROOT_DIR = "/path/to/precrawler/rootdir";
PRECRAWLER_START_URI_ID = "w16JlLSySWQ";
PAGERANK_FILE_NAME = "PageRank.txt"
NUM_OF_PAGES_TO_PRECRAWL = 10000;
```

After these values set the Precrawler class can be started within Eclipse or using a command shell. If the Precrawler is started on a command shell, then open a shell, navigate into the bin folder of the project, e.g., /home/matterr/projects/ajaxsearch/ajax/bin and type:

```
[matterr@fifthelement bin]$ java org.ajax.crawler.Precrawler
```

When the program has finished you find the outbound link structure along the PageRank structure in the directory /path/to/precrawler/rootdir/ (the one that was defined as precrawler root directory). To build the directory structure for the Crawling Phase we have to run URLPartitioner. Assume we want a partition size of 50 and the corresponding directory structure in /path/to/rootdir, then following parameters in the AJAXConfig should be set as follows:

```
PARTITION_SIZE = 50;
URI_PART_FILE_NAME = "URIsToCrawl.txt"
YOUTUBE_CRAWLDATA_ROOT_DIR = "/path/to/rootdir"
```

Again, URLPartitioner can directly be started within Eclipse or using a command shell analogous to above with:
Chapter 8 Infrastructural Setup

[matterr@fifthelement bin]$ java org.ajax.crawler.URLPartitioner

After URLPartitioner has finished, the \(1000/50 = 200\) 50-page-partition-subdirectories can be found in /path/to/rootdir.

8.2 Crawling Phase

The most complex classes for the Crawling Phase – apart from the AJAXConfig class – are MPAjaxCrawler and SimpleAjaxCrawler class. The following variables defined in the AJAXConfig class can be configured:

1. **MP_CRAWLER_CLASSPATH** of type String. This is used as value for the `-classpath` argument for the `java` command when MPAjaxCrawler starts a SimpleAjaxCrawler.

2. **MP_CRAWLER_JVM_XMS** of type String. This variable defines the value of the `-Xms` argument and is also used for the `java` command.

3. **MP_CRAWLER_JVM_XMX** of type String. Dito, but now the `-Xmx` argument. This variable sets the maximum heap space demanded by the Java Virtual Machine per SimpleAjaxCrawler. For example, if you have 1 GB of available RAM on a machine and you intend to crawl using 2 Process Lines, you can roughly divide 1 GB by 2 to get a reasonable number for this variable.

4. **MP_CRAWLER_START_PARTITION** of type int. This variable denotes the first partition to crawl.

5. **MP_CRAWLER_END_PARTITION** of type int. This variable denotes the last partition to crawl.

6. **MP_CRAWLER_NUM_OF_PROC_LINES** of type int. This variable denotes the maximum number of concurrently working SimpleAjaxCrawler processes called Process Lines. The concept of Process Lines is introduced in chapter 6. This number set to 1 means no parallelization at all.

7. **SACR_NUM_OF_ADDITIONAL_STATES** of type int. This variable defines the maximum number of additional states to be crawled, not counting the initial one. Intuitively, the higher this number is, the higher gets the memory consumption of an individual SimpleAjaxCrawler. This has to be taken into account when choosing the values for **MP_CRAWLER_NUM_OF_PROC_LINES** and **MP_CRAWLER_JVM_XMX**.
8. **USE_DEBUGGER** of type boolean. This variable turns the Hot Node mechanism on (true) or off (false).

9. **IS_YOUTUBE** of type boolean. This variable lets some classes know that the pages to crawl are YouTube pages.

10. **TRADITIONAL_CRAWLING** of type boolean. When set to false, AJAX Crawling is turned off. This lets the crawler behave like a traditional crawler. Javascript is disabled, i.e, no events are triggered, not even the onload event of the body tag.

As with the classes from the Precrawling phase, the **MPAjaxCrawler** does not provide a Graphical User Interface yet. It can be started in a shell using the java command or within a development environment, e.g., Eclipse. If started using the java command, the -classpath switch for several project specific .jar files (like COBRA, Rhino, and so on) has to be declared. A valid -classpath argument can be found in the variable **MP_CRAWLER_CLASSPATH** of the **AJAXConfig** class.

It follows an example that extends the one from section 8.1.2: Consider a crawling session of 10000 videos on one machine with 4 GB of available RAM. The number of additional comment pages should be 10. We assume here that the Precrawling Phase with a partition size of 50 has already finished and the root directory is /path/to/rootdir/. We assume further that crawling should be done using the hot node policy. How many Process Lines are allowed? According to the requirements (number of additional states is 10 and the partition size is 50), each **SimpleAjaxCrawler** instance demands approximately 500 MB. This said we can now decide on the allowed number of Process Lines: 4GB/500MB ≈ 8, so a maximum of 8 Process Lines is possible. The according variables in the **AJAXConfig** class would be set as follows:

```java
YOUTUBE_CRAWL DATA_ROOT_DIR = "/path/to/rootdir/";
MP_CRAWLER_JVM_XMX = "-Xmx512m";
MP_CRAWLER_START_PARTITION = 1;
MP_CRAWLER_END_PARTITION = 200;
MP_CRAWLER_NUM_OF_PROC_LINES = 8;
SACR_NUM_OF_ADDITIONAL_STATES = 10;
IS_YOUTUBE = true;
USE_DEBUGGER = true;
TRADITIONAL_CRAWLING = false;
```
8.3 Indexing and Query Processing

We developed a simple GUI application for the reason of administrating Application Models and Indexes as well as accomplishing user-defined queries. The corresponding class is called AJAXSearchSetupApp and belongs to the org.ajax.setup package.

The application provides functionality for:

- Building an Index from Application Models.
- Saving an Index to disk for later use.
- Reading an Index from disk.
- Manually process user-defined queries.

After starting AJAXSearchSetupApp one can choose between three tabs.

1. **Build New Index.** This tab helps building indexes from application models on disk.

2. **Load Existing Index.** Here one can load stored index files from disk into memory.

3. **Search.** This tab serves as a simple search facility to test the built or loaded index.

The following subsections focus on the handling of these three tabs.

### 8.3.1 Building a New Index

Figure 8.2 shows the application for building an index (Inverted File) from stored application models.

By clicking the [Choose Dir] button the user can pick a root directory from which she wants to build an index. This directory should be of the structure as discussed in section 8.1.2. In the text fields [Starting Partition] and [Ending Partition] one can define which URL partitions are being used to build the index. In [Max. State ID] the user can define how many states should be included in the index. For example, if one chooses 1 as Maximum State ID, only the initial state is used for building the index (although more states had been crawled and stored in the application model).

Then the [Build Index] button is clicked to build the index. Depending on the values set and on how big the application models are this can take some time.
8.3 Indexing and Query Processing

Figure 8.2: The UI for building an Index from stored Application Models.

If the user wants to store the index on disk for later use, she can do so by clicking the ⟨Save Index⟩ button.

8.3.2 Loading an Index

The application also lets the user load an Index from disk, without going through the process of building the index from the application models. To do so one changes to the ⟨Load Existing Index⟩ tab and selects the Inverted File by clicking the ⟨Load Index⟩ button. If the user also wants to load a certain PageRank structure that is different from the one specified in the AJAXConfig class, he can do so by clicking the ⟨Load PageRank⟩ and choosing the corresponding file. Figure 8.3 shows the GUI for loading an Index.

8.3.3 Query Processing

To search in an Index, for example for testing purposes, the application also provides this functionality. Any user-defined query can be typed in the ⟨Query⟩ text field on the ⟨Search⟩ tab. By clicking the ⟨Search⟩ button the query is processed and the results
Chapter 8 Infrastructural Setup

Figure 8.3: The UI for loading an Inverted File into memory.

are listed in the text area below the query field. Figure 8.4 for example shows the results for the query "morcheeba enjoy the ride tzuke judie skye" in a sample index.

Figure 8.4: The UI for testing the Query Processing.
Chapter 9

Related Work

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

**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 avoid repeated access to the same URL. For the semantics duplicates, several algorithms exist to detect near-duplicate web pages: [26], [29], Broder et al.’s shingling algorithm[12] and Charikar’s [14], [29] random projection. DustBuster (Different URLs with Similar Text) [5] tries to detect duplicate URLs using mining. This is not possible for AJAX application crawling. Traditional Web Crawling has been addressed in works such as [11]. [32] crawls YouTube but for Data Mining purposes.

**Javascript Analysis.** The work of [28] 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 such as [15] works deal with detecting common spam redirects on web pages.

**Application Models.** Colorful XML [27] 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 [30], [36], [34], [35]. 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.

**Parallelization/Optimization.** UbiCrawler [6], [7] and [8] is parallelized using autonomous agents for scalability as well as fault-tolerance. Although its architecture is more sophisticated than ours, they share several commonalities. We didn’t particularly focus on fault-tolerance. Also, UbiCrawler doesn’t need a Precrawling phase, because the web-graph is computed during crawling. However, our goal was to avoid any communication between different crawling processes. They also propose techniques to compress the web-graph [9]. We did not consider this kind of optimization, because we focused on the minimization of network calls (Hot Node detection mechanism).

Other work in the database community applies: ranking models such as XRank [25], which itself extends PageRank [10], have been used for generic XML structures. In the Annotated Application Structure we propose, we adapted such techniques and used the interactions between content and states for ranking. Techniques such as [3] can be used to rank based on the lowest common ancestor. TopXSearch[33] results or Threshold Algorithms [19] can be applied for an optimized computation of results and ranking. Incremental Indexing [13] is also relevant.
Chapter 10

Conclusions and Future Work

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. 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.
Chapter 10 Conclusions and Future Work
Chapter 11

Bibliography


