Indexing AJAX Web Applications

Master Thesis

Gianni Frey

30.11.2007
# Contents

1 Abstract 5

2 Goal 7
   2.1 Motivating Example: News page ................................. 7
   2.2 The Problem of Impedance Mismatch ............................ 8
   2.3 Indexing AJAX Applications ..................................... 9
   2.4 State-of-the-Art ................................................... 9
   2.5 Proposed Approach ............................................... 10

3 Crawling 11
   3.1 Application Structure ............................................. 11
   3.2 Application Model .................................................. 12
   3.3 Relational Structure ............................................... 13
   3.4 Optimizing Crawling ............................................... 13
   3.5 Implementation .................................................... 14
      3.5.1 Class Structure ............................................. 14
   3.6 Executing all Javascript Events ................................. 15
   3.7 Comparing States .................................................. 16
      3.7.1 Fixes to Cobra and Rhino ................................. 16
   3.8 Store Implementation ............................................. 20

4 Indexing 23
   4.1 Index Construction .................................................. 23
   4.2 Keyword Tokenizer .................................................. 24
   4.3 Keyword Rewriting .................................................. 25
   4.4 Correlations .......................................................... 25
      4.4.1 Correlations in AJAXSearch ............................ 26
   4.5 Optimizing Indexing ............................................... 27

5 Query Processing 29
   5.1 Processing Simple Keyword Queries .............................. 29
   5.2 Processing Conjunctions .......................................... 29
   5.3 Implementation .................................................... 30

6 AJAXRank 33
   6.1 Ranking Requirements .............................................. 33
   6.2 Ranking Application States ....................................... 34
      6.2.1 AJAXRank vs. PageRank ................................. 35
      6.2.2 Influence of the Click-Distance ........................ 36
Chapter 1

Abstract

Current search engines such as Google and Yahoo! are prevalent for searching the web. Search on dynamic web pages, however, is either inexistent or far from perfect. This is a real impediment since client-side web applications are already part of the web and AJAX and Rich Internet Applications are the applications of the future, on the web and on mobile devices. Current search engines either ignore AJAX applications, or produce false positives and false negatives. The reason is that the application logic is embedded in a single web page, identified by a single URL. The application itself, however, is composed of states which can be seen by the user, but not by the search engine, and changed using by the user using events. This thesis sets the stage for this new search challenge and proposes a solution: it shows how an AJAX web application can be indexed and searched automatically in the granularity of the application states. The application states are automatically extracted, a new ranking model based on application states and transitions is proposed and a practical implementation on an actual AJAX application is presented.
Chapter 2

Goal

Google Mail[11], Yahoo! Mail[24], Google Maps[12] are well-known AJAX applications. Their goal is to enhance the user experience by running client code in the browser instead of always refreshing the web page, and minimizing server traffic and server load. A dynamic web application at a given URL is usually based on Javascript and 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. This approach of the state-of-the-art is however insufficiently precise, as shown in the following example:

2.1 Motivating Example: News page

Figure 2.1: An AJAX Application is a sequence of states under the same URL.

The following figure shows a simple news web application. The application displays a piece of news at a time. It allows the user to change the current piece of news by pressing a button. A search engine wants to index this page in order to perform keyword search on it. Three problems may occur:

- (Dynamic) Content ignored: The application loads the news into a script variable and updates the page using client-side methods executed after the page is loaded, such as `document.write(newsText)`. Current search engines do not see the dynamic code part and do not index it at all. This is an obvious case of False Negatives produced by current search engines (Figure 2.2).

- Content-as-whole: the application is a pure client-side application (so no AJAX communication with the server). In this case, all relevant news are already copied directly in the body of page when the URL of the application is accessed. Layers
Figure 2.2: False negative in Google, the search for *ross perot* does not return a result

can also be used to just display the relevant piece of news, although they are all in the body of the page. Current search engines will see all the news, although the interface displays just one, since it does not understand the user view given by the client-side code. This is a clear case of False Positives, since two pieces of news are indexed together although they are unrelated. A similar case of False Positives can occur if the search engine only indexes initial text such as loading information (Figure 2.3)

Figure 2.3: False positive in Google, the search for *loading data* should not return a result

• *Content-at-a-time*: the application is an AJAX application which also communicates with the server. At load time, the first piece of news is loaded on the page. This is what a search engine sees and indexes: the first piece of news. Afterwards, based on user actions, the other pieces of news are dynamically loaded and displayed on the page. A traditional search engine indexes just one line of news and the other lines are not indexed. This is a clear case of False Negatives.

## 2.2 The Problem of Impedance Mismatch

Generically, the problem can be presented in the following broader context: there is an impedance mismatch between the user view (seen through the application) and that of the search engine (which accessed the data directly). This situation is illustrated in Figure 2.4
and has been previously addressed in our previous work in the case of desktop search [7] and Enterprise Application Search[9].

![Diagram showing User View differs from the Search Engine View.](image)

Figure 2.4: The User View differs from the Search Engine View.

In traditional web and desktop search, a result is a link. The link is either to a document in desktop search or to a web page in traditional web search. In case of Enterprise Web Application Search in general, and in particular in case of AJAX Applications, the result is a link (a URI) but which should also contain the information for recreating the function calls (events), objects (DOM objects), parameters, and even state. In our approach we define a special URL which encodes enough information to reconstitute the application state in order to show to the user, as if it the user itself had reached that particular state in the AJAX application.

### 2.3 Indexing AJAX Applications

We summarize the particularities of Indexing AJAX Applications:

- **The actors are the events.** Not individual pages and their URLs need to be indexed, but states generated by user events.

- **An application model is needed.** The model of a single page is not granular enough for correct search. A special model needs to be built.

- **A new ranking model is needed.** Different parts of the page react differently to events and contain more or less important parts of the application. Page hierarchy enhances or reduces the ranking of individual parts. A ranking function must be defined to reflect these particularities.

- **“Links” must be redefined.** Not only a URL, but also function calls, objects, parameters and state must be part of a link to reconstruct the result.

### 2.4 State-of-the-Art

Current search engines such as Google do try to solve the problem of indexing AJAX applications in two ways. The first approach is to index the entire content of the page without running client-side code. This causes either false positives and false negatives as shown before. The other approach is to expose the data to the search engine directly (in an agreement with the application provider). This increases precision since some results are returned, however, the granularity might be too coarse and is hard coded. The third, obvious approach, is to hard-code the search functionality in each application. This is time consuming and something which small application providers cannot afford. We provide an automatic approach.
2.5 Proposed Approach

We take the approach of indexing the application in the granularity of application states and we aim to automate the process. We identify the states using generated URLs which can be even used to reconstruct the indexed application state. The enhanced architecture of an indexer and crawler for dynamic web applications is presented in Figure 2.5:

![Figure 2.5: Architecture of an AJAX Search Engine.](image)

We identified the following phases of an AJAX Search Engine:

**Crawling.** A special, Javascript-aware Crawler constructs the application model. Based on the events on the page, the transition graph containing states, events and transitions is built. This is a contribution over traditional search.

**Indexing.** Indexing works on the application model. The result is a special enhanced index, an extension to traditional inverted files. The index contains links to the actual application state which contains a given keywords.

**Query Processing.** Keyword Search can be performed on the index. The results are links to application states (URI, state, transition and parameters). An important part of the process is ranking the result. A new ranking scheme based on the application model needs to be defined (AJAXRank), including application states and the relative importance of the page part containing the result.

**Result Aggregation.** Results must be presented to the user. Either the initial URI (as in traditional search) or the actual application state (in our enhanced version) must be reconstituted and returned. This phase in therefore complex in the enhanced search and must be defined. The result aggregation phase generalizes the implementation of this step in order to make it compatible with both traditional and enhanced search on AJAX application.

The rest of the report is organized as follows: Section 3 describes the crawling process and the application model with states, transitions and events. Section 4 describes the Indexing process while Section 5 describes Query Processing for AJAX applications. Section 6 is dedicated to the AJAX Rank. Result Aggregation is presented in Section 7 and the entire framework is applied on a custom-built AJAX Web Application in Section 8. Results show that AJAX Indexing obtains better quality results than traditional search engines which do not understand the application logic embedded in the client. Finally, Section 9 describes the tools used to build the AJAX search engine.
Chapter 3

Crawling

Crawling is an operation which is able to read a dynamic web application and build its application model, in order to enable the application for search. As mentioned before, this is opposed to traditional search, where crawling just indexes a the content of a simple web page.

3.1 Application Structure

In order to present the crawling and indexing sections, we detail the application structure (DOM) of the AJAX News application of the Example in Section 2 in Figure 3.1 intuitively and 3.2 (as a DOM).

Next, we define the application model and how crawling can be optimally performed in order to build an optimal data structure for the application model. It is important to notice that the DOM presented in Figure 3.2 does not specify the textual information, i.e., it is just a skeleton of what will be used further on. However, for each state in the

Figure 3.1: Structure of the AJAX news application (Figure 2.1).

Figure 3.2: DOM of the AJAX news application (Figure 2.1).
application, the text under `newsTitle` and `newsContent` changes with each button click, generating a new DOM.

### 3.2 Application Model

Section 2 presented an AJAX application as not only a simple page identified by an URL, but as a series of states, events and transitions. The Application Model is a view on the application. In particular it is a graph, called Transition Graph. The Transition Graph contains all application entities (states, events, transitions) encoded as an annotated graph. 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 (position, visibility) and text.

- **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 a figure. Figure 3.3 models the `onClick` event activated on the `<a id = "nextArrow">` source, applied to the `newsTitle` and `newsContent` targets. The content of the targets changes using the action: `newsTitle.innerHTML = ...`

![Transition Graph for the AJAX news application (Figure 2.1)](image)

<table>
<thead>
<tr>
<th>Event</th>
<th>Start State</th>
<th>End State</th>
<th>Source</th>
<th>Event</th>
<th>Target</th>
<th>Action</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>s1</td>
<td>s2</td>
<td>nextArrow</td>
<td>onclick</td>
<td>newsTitle</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e1</td>
<td>s1</td>
<td>s2</td>
<td>nextArrow</td>
<td>onclick</td>
<td>newsContent</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e2</td>
<td>s2</td>
<td>s3</td>
<td>newsArrow</td>
<td>onclick</td>
<td>newsTitle</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e2</td>
<td>s2</td>
<td>s3</td>
<td>newsArrow</td>
<td>onclick</td>
<td>newsContent</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e3</td>
<td>s3</td>
<td>s2</td>
<td>prevArrow</td>
<td>onclick</td>
<td>newsTitle</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e3</td>
<td>s3</td>
<td>s2</td>
<td>prevArrow</td>
<td>onclick</td>
<td>newsContent</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e4</td>
<td>s2</td>
<td>s1</td>
<td>prevArrow</td>
<td>onclick</td>
<td>newsTitle</td>
<td>innerHTML</td>
<td>...</td>
</tr>
<tr>
<td>e4</td>
<td>s2</td>
<td>s1</td>
<td>prevArrow</td>
<td>onclick</td>
<td>newsContent</td>
<td>innerHTML</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table 3.1:** Annotation for the Transition Graph of the AJAX news application (Figure 2.1).

Furthermore, a corresponding table can be constructed as presented in Table 3.1 (the table is not fully normalized) and contains the information about the aforementioned entities.
3.3 Relational Structure

The previously shown application model can be mapped to a relational structure. Figure 3.4 shows the E/R diagram for the application model containing the following entities:

- **AJAXApplication** stores general information about the application, such as the DOM structure and the URL.

- **State** represents an application state by storing an ID, but no element data. An *AJAX Application* usually consists many *States*.

- **Transition** connect two *States*. A transition has exactly one start state and one end state, but two states can be connected be several transitions. Each transition stores the event-trigger, such as `onclick`, the Javascript, and has a relationship to a *DOMElement* defining the source element of the transition.

- **DOMElement** is the relational representation of a DOM element. The text content of a *DOM Element* is only defined when connected to a state, this is why its value is stored in the relationship between *State* and *DOM Element*.

- **JSVariable** is similar to *DOMElement*, but applies to Javascript variables.

![E/R diagram of the application model](image)

**Figure 3.4:** E/R diagram of the application model (not all attributes are included)

3.4 Optimizing Crawling

The challenge of crawling is to index as many application states as possible. Each state is a DOM tree to which events are attached. However, especially since events leading to
CHAPTER 3. CRAWLING

a new state can be so granular as to change only a minimal amount the content from the page, it is important to minimize the number of generated DOM trees. As also shown in the Transition Graph of Figure 3.3, 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 nextArrow from state 1 or prevArrow from state 3. There are a few optimizations which can be applied:

*Identify duplicate states.* We do this by computing a hash value of the content and structure of the DOM and comparing it with existing values.

*Limit infinite state expansion.* The tree can become infinite if the same events can be invoked indefinitely on the same state. We limit the amount of iterations so that this case does not occur.

*Diff changes.* Currently, the change events of DOM elements are captured and indexed. The unchanged content is marked accordingly in the index and the information is used during ranking (Section 6 and query processing Section 5 (i.e., it gets a lower importance compared to changed content).

### 3.5 Implementation

In our Java implementation of the crawler, we make use of existing components to load and parse HTML (Cobra, [6]), and to evaluate Javascript (Rhino, [17]). Cobra can be used to load a web page from a given URI, where initial JavaScript is automatically executed and a document object model (DOM) of the HTML structure is created. We extended the Cobra Toolkit with our own implementations for HTTP requests, as the provided classes do not support the HTTP POST method, which is necessary for XMLHttpRequests (the AJAX calls). Furthermore, we have our own implementation of the UserAgentContext, which is mainly used to disguise as a commonly used web browser (currently Internet Explorer 6). This is necessary as some web servers present simplified web pages if the browser is not known to support AJAX. There are many more additions and mostly bug-fixes to Cobra, which are described in Section 3.7.1.

Another addition done to Cobra is the ability to attach listeners for the changes done to the DOM. We used the well-known observer pattern. A Java class implementing the `JSListener` interface can subscribe to actions possibly changing the DOM, such as `innerHTML`, `innerText`, `document.write`. Furthermore, changes done to Javascript variables are also notified. This is necessary to build the application model and roll back changes done by a JavaScript call in order to reach all states.

#### 3.5.1 Class Structure

Figure 3.5 shows the most important classes of the AJAXCrawler. Cobra and Rhino are displayed as packages.

- **AJAXDocument** is the main entry point for the crawler. Given a URI, 
  *AJAX Document* loads the web page, parses it using Rhino to create the initial DOM (*HTMLDocument*), and executes the initial Javascript if any.
  *AJAX Document* extracts all Javascript events from the elements in the DOM and provides methods to execute custom Javascript or execute all events. The initial state is stored in the database. Whenever a piece of Javascript is executed, a transition is created in the database, along which all affected DOM elements and
3.6 Executing all Javascript Events

To execute all possible Javascript combinations, AJAXDocument needs to be able to restore a previous state. This means we need to be able to roll back the changes a certain piece of JavaScript does. We are using two methods undo() and redo(), which can undo the changes done to the DOM by using the data from the database, or redo a Javascript not by actually executing it again, but by re-setting the values in the DOM Elements and Javascript variables using the database. Using these two methods, we implemented a gotoState(State) in AJAXDocument, which can go from the current state to any other. Note that a path can always be found, since the crawler can only reach states by executing Javascript. Figure 3.6 shows an example of an application with two changeable sections. It is an extension of the previous news application with another element Quote, where a new quote can be loaded by invoking nextQuote(). Note that there is no prevQuote() method. In a given state, the crawler invokes the Javascript methods in this order: nextQuote(), next(), prev(). Assume the current state is state 8 (red). After executing nextQuote(), we reach a new state, numbered 10 (green). Since the crawler now wants to execute next(), it first needs to do that by restoring state 8. The method gotoState() will find the shortest path from state 10 to state 8, which in this case consists of one undo(). The crawler can then continue executing the next method on state 8.
3.7 Comparing States

Determining whether two states can be difficult. Currently, we’re comparing the hash value of the full serialized DOM. This means that the DOM structure must match with all attributes and text. However, this strict equality could result in too many states. For instance, if a Javascript moves an element by a few pixels only, we may want to consider this as the same state.

3.7.1 Fixes to Cobra and Rhino

Using Cobra and Rhino, we were able to implement our fully Javascript-aware browser in little time. It worked well on simple applications such as the news application [2], however, when we tried to crawl a more complex application like GMail or Yahoo! Mail, we discovered that there were many things missing which are available in Internet Explorer, which is the browser we are trying to emulate.

- **debugger (reserved keyword)** used as variable name in the y!mail application
  
  To solve this issue, we changed the interpreter settings to allow debugger as variable name, since its not used elsewhere. The affected class is `src.org.mozilla.javascript.TokenStream`.

- **clearInterval** and **setInterval** functions not implemented. These functions are essential for the application and therefore were implemented in the class `src.org.xamjwgh.html.js.Window`.

- **_defineGetter_** and **_defineSetter_** not properly implemented (as in Firefox).
  
  Internet Explorer does not support these functions, and therefore we do not implement them. In the beginning, when we started with disguising as Mozilla, we would have needed this functionality. In the y!mail application, these methods are used to add XML-related functionality such as `appendChild`, `getChildNodes`,
3.7. COMPARING STATES

insertAdjacentHTML. By using Internet Explorer, we could take advantage of the corresponding DOM-methods, or, in case of insertAdjacentHTML, implement it in Java, which made things easier.

- *Number.valueOf()*: the expression `1280.valueOf()` was not interpreted correctly, the interpreter passed number as argument to a static function `valueOf`. This was solved in `ScriptableObject.getNameFunctionAndThis()`.

- *XML HTML Elements*: Internet Explorer supports XML HTML elements of the form

  ```
  ⟨xmlid="someId"⟩
  ⟨root⟩
  ⟨data/⟩
  ⟨data/⟩
  ⟨/root⟩
  ⟨/xml⟩
  ```

  When such xml elements are encountered, the xml document is parsed and made available through the id in the window scope. For instance, JavaScript code can access the above xml document using `window.someId.XMLDocument.documentElement` or `window.someId.documentElement`, which both return a pointer to the xml document. Internet Explorer then supports a variety of methods to manipulate these XML documents, including `selectNodes` and `selectSingleNode`, which take XPath expressions. All the necessary methods are implemented with the help of xerces, the Apache implementation of the DOM. The core class for this is `org.xamjwg.html.domimpl.HTMLXMLXmlElementImpl`, which takes the raw XML string from the parser and converts it to a valid XML document using the class `XMLDocumentImpl`. Elements in the DOM are of type `org.apache.xerces.dom.ElementNSImpl`, as the XML documents may include namespaces. We also had to make sure that the elements support the non-DOM `selectNodes` and `selectSingleNode` methods. A new class in `org.xamjwg.html.domimpl.HTMLXmlElementBuilder` makes sure that an instance of `HTMLXMLXmlElementImpl` is created when encountered in the HTML document.

- *onLoad* not in correct scope: For a reason we could not figure out, in the y!mail application the function called on the event `body.onload` was never found, although it was correctly parsed in stored. We could fix the problem by changing a few lines in `org.xamjwg.html.js.Executor` to make sure the `onload` function is always found. Similar problems occurred with other variables, which were solved in the same way.

- *window.location* provides access to methods which control the browser, and return information about the current location. The necessary location object was implemented according to http://developer.mozilla.org/en/docs/DOM:window.location.

- *Object enumeration*:

  JavaScript supports enumeration of objects:

  ```javascript
  var anObject = {a : 1, b : 4, c : 19};
  for(var nm in anObject) {
      document.write(" + " + anObject[nm]);
  }
  ```
Object enumeration was properly implemented for JavaScript objects, but not for native objects such as window.location. To provide enumeration also for native objects, changes were made to src.org.xamjwg.js.JavaClassWrapper and src.org.xamjwg.js.JavaObjectWrapper to return all ids and not only the ones from the variables set by JavaScript.

- `String.localeCompare(other)`
  Internet Explorer has a method `localeCompare` on strings, which takes another string and returns -1, 0, or 1 indicating which of the strings lexicographically comes first. `localeCompare` takes locality into account. Rhino didn’t have the function implemented at all, which caused an error in y!mail. We implemented a simple version of this function in src.org.mozilla.javascript.NativeString, which uses the Java method `String.compareTo(String other)`.

- `document.all[elementId]` and `window[elementId]`
  Elements can be found by `document.all[elementId]` or even `window[elementId]` in Internet Explorer.

- `element.id`
  The id of an element can be received by using `element.id` in Internet Explorer.

- `appendChild appends node even if not same document`
  If trying to append a node to a document which belongs to a different document, an error is thrown. However, Internet Explorer imports the node first if this is necessary.

- `oncut, onpaste, oncopy added for all HTML elements`

- `HttpRequestImpl.send()`
  To do AJAX calls, `XMLHttpRequest` does an http post request. While `XMLHttpRequest` was properly implemented, `HttpRequestImpl` did not support http post, only get. We added http post functionality to `src.org.ajax.implementations.HttpRequestImpl`, the first step towards an AJAX-aware crawler.

- `Cookies`
  the Cobra package did not implement cookies at all. However, it was necessary to support cookies to allow accessing secured websites, such as the y!mail application. We did not fully implement cookies (that is, separate the cookies for different domains and expire cookies), however, from a technical point of view, cookies are fully supported in both, http requests and JavaScript. The necessary modifications where done in the following classes:
  `src.org.xamjwg.HttpRequestImpl`: receives new cookies and sends all existing ones
  `src.org.ajax.implementations.UserAgentContextImpl`: stores the cookies
  `org.xamjwg.html.domimpl.HTMLDocumentImpl`: the JavaScript interface to the cookies
3.7. COMPARING STATES

- **Incremental parsing**
  
  Another nasty problem for which it was hard to detect the cause is the parsing order. Normally, a HTML document is parsed from top down. However, since JavaScript can change the document while it is being parsed (using `document.write` and `document.writeln`), these changes need to be reflected immediately. Consider the following example page:

  ```html
  ⟨html⟩
  ⟨head⟩
  ⟨scripttype = ”text/javascript” language = ”javascript”⟩
  var aScript = ”⟨script⟩var x = ’Hello World’; ⟨/script⟩”;
  document.write(aScript);
  document.write(x);
  ⟨/script⟩
  ⟨/head⟩
  ⟨body⟩
  JavaScript says :
  ⟨/body⟩
  ⟨/html⟩
  ```

  In Internet Explorer (or any other browser), the page would read "JavaScript says : Hello World". However, the Cobra Implementation failed, as it could not find the variable `x`. The reason was that the whole content of the script element was interpreted, but the written content not immediately parsed. Dynamic content is always written after the current script element, and the parser automatically reaches this new content. However, we need it immediately after it is being written. To solve this, we changed the parser to incrementally parse new content as soon as it is written. This incremental parsing is triggered in the `write` and `writeln` methods of `HTMLDocumentImpl`.

- **iframes and frames**
  
  Implementation of frames an iframes was missing, and another problem was a mixup of the name and id attributes, which applied to all html element. In standard HTML, they are not the same, many elements can have the same name, while the id needs to be unique. Cobra used to overwrite the id value with the one of name, which lead to a number of problems. Also, the frames need to be made available to javascript, to change its location.

- **Deferred Javascript**: Internet Explorer supports an attribute defer for script elements, which defers the execution of the JavaScript source contained in this element. y!mail uses this fact to check for the document to be loaded, which means that we needed to implement this functionality in order to make y!mail work.

- **getResponseHeader** return invalid result AJAX applications use `getResponseHeader` to determine when an AJAX call returns or if it failed. The Cobra implementation did not take care of the fact that many headers can have the same name, and therefore, the underlying JavaScript failed.
• cloneNode was not correctly implemented

3.8 Store Implementation

We decided to implement our own Store according to 3.4, as we have mixed data in the form of basic types, XML, and DOM Elements. Figure 3.7 displays the UML diagram of our database in Java. The Store is able to hold the complete application model for many ajax applications. Although we used Java for the implementation, we tried to keep the Store relational, meaning the objects only store values, but do not provide any methods (except the creators). This would allow us to easily move to stronger database in the future.

Figure 3.7: The UML diagram of the Store

The class DataBase should be used as static instance. It is the only class needed to interact with the Store, providing all the necessary features to insert, update, in receive values. DataBase contains lists of all other classes, which can be interpreted as relations. All objects have a field sysID, which is their position in the list and also used for serialization. DataBase further maintains indexes for quick access to elements:

• index_states an indexing linking state ids (set by the database user) to the state object
• index_domelements index from HTMLElement to the Store DOM Element
• index_domelements_dewey an index to find elements by their dewey id
• index_domelements_id to find elements by their id attribute
• index_jsvariables finds jsvariables by their name
The reader may notice that the Store is completely in memory, yet *DataBase* provides methods to serialize the data and to load it at a later point. Only the data is serialized, the indexes are recreated when the data is loaded again. This also applies to the inverted file, which is detailed in Section 4.
Chapter 4

Indexing

In traditional information retrieval, an index is an inverted file 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 application URI, a state and the information on the element containing the text. As we will define in Section 6, ranking might be influenced by the prominence of a certain part of the document in the whole application cycle. As a consequence, the result of the indexing phase is an enhanced inverted file, also containing the AJAX-specific information.

4.1 Index Construction

Indexing is an operation which starts from the Application Model (the Transition Graph) and builds the physical inverted file. As an example, the inverted file for for the AJAX News Application in Figure 2.1 is presented in Table 4.1. The enhanced inverted file contains a link to the document containing the word, to the state containing the word and to the element in the DOM which contains the corresponding keyword. The score is computed based on the number of occurrences of the word in the state. The rest of the information about the elements and their participation in the application is also encoded in the Transition Graph - a structure which is also needed in the phases of Query Processing (Section 5) and Result Aggregation (Section 7).

<table>
<thead>
<tr>
<th>Word</th>
<th>URI</th>
<th>State</th>
<th>Element</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ross</td>
<td>s1</td>
<td><a href="http://www.giannifrey.com/ajax/news">www.giannifrey.com/ajax/news</a></td>
<td>newsTitle</td>
<td>1</td>
</tr>
<tr>
<td>perot</td>
<td>s1</td>
<td><a href="http://www.giannifrey.com/ajax/news">www.giannifrey.com/ajax/news</a></td>
<td>newsTitle</td>
<td>1</td>
</tr>
<tr>
<td>investoren</td>
<td>s2</td>
<td><a href="http://www.giannifrey.com/ajax/news">www.giannifrey.com/ajax/news</a></td>
<td>newsTitle</td>
<td>1</td>
</tr>
<tr>
<td>ein</td>
<td>s2</td>
<td><a href="http://www.giannifrey.com/ajax/news">www.giannifrey.com/ajax/news</a></td>
<td>newsContent</td>
<td>1</td>
</tr>
<tr>
<td>ein</td>
<td>s1</td>
<td><a href="http://www.giannifrey.com/ajax/news">www.giannifrey.com/ajax/news</a></td>
<td>newsContent</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.1: Inverted File for AJAX news application (Figure 2.1).

In our implementation, the index is directly linked to the database storing the application model. Whenever a new state is created in the database its content is indexed. Since the text content is actually stored in HTMLElements, the DOM tree is traversed, and for each element, the following actions are taken:

1. The database DOMElement is retrieved or created if it does not exist
2. The text content of the element is extracted and set as content of the `DOMElement` in the current state

3. The content is tokenized and each keyword is added to the index, along with information on the state, element and position within the element

The structure of the index is shown in Figure 4.1, and Figure 4.2 shows its implementation in Java.

### 4.2 Keyword Tokenizer

In order to generate a powerful index, the keyword tokenizer cannot just split along some stop characters. For instance, the . is normally considered as stop character, however, if found between numbers, it should not be used as such.

As shown in Table 4.2, `KeywordTokenizers` contains two sets of characters: separators and concatenators. The input is split at separators and also concatenators, unless the characters before and after a concatenator are numbers. Using this technique, we properly split into words, numbers and dates.

<table>
<thead>
<tr>
<th>Separators</th>
<th>[newline] [tabulator] [carriage return] [space] ; : . + &quot; &amp; = ? ! { } [ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenators</td>
<td>. - /</td>
</tr>
</tbody>
</table>

Table 4.2: Stop characters and concatenating characters in `KeywordTokenizer`
4.3 Keyword Rewriting

Another technique used to optimize the index is keyword rewriting. The goal is to group keywords with the same meaning, but different notation. Currently, we apply keyword rewriting to dates. The \textit{KeywordRewriter} tries to recognize dates in the format mm/dd or mm/dd/yy and rewrites them as mm/dd/yyyy, where the current year is used if it is missing.

Keyword rewriting has two purposes. First, the user of the search engine does not need to try different date formats to obtain results as the search keywords are also rewritten. Second, for correlation between parts of a web application (Section 4.4) we also need to normalize dates in order to compare them (Figure 4.3).

4.4 Correlations

After crawling the a web application and building the index, further processing can be performed on the index. Consider a slightly enhanced web application which not only shows news, but also a calendar with holidays (Figure 4.4). Both the news and the calendar feature previous and next buttons to display another piece of news or holiday respectively.

Consider such an application with 8 news items and 5 calendar items. Our crawler will index 40 states by combining every piece of news with every calendar entry. While this is correct in terms of the application model, the developer might want to limit the states to only those who have matching dates in the news and the calendar. In other words, the developer wants to correlate news and calendar.

Up to this point, the crawling and indexing of a web application was a completely automatic process. By introducing the concept of correlations, the developer of a web application can enhance the search engine performance by specifying rules to limit the generated states. A rule can be specified using an XPath-like syntax, for example

\[
// [id = 'article_intro'].text()[2] \text{eq} // [id = 'caltext'].text()[2]
\]

This rule would correlate the date found in the news article with the date from the calendar. \texttt{// [id = 'article_intro']} gets element with id \texttt{article_intro}, \texttt{.text()} extracts its text and \texttt{[2]} takes the second keyword. It is important to notice that we can access the list of keywords, as in the case of dates, the text to correlate may differ in its length and therefore, extracting text using \texttt{substring} may not be suitable.

If given such rules, the crawler starts to drop states which do not satisfy the expression. Figure 4.5 shows an example of news and calendar combinations, indicating which states
Figure 4.4: An enhanced newsreader application with calendar

are valid (green) and which are not (red) according to the date-matching rule. For news items that have no matching holiday (blue), we keep all states. This is necessary, as we would lose information if these states would be dropped. Later, in Query Processing (Section 5), similar states will be grouped to avoid showing the user duplicate results.

Figure 4.5: States of the news and calendar application. Green states have matching dates.

In a real-world scenario, the developer of a web application could provide a specially named XML file containing the correlation rules for the AJAX applications on the server, similar to a robots.txt file.

4.4.1 Correlations in AJAXSearch

In AJAXSearch, Correlations are expressed in XPath, currently only supporting a subset of the language. To find an element, the expression `// *[@id = 'element_id']` must be used. The text content of an element can be obtained using the function `text()`. The keywords of a text expression are organized in an array, where the i-th element can be accessed using `[i]`.

AJAXSearch can compare any kind of keywords, not only dates. By comparing the rewritten form of the keywords AJAXSearch can properly recognize equal keywords even if they are not represented equally. In the example of dates, 12/25/07 would match 12/25/2007.
The rule is applied after crawling the application, all States not fulfilling the rule are marked as invalid and not returned in results.

4.5 Optimizing Indexing

The complexity of the indexing process is directly proportional to the size of the application model defined in Section 3.2. Therefore, the large size of the model in case of complex application with a rich structure and several elements can also negatively influence the indexing. There are still steps which can be performed to improve this: index pipelining and incremental indexing. We implement the first step, i.e., crawling and indexing are pipelined in the final implementation and words retrieved by the crawler are immediately pushed to the IndexWriter. The latter step was not needed for our test applications (i.e., the AJAX news application) and is left as future work.
Chapter 5

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.

5.1 Processing Simple Keyword Queries

The index constructed in Section 4 can be used to extract this information as shown in Table 5.1. It shows the results of two queries: war and won. Each query returns the URI, the state, and the element containing the keywords. Additionally, a rank is computed as it will be shown in Section 6.

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

Table 5.1: Query Processing on the AJAX news application.

5.2 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 State is the same. Since results are returned in the granularity of elements, the URL and the state must be the same between the two entries of the posting lists. An additional test is performed to check if the elements belong to a compatible element. In that case, the Least Common Ancestor (LCA) of the two corresponding elements is returned.

As an example, let’s take the query: Oscar won. This will result in the conjunction between two posting lists. The posting lists of Oscar and won are presented and merged
in the first row of Figure 5.1. The second row indicates the second phase of Processing Conjunctions, and shows how incompatible states are eliminated and how the LCA is used for one case.

<table>
<thead>
<tr>
<th>List 1: &quot;Oscar&quot;</th>
<th>List 2: &quot;won&quot;</th>
<th>Result: &quot;Oscar&quot; ∧ &quot;won&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL1, s2, newsContent</td>
<td>URL1, s1, newsTitle</td>
<td>URL1, (s1, s2)</td>
</tr>
<tr>
<td>URL1, s2, newsTitle</td>
<td>URL1, (s2), newsContent</td>
<td>URL1, (s2), newsContent</td>
</tr>
</tbody>
</table>

**Phase 1: Merge.**

Result 1: URL1, (s1, s2)

Result 2: URL1, (s2), newsContent

**Phase 2: Eliminate Incompatibilities and Compute LCA.**

In the results of Figure 5.1, the incompatible states are eliminated (and stroked out). The result is given by ⟨www.fakenews.com, s2, news⟩. It can be noticed that s2 contains both keywords. The result where the posting lists specify s1 and s2 is invalid (and stroked out in Figure 5.1). The rank is not mentioned here, but is computed as shown in Section 6, based on the importance of the keywords and resulted states. The first valid result is the one where the two keywords are contained in s2, in newsContent. The second compatible result is the one when the two keywords are contained in s2, but won is contained in newsTitle and Oscar is contained in newsContent. In this case, the LCA of the two elements is returned, as shown in the DOM Structure of Figure 3.2, is the Element news. As a mention, we compute the LCA using special hierarchical IDs, Dewey ID [20] assigned during indexing, as shown in Figure 5.2. The rank of the LCA is computed using an average of the ranks of the individual elements (Section 7.2).

**Figure 5.2: Dewey ID assigned to the application structure.**

If the Dewey IDs of more DOM Elements share the same prefix, then the LCA is computed as the LCA of the DeweyIDs. An index indicates which elements correspond to the result DeweyID. In this case, in state s2, the Dewey ID newsTitle is 1.3.1, and the Dewey ID of newsContent is 1.3.2. The LCA of 1.3.1 and 1.3.2 is 1.3. This corresponds to the element with ID news. This approach is compatible with XML and XML-IR specific techniques such as [3] or XRank [13].

### 5.3 Implementation

In this section we will discuss the implementation details for processing queries. It is assumed that a complete application exists in a serialized form on the disk. A search application would then load the data into memory and create both the indexes and the inverted
file according to Sections 3.8 and 4. Our query processing application, **AJAXSearch**, uses the same database classes as the crawler and shows a GUI to the user for entering a query and display the results. Figure 5.3 shows the UML diagram of our search application.

![UML Diagram of AJAXSearch](image)

**Figure 5.3: UML diagram of AJAXSearch**

The main entry point for a query is the method `query(String)` in the class `DataBase`, which passes the call to `InvertedFile`. When a query is issued, the following actions are taken:

1. **Tokenizing query** exactly like indexed text content, the query is split into keywords and keywords are rewritten (Sections 4.2 and 4.3).

2. **Retrieving keyword to states lists** for each keyword, the list of states and elements is obtained using the inverted file. Since the keywords with their attributes are organized in a hash table, this operation can be done in linear time.

3. **Merging lists** the lists of keywords to states are merged using an incremental merging algorithm. This is possible because the keyword to state lists come sorted by state. The result of the merging is a `QueryResult` instance (Figure 5.4), which corresponds to exactly one state, provides a method to compute the rank, and stores information about the keyword positions in this state.

4. **Result aggregation** as will be discussed in Section 7, the results are grouped in order to show the user a minimal amount of states containing all relevant information. In AJAXSearch, we group states to one result if for every keyword, the content of the element in which they appear is the same (Section 7.1)

5. **Computing ranks** for each state (or group of states), the rank is computed according to the techniques described in Section 6

6. **Sorting results** finally, the results are sorted according to the rank. AJAXSearch uses an in-place quicksort algorithm [15] which on average runs in $O(n\log(n))$.

![UML Diagram for a Query Result Item](image)

**Figure 5.4: UML diagram for a query result item**

The `query()` method returns an array of `QueryResults`, which then are displayed in a formatted output to the end user. Each result entry will show the following items (Figure 5.5):
• The state Id as link to the cached version of the state
• The rank of the state
• The LCA of the elements containing the keywords
• The path as Javascript methods to recreate the state with a link to open the application and go to the state
• A snippet with highlighted keywords

If the user clicks on a link to display a cached version, the corresponding DOM is written as .html file to the disk and opened in the default browser. The goal is to move the search application to a web server, which could send the cached state directly to the client. To recreate a state, a special html page is used, which shows the real application in an iframe and invokes all Javascript methods until the desired state is reached.

Figure 5.5: AJAXSearch Application
Chapter 6

AJAXRank

Ranking an AJAX Application raises both challenges and opportunities. Instead of relying on a static, flat page model as in traditional IR, a more dynamic ranking method, based more on XML-IR approaches, can be defined. The challenges arise from the need to assign a weight not only to the page, but also to the application state and to individual elements in the page.

6.1 Ranking Requirements

We can summarize the special requirements for ranking in an AJAX application as follows:

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

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

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

Before more details, we mention that in this Section we use the application model in Figure 6.1 in order to illustrate the capabilities of ranking. It is still a news application which has the states A, B, C, X, D, where A is the first state (i.e., the first piece of news) and D, the last one.

In order to have a more realistic scenario, we assume the application also has a link from the main page to a special piece of news (the favorite), which we choose to be X. By
adding this link, the application model is not symmetrical anymore and is more representative for the ranking function. Figure 6.1 already assigns weights to the edges, equally distributed between the outgoing edges. We detail the building blocks of AJAXRank as follows.

6.2 Ranking Application States

Ranking Application States is the most important building block of AJAXRank. It encompasses the fact that an AJAX application is composed of states connected through transitions which correspond to events. We define this by adapting PageRank [5] to the specific requirements of an AJAX Application. Furthermore, we also show that PageRank itself does not apply well to an AJAX Application.

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

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

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

The intuition behind PageRank [5] is:

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

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

- **A user cannot jump to any state randomly.** This is only valid to the first state. The first state can be reached either directly through the URL which points to that state, or from any other state of the AJAX application.
6.2. RANKING APPLICATION STATES

- “Three-click-rule”. Given a chain of states linked by events, the user is most likely to browse just through some of them (in an extreme case just three) as it gets further from the initial state. The name “three-click-rule” is frequent in web usability studies; we adopt it here and adapt it to the requirements of an AJAX Web Application and of search.

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

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

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

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

Similarly to PageRank, AJAXRank has a result of vector of values computed using an iterative algorithm. The iteration defines a convergent series, the convergence limit being the vector with the final values of the AJAXRank. We now compare AJAXRank and PageRank.

6.2.1 AJAXRank vs. PageRank

We applied PageRank and AJAXRank to the example in Figure 6.1 and it converges after 24 iterations. This improves with the number of states as shown in Section 8. We concluded that the most accurate value for the \(d\) parameter is lower than in case of PageRank, and we chose \(d=0.75\). Intuitively, this is lower than the usual value for \(d\) in PageRank (e.g., 0.85) because in case of AJAX Application the probability to jump to the first state \((1-d)\) is a lot higher than in case of PageRank, applied to the whole web. The results of the ranking for PageRank and AJAXRank, and the order of the states, are presented in Figure 6.2.

<table>
<thead>
<tr>
<th>State</th>
<th>PageRank ((d=0.85))</th>
<th>AJAXRank ((d=0.75))</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.1048</td>
<td>0.3173</td>
</tr>
<tr>
<td>B</td>
<td>0.1763</td>
<td>0.1791</td>
</tr>
<tr>
<td>C</td>
<td>0.2388</td>
<td>0.1609</td>
</tr>
<tr>
<td>X</td>
<td>0.316</td>
<td>0.2491</td>
</tr>
<tr>
<td>D</td>
<td>0.164</td>
<td>0.0936</td>
</tr>
</tbody>
</table>

**Order:** X, C, B, D, A

Table 6.1: Comparison: AJAXRank vs. PageRank

Obviously, PageRank ranks better the states which are highly linked to (i.e., X as the mostly linked, then C, B, D, as moderately linked, and only then the start state A). Therefore, PageRank ignores the importance of the starting state - a fair result for the web. On the contrary, in an AJAX application, the starting state can be reached from all other states, or just by typing the URL of the application. Also the importance of the other states decreases as they are further away from the states which are directly reachable (A,
CHAPTER 6. AJAXRANK

36

Figure 6.2: AJAXRank vs. PageRank in the application model of Figure 6.1.

X) (“Three-click-rule”). This is not reflected by PageRank since the first state is poorly ranked and this is an argument which makes PageRank not suitable for AJAX application. On the other side, AJAXRank favors the first state by insuring that there is a possibility of a random walk to it from all other states with probability \(1-d\) where \(d=0.75\). Naturally, state X is also highly ranked since it is easily reachable. The other states, B, C, D are ranked increasingly lower, conformant to the “three-click-rule”. We further detail this point as follows.

6.2.2 Influence of the Click-Distance

The advantage of AJAXRank vs. PageRank in order to favor states which are closer to the states that are directly reachable is better illustrated if the model is simplified, such as the one presented in Figure 6.3, corresponding to a double-linked list of states (e.g., news items accessible from the main page by clicking on the next or previous arrow, and state X is not favored through an additional link).

Figure 6.3: Influence of click-distance on AJAXRank and PageRank.

As also shown in Figure 6.3 (right side), PageRank favors the middle states, states A and D being ignored as they are in the extremities. AJAXRank acts according to both semantics of the first state and to that of the distance semantics. When additional links are present, AJAXRank also boosts middle states as has been shown before. The states which are further away from A rank low. They will rank however higher in the final result based on the importance of their content.
Another interesting case is the one with a singly linked list. Consider an AJAX application that shows news, but a user can only go the next news item, not back to the previous. PageRank would put more weight to the states in the end, as they have a higher probability to reach by either going directly there, or following the next link from a previous state. The initial state has the lowest rank, as the only way to reach it is by directly jump there. Figure 6.4 shows a comparison between PageRank and AJAXRank for the single linked list. Since AJAXRank allows jumps only to the initial states, it receives a much higher rank, which is obviously more suitable for an AJAX web application.

![PageRank vs. AJAXRank in the case of a single linked list](image)

**Figure 6.4: PageRank vs. AJAXRank in the case of a single linked list**

### 6.3 Ranking Application Events

We presented in Section 3.2 the different types of events that can appear in the document. In particular, the events are characterized by a **type** of an event triggered on a **source** element, and on one or more **targets** to which **actions** are applied. Each state is derived from another state based on a single event. We use this information for assigning a different weight to edges than the weight based only on the number of outgoing edges. In particular, we use the **type** of event and the **actions** applied on the target elements in order to determine the importance of an event (an edge in the graph).

Each edge between two states is annotated during crawling with the event information as shown in Figure 3.3. Each transition \( tr \) between two states \( s_i \) and \( s_j \) is annotated with a source \( src \), one event type \( e \), more targets \( t_i \) and corresponding actions \( a_i \), as shown in Figure 6.5. Each target \( t_i \) has a single corresponding action \( action_i \) (such as innerHTML, move, etc.).

The actions and events have a relative importance which influences the weight of the transition edge. In particular, we defined the importance of actions and events as scores, as shown in Table 6.6a for events and Table 6.6b for scores. The scores are empirically defined based on the observed frequency of these events and on the importance of the modification caused by the actions.

The intuition is that click events (onClick, onDblClick, etc.) rank higher than mouseover events. The rest of the events are classified equally, as shown in Table 6.6a. The actions applied to the targets also have a differentiated importance. Table 6.6b shows that the
greater importance is given to actual content change (innerHTML, outerHTML), visibility change (using properties of the stylesheet), append at the end of the document (document.write) and then other move and change events. Consequently, after defining the importance of actions and events, we distribute these weights on the edges. The weight of each edge is proportional to the importance of the corresponding actions and events.

The weight of an edge $s_i, s_j$, generated by the event $e$ on source element $s$, applied to the targets $t_i$ through the actions $a_l$ is:

$$w(s_i, s_j) = \frac{1}{\sum_{s_k \in H(s_i)} w(s_i, s_k)} (score(e) \cdot \sum_{a_l \in A(s_i, s_j)} score(a_l)),$$

where $s_k$ are all states reachable from $s_i$ and $A(s_i, s_j)$ are all actions that the event applied on the targets. This formula ensures that the sum of all weights of the outgoing edges is 1. The new weights influence AJAXRank and will be used accordingly in Section 8.

## 6.4 Ranking the Application Structure

The goal of AJAX Search is to correctly identify, index, search and rank the relevant parts of AJAX applications. These consist of application states. However, the results presented to the user should also contain information about the part of the page into which the content appears, i.e., the corresponding DOM element. The intuition is: if an element is a frequent target of changes, it is more important. More information on the actual AJAX application can be therefore derived.

We can decide the importance of an element based on another view of the application
model: the Annotated Application Structure. We model it as a DOM annotated with information on the elements whose content changes, or which act as source or target of an event in the Transition Graph (see Section 3.2). The importance is computed based on the element’s participation to roles (source or target), events and actions.

An example of the annotated Application Structure is shown in Figure 6.7. Each node is annotated with the event, role and operation (event type, e.g., click or action, e.g., innerHTML) to which it participated, based on Table 3.1.

![Figure 6.7: Annotated Application Structure with (Event, Role, Operation) for the AJAX news application (Figure 2.1)](image)

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

Table 6.2: Annotation for the Application Structure of the AJAX news application (Figure 2.1).

This information at hand, the ranking of an element can be relatively easy computed as the average of its contributions throughout the application. We base the result on the already defined scores of the events (Table 6.6a) and actions (Table 6.6b). Furthermore, we define a score on whether an element acts as a source (less important) or a target (very important since the content changes), as shown in Table 6.3.

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

Table 6.3: Scores for Roles
In order to compute the \( ER \) (ElementRank), we consider an element \( E \) which participated to \( N \) events. We encode each event with \( event_i \), the role with \( r_i \) (either “source” or “target”) and the operation with \( o_i \). If \( E \) was a source (less important), then \( o_i \) encodes the event type \( e_i \) which triggered the transition (e.g., click); if \( E \) was a target (more important), then the \( o_i \) encodes the action \( a_i \) applied on it to change the content. The \( ER \) is computed using the following formula:

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

The astute reader may notice that it is not always possible to generate a stable DOM, valid throughout the application, simply because the page might change completely during the transitions. However, this is possible if the structure is stable, which is the case of the simple news application but also Google Maps, GMail or Yahoo! Mail. We leave the most complex case as future work. In this case, the structure of the application (such as the one in Figures 3.1 and 3.2) is stable and only the content of individual subparts changes. As shown in the experimental section, the Yahoo!Mail application also exhibits the same stable behavior, so this assumption was sufficient.

We discussed in the Introduction the need to be flexible as what concerns the results returned to the user. They may be just the URL of the whole application (to be in the extreme of traditional search) or a full AJAX link needed to reconstruct the whole application state of the result. There is a varying degree of flexibility even inside the same application state: results can be returned either at a very granular level (i.e., the application “state” + the most specific DOM element containing the keyword) or very loose (e.g., return the application “state” but return a coarser DOM Element of the application, which is more prominent (e.g., the E-mail pane) and contains the keyword at a possibly lower level). The latter case is important since the application model contains enough information in order to reflect the relative importance of the application elements in the model. As a consequence of this fact, we propose a ranking propagation mechanism to be used, in order to allow elements which by themselves are neither sources nor targets of events, to also receive an aggregated ranking from the sub-elements involved in events (Section 7).

### 6.5 Cumulative Ranking

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

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

The AR of a state and ER of an element have already been defined. The \( tf \) and \( idf \) are defined as follows, and can be computed at indexing time, based on the following formulas:
6.5. CUMULATIVE RANKING

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

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

The \( tf \) of term \( k \) in state \( s \) is the term frequency of term \( k \) in a state \( s \), normalized with report to the count of all terms in state \( s \). The \( idf \) of a term \( t \) is the log of the number of total states divided by the total number of states the keywords appears in. Therefore, the semantics of the formulas follows the well-known semantics in IR [4], with states acting as documents. Of course, the \( tf \) and \( idf \) values are computed at indexing time and just read at query time.
Chapter 7

Result Aggregation

The purpose of the phase of Result Aggregation is to present results to the user. Several techniques are necessary, and are detailed below:

1. **Grouping Results.** States and elements are very granular. Grouping of results is necessary for usability.

2. **Rank Aggregation.** A Ranking strategy for grouped results is necessary.

3. **Reconstructing Application States.** Results are complex links to application states. States must be reconstructed, just as traditional IR returns the initial pages.

### 7.1 Grouping Results

Since any minor event generates a state, a lot of almost identical states can be generated. It is therefore clever to group the result so that they are more compact. We apply the following strategy: if the result occurs in many states, but always in the same element, an operation similar to a `GROUP BY` element and state is performed. We give an example in Table 7.1.

<table>
<thead>
<tr>
<th>Query</th>
<th>URI</th>
<th>State</th>
<th>Element</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>won</td>
<td>s1</td>
<td>newsContent</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>newsTitle</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>newsContent</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>newsTitle</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>quote</td>
<td>0.4</td>
<td></td>
</tr>
</tbody>
</table>

**Phase 1: Initial results, no aggregation.**

<table>
<thead>
<tr>
<th>Query</th>
<th>URI</th>
<th>State</th>
<th>Element</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>won</td>
<td>(s1,s2)</td>
<td>newsContent</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>newsTitle</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>quote</td>
<td>0.4</td>
<td></td>
</tr>
</tbody>
</table>

**Phase 2: Results after grouping.**

Figure 7.1: Grouping query results.
Phase 1 in Table 7.1 shows the results of the query without grouping. Phase 2 shows how the entries with the same URI and same element ID have been grouped, independently of the state they appear in. In particular, a single result is displayed to the user who can select which state it wants to see from the aggregated result (for example from a context menu). In particular the rank is aggregated as the average rank of the combined result, as shown in Subsection 7.2.

7.2 Rank Aggregation

The problem of grouping raised the problem of rank aggregation. Rank aggregation is needed in two contexts:

(i) Rank aggregation in case of grouping (Subsection 7.1). In case of grouping (shown in the example in Figure 7.1), two or more results are combined if they appear under the same element, but in different states. The common group rank of \( n \) grouped result entries \( res_1, \ldots, res_n \) with cumulative ranks \( rank(res_1), \ldots, rank(res_n) \), is computed as:

\[
rank(res_1, \ldots, res_n) = \frac{\sum_{i=1}^{n} rank(res_i)}{n}.
\]

(ii) Rank aggregation in case of LCA (Subsection 5.2). The LCA of two elements is computed during Processing of Conjunctions (Section 5). For example, as shown in Figure 7.2, the LCA of “newsTitle” and “newsContent” is “news”. The LCA was computed during Query Processing, as shown in Figure 5.1.

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

\[
rank(LCA(e_1, \ldots, e_n)) = \sum_{i=1}^{n} rank(e_i).
\]

One can draw similarities between this and XRANK [13]. Generally in an XML structure, the rank of the top elements influences the rank of the low element and vice-versa. In case of applications, we only propagate the rank to the top of the DOM, from very specific elements (e.g., newsContent), to less specific elements (e.g., body), higher in the hierarchy (but maybe more important visually).
7.3 Reconstructing Application States

The results of the query processing phase are links. As opposed to traditional search, a link is a function from the application to the application states. The results are presented to the user as links, along with a summary of the text in the current section. The user can then decide to view a cached version of the state or have the state reconstructed in the real application. In the case of the cached version, the DOM can be obtained from the state, which stores the value of all elements. If the user wants to go to the real application, the following actions have to be taken:

- Compute the path from the initial state to the result state.
- Open a special web page, which includes the original application, and pass the Javascript to invoke.
- The web page invokes the Javascript methods one by one, waiting for AJAX calls to return, and the user obtains the reconstructed state in the application.
Chapter 8

Experimental Results

We implemented a prototype version of AJAX Search and we applied it to a custom-built AJAX Web Application (a more complex News Application). The goal of the experiments were:

1. Derive statistics in terms of number of states and transitions of a real AJAX Application.
2. Show superiority of AJAXSearch over a Traditional Search Engine (i.e., no client-side-logic).
3. Analyze Index Size and Query Processing Time for an AJAX Web Application.

8.1 Experimental setup

In the experiments we compare two types of search engines:

1. **Traditional Search Engine** (*TradJS*). We configure AJAXSearch to read just the first state of the AJAX application. This is the *Content-at-a-time-approach* from Section 2. The version with no Javascript capabilities is not evaluated since it is too poor in result quality.
2. **AJAXSearch**. This is the full-fledged AJAXSearch engine, with full capabilities of client-side code evaluation, enhanced crawling, indexing, query processing as presented in this work.

Based on the initial DOM Model of the page, the Transition Graph was built (as in Section 3.2) and, based on it, an inverted file was constructed and serialized. Generically, since a dynamic web page can react to many events applied to many HTML elements, we restricted the number of indexed entities to those in Table 8.1. We argue in favor of this choice because we use the most commonly used elements to which events are attached (*a*, *div*) and the events are simple click/move events (*onClick*, *onDbClick*, *onMouseMove*). The actions are also the main ones corresponding to a change in the document content (e.g., *document.write*, or *innerHTML = ...*) or in the importance of parts of the page (e.g., *el.visible = true/false*). Therefore, this choice ensured a high quality and relevance of the constructed index.

We tested the results on the AJAX News Applications which has just simple events. This application is similar to that presented in the paper, but more complex and custom built by us. The test application is used to show results and control the application, at a small scale.
CHAPTER 8. EXPERIMENTAL RESULTS

We used the following configuration for the tests: MacBook w/ OS X 10.4.10, 2 GHz Intel core Duo, 2GB Memory (667 MHz DDR2 SDRAM), Harddisk: 120 GB, Serial ATA, 5’400 rpm, 8 MB buffer. External tools were used in order to crawl and index the AJAX web application. The open source Javascript toolkit COBRA [6] was used to execute the Javascript code embedded in the web page. AJAX calls were also possible through the toolkit. The Xerces [23] XML parser was used to process XML parts of the code. The COBRA toolkit was set to emulate Internet Explorer during crawling. Several modifications were applied to the framework in order to correctly emulate the behavior of a real browser and were programmed explicitly.

8.2 AJAX News Application

The test AJAX News Application was at the base of our running example of this paper, and it is found at this URL[2]. The text consists of real news, in German. The simple body consists of a side menu with title news, a news title section and a news content section. News are changed by using two buttons (next and previous arrows) or the side menu. Therefore, the model is more complex: each piece of news is linked to all other pieces of news through AJAX events because of the side menu, and all news have next previous links to the next piece of news (as Shown in Figure 8.1).

![Figure 8.1: Model of the experimental AJAX News Application.](image)

Each click causes an AJAX Call to be sent to the server to fetch the news, and the interface is changed through innerHTML events. The statistical information on number of states, events, actions, and number of keywords of this application is listed in Table 8.2.

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

<table>
<thead>
<tr>
<th>Triggers</th>
<th>Events</th>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>onClick, onDblClick, onMouseOver</td>
<td>document.write(...)</td>
<td>a div</td>
</tr>
<tr>
<td></td>
<td>innerHTML=...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e1.visible = true/false, e1.position.x = ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e1.backgroundColor = ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e1.text = ...</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Indexed Entities in AJAX Web Applications
8.3 Application Model Size

We crawled the AJAX news application and created the application model (transition graph), when the number of states varied from 8 to 48. The number of transitions is quadratic since states are linked to each other and two buttons link “neighboring states”. As the number of states is increased, the number of events increases accordingly, causing a number of AJAX Calls and innerHTML invocations to several elements for each AJAX Call, as shown in Figure 8.2.

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

Table 8.2: Number of AJAX Entities in the sample News Application.

Figure 8.2: Number of transitions (AJAX Call events) and actions (innerHTML invocations) with report to the number of states in the AJAX News Application.

The number of transitions affects the size of the application model, which grows with the number of transitions (quadratic in the order of states), as shown in Figure 8.3.

Figure 8.3: Size of Application Model(kb) for AJAX News Application.
This affects the crawling process as shown in Figure 8.4. Actually, crawling time is taken by the invocation time for AJAX Calls and is polynomial in terms of the number of states. Crawling varies with the number of transitions, which is quadratic in the number of states for this application.

<table>
<thead>
<tr>
<th>States</th>
<th>Parsing(secs)</th>
<th>Crawling(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3.18</td>
<td>77.517</td>
</tr>
<tr>
<td>16</td>
<td>3.05</td>
<td>274.951</td>
</tr>
<tr>
<td>24</td>
<td>3.27</td>
<td>671.454</td>
</tr>
<tr>
<td>32</td>
<td>3.77</td>
<td>1317.997</td>
</tr>
<tr>
<td>40</td>
<td>2.94</td>
<td>2167.426</td>
</tr>
<tr>
<td>48</td>
<td>2.94</td>
<td>3445.568</td>
</tr>
</tbody>
</table>

Table 8.3: Crawling and DOM Parsing Time of the AJAX News Application.

Figure 8.4: Crawling time for different number of states in the simple AJAX News Application.

### 8.4 Index Creation Time

The largest amount of time in building the index is taken by the previous phase: crawling (building the application model). Indexing creates the enhanced inverted file (Section 4) and is a fast process (milliseconds) compared to Crawling (seconds), as shown in Table 8.4.

<table>
<thead>
<tr>
<th>States</th>
<th>Crawling Time(secs)</th>
<th>Indexing Time(ms)</th>
<th># AJAX Rank Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>78</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>248</td>
<td>44</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>671</td>
<td>122</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>1318</td>
<td>611</td>
<td>14</td>
</tr>
<tr>
<td>40</td>
<td>2167</td>
<td>1413</td>
<td>14</td>
</tr>
<tr>
<td>48</td>
<td>3446</td>
<td>2884</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 8.4: Crawling Time(secs), Index Creation Time (ms), and Number of Iterations for AJAXRank for the AJAX news Application.

Crawling time is dependent on the number of transitions (which is, at its turn, quadratic in the number of states). However, very importantly, AJAXRank needs a constant number
of iterations until it converges, which is around 14 iterations using no optimizations like matrix decompositions, as shown in Table 8.4.

### 8.5 Index Size

It can be seen from Table 8.5 that also the size of the Index is significantly smaller than that of the Application Model, and it varies corresponding to the number of states and transitions. It makes sense to show them comparatively, as Figure 8.5 shows, since both need to be materialized for the purpose of Query Processing and Result Aggregation.

<table>
<thead>
<tr>
<th>States</th>
<th>App.Model Size(kB)</th>
<th>Index Size(kB)</th>
<th>Total Size(kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>359</td>
<td>53</td>
<td>412</td>
</tr>
<tr>
<td>16</td>
<td>1167</td>
<td>105</td>
<td>1272</td>
</tr>
<tr>
<td>24</td>
<td>2389</td>
<td>155</td>
<td>2544</td>
</tr>
<tr>
<td>32</td>
<td>3954</td>
<td>206</td>
<td>4160</td>
</tr>
<tr>
<td>40</td>
<td>6314</td>
<td>256</td>
<td>6570</td>
</tr>
<tr>
<td>48</td>
<td>9046</td>
<td>312</td>
<td>9358</td>
</tr>
</tbody>
</table>

Table 8.5: Index Size(kB), Application Model Size(kB) and Total Size(kB) for the AJAX news Application.

![Figure 8.5: Index Size(kb) and Application Model Size(kb) for the AJAX News Application.](image)

As mentioned in Section 4, the Index of the Transition Model is built as an enhanced inverted file. Table 8.6 shows the indexing times and the iteration needed to compute AJAXRank. Indexing is also polynomial in terms of states (Figure 8.6) and very importantly, AJAXRank needs a constant number of iterations until it converges.

### 8.6 Query Processing Time

We also evaluated query processing time with the random queries. Since we did it by reading the index in memory, query processing is in average below one millisecond, as reported in Table 8.7.

We mention again that the queries were two-keyword-queries, and the stopwords were eliminated, using a basic list found at [18]. This actually helped increase the Precision
Table 8.6: Index Creation Time(ms), Index Size(kB), and Number of Iterations for AJAXRank for the AJAX news Application.

<table>
<thead>
<tr>
<th>States</th>
<th>Indexing Time(ms)</th>
<th>Index Size(kB)</th>
<th>#AJAXRank Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>17</td>
<td>53</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>44</td>
<td>105</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>122</td>
<td>155</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>611</td>
<td>206</td>
<td>14</td>
</tr>
<tr>
<td>40</td>
<td>1413</td>
<td>256</td>
<td>14</td>
</tr>
<tr>
<td>48</td>
<td>2884</td>
<td>312</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 8.7: Query Processing Time(ms) for the AJAX News Application

<table>
<thead>
<tr>
<th>States</th>
<th>Query Processing Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.035</td>
</tr>
<tr>
<td>16</td>
<td>0.0521</td>
</tr>
<tr>
<td>24</td>
<td>0.0692</td>
</tr>
<tr>
<td>32</td>
<td>0.1123</td>
</tr>
<tr>
<td>40</td>
<td>0.1929</td>
</tr>
<tr>
<td>48</td>
<td>0.4878</td>
</tr>
</tbody>
</table>

Figure 8.6: Index creation time(ms) for the AJAX News Application.

8.7 Improvement in Search Quality

We conclude the evaluation by showing that our goal is fulfilled: return better search results to the user. Therefore, we evaluate the quality of search results of Traditional Search and AJAXSearch. Traditional Search just sees the non Javascript part causing almost only False Negatives and Recall 0, so we do not include it. We compare TradJS, Javascript-Aware, which sees the first piece of news (still causing many False Negatives) and AJAXSearch that sees all news. We generated 10000 random two-keyword-queries and ran them on the index. The recall values are presented in Figure 8.7 and show the clear advantage of AJAXSearch.

The correct result is composed of all news containing the query keywords. TradJS
Figure 8.7: Recall for Traditional Search and AJAXSearch.

can return, if any, just the first state. If valid for the query, the first state is considered
relevant for traditional search. AJAXSearch has Recall one. Queries with no result have
been considered as Recall 1 also for TradJS. Had we not done it, the Recall in case of
eight states would have been 0.08.

8.8 Yahoo! Mail

To compare our results also in terms of quality, we indexed a Yahoo! Mail account with
24 messages (Figure 8.8). Yahoo! Mail is a state of the art AJAX application allowing
browser based access to a Yahoo! mail account. When logging in, the user first sees the
home screen with news. On the left side, the application displays the folders panel for the
Inbox, Drafts, Sent Messages and so on. When clicking on a message folder or the Inbox
tab on the top, the user is taken to the message pane which shows a list of messages and
the message view. On all panes, a small calendar stripe is shown at the bottom displaying
holidays and items entered by the user.

Yahoo! Mail features a built-in search that has, as it is programmed by the develop-
ers of Yahoo! Mail, direct access to the data and knows about the application structure
and message formats. This is a clear advantage over our search engine, which is totally
unaware of the application and blindly indexes everything. However, we can show that
AJAXSearch can provide results of similar quality when concerning messages and also
returns other results such as news, which are also part of the application, but are not found
by the built-in search.

Table 8.8 shows some statistics for crawling Yahoo! Mail.

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawling time [s]</td>
<td>124</td>
</tr>
<tr>
<td>InnerHTML calls</td>
<td>1802</td>
</tr>
<tr>
<td>AJAX calls</td>
<td>8</td>
</tr>
<tr>
<td>States</td>
<td>25</td>
</tr>
<tr>
<td>Transitions</td>
<td>625</td>
</tr>
<tr>
<td>DOM Elements</td>
<td>1786</td>
</tr>
<tr>
<td>Keywords</td>
<td>2980</td>
</tr>
<tr>
<td>Index generation time [ms]</td>
<td>338</td>
</tr>
<tr>
<td>AJAX Rank iterations</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 8.8: Statistics for crawling Yahoo! Mail
It is interesting to notice that only 8 AJAX calls are required. The reason behind this is that Yahoo! uses a concept called **batch calls**, where multiple messages are retrieved at a time. Whenever the user loads a message, the next two messages are retrieved as well and stored in a local cache.

This batch execution also saves a lot on crawling time. Only 2 minutes are required to crawl the whole application with 25 states. The fact that 625 transitions were found show that the application model is a full graph where every state is linked with all other states.

We want to point out at this place that we needed to tweak the crawler to only perform two kinds of calls, loading messages and switching between home screen and message pane. If we would run the crawler without any limitations, that is, allow the crawler to click anywhere and therefore execute any Javascript, it would for example delete and reply to messages in the case of an e-mail application, or do a check out in an electronic store.

Figure 8.9 shows the ranking for our Yahoo! Mail account. The start screen has the highest rank, while all the messages receive the same rank. The reader may find the rank of the start state very high, but we want to emphasize that the state rank is only part of AJAXRank (Section 6) and as the examples will show, the high rank of the start state will not influence the overall performance of the ranking in a negative way (Section 8.8.3).

We decided on four hand picked queries for which we know where the results are located, i.e. in the messages, news, or GUI elements. The reason for the hand picked
queries is that we cannot perform random queries on Yahoo! Mail and we do not have enough data for representative tests. Table 8.9 shows the query keywords along with their location within the Yahoo! Mail application.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ajax</td>
<td>Messages</td>
</tr>
<tr>
<td>hollywood</td>
<td>News</td>
</tr>
<tr>
<td>trash</td>
<td>GUI and messages</td>
</tr>
<tr>
<td>notepad</td>
<td>GUI</td>
</tr>
</tbody>
</table>

Table 8.9: Keywords and their Location in the Yahoo! Mail application

In the following sections, we will discuss the results of the four queries. We only compare the quality of the results, not the query evaluation time. AJAXSearch takes sub-milliseconds to evaluate, while Yahoo! Mail requires about 3 seconds per query. While we realize that Yahoo! Mail needs to search a lot more data, we want to point out that the query time is very slow.

### 8.8.1 Search for ’ajax’

Some of the messages in our Yahoo! Mail account are about indexing AJAX applications. Searching for ’ajax’ should therefore return a subset of all messages. This query is useful to compare precision and recall of the built-in search and AJAXSearch, and to check the ranking. Table 8.10 shows the results of the two search engines. Yahoo! Mail returns results in the granularity of messages, whereas AJAXSearch returns states. As some of the states correspond to viewing a message, it is possible to directly compare the two. The messages (or corresponding states) are numbered from 1 to 24, 1 being the newest message.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo!</td>
<td>AJAXSearch</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.10: Result for ’ajax’

First thing to notice is that Yahoo! Mail returns two more results. This would mean that AJAXSearch has false negatives and therefore a lower recall of 0.833 compared to Yahoo! Mail (Precision is 1). However, when investigating the two missing results, we notice that the keyword ’ajax’ is included in an URL (Figure 8.10).

It is more or less a design decision, whether parts of URLs should be considered in the results. Since the whole URL is correctly indexed in AJAXSearch, it is only a matter of the tokenizer to also include URL substrings in the results.
Second, we notice that the order of the results is completely different between Yahoo! Mail search and AJAXSearch. Yahoo! returns the results ordered descending by date. This makes sense as we are dealing with mail messages. In the case of AJAXSearch, we cannot sort by date as the crawler does not know which is the message date. Since all messages have the same distance from the start state (i.e., their headers are displayed in the same state) and the keywords are only found in the same DOM Element, the ranking solely relies on term frequency / inverse document frequency. If not all message headers were visible at once and therefore older messages would require more clicks, the ranking would be different and correspond in some way to the message age, without taking the message date into account.

We want to point out that similar to correlations (Section 4.4), the developer of the AJAX application could provide AJAXSearch with directives on where to find the message dates, thus allowing AJAXSearch to better rank (or sort) query results.

8.8.2 Search for 'hollywood'

The term 'hollywood' is contained in the news which are displayed when logging into the Yahoo! Mail account. When searching for 'hollywood', Yahoo! doesn’t return a result, while AJAXSearch returns the start state (Table 8.11).

<table>
<thead>
<tr>
<th>Yahoo!</th>
<th>AJAXSearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.11: Result for 'hollywood'

It is unclear to determine whether Yahoo! Mail has a false negative or AJAXSearch has a false positive. If defining the whole account as AJAX application, the start page is clearly a part of it and should therefore also appear in the results, as a user may want to find that particular state. If the search should only apply to messages, the developer would need to add directives, similar to correlations, in order to invalidate states that do not display messages.

8.8.3 Search for 'trash'

The search for 'trash' is interesting as the keyword appears in both, a message and in the GUI. Table 8.12 shows the results for both search engines.

<table>
<thead>
<tr>
<th>Yahoo!</th>
<th>AJAXSearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>-</td>
<td>0, 1, 2,...</td>
</tr>
</tbody>
</table>

Table 8.12: Result for 'trash'
8.8. YAHOO! MAIL

Yahoo! Mail and AJAXSearch both correctly return the message containing the keyword. AJAXSearch also returns state 0 as result (grouped with all other states), since 'trash' can also be found in the GUI. It is important to notice that the ranking is strong enough to promote state 23 over the start state. The two results are not grouped as the results appear in different elements, which is a fair result. While one could argue that including all other states as result is a case of false positive, we want to point at that from an application model point of view, including the other states in the results appears to be correct. Since GUI elements are never affected by transitions, it would be easy to filter them out and thus receiving exactly the same result as Yahoo! Mail. For the before mentioned reasons of correctness, we decided not to do that.

8.8.4 Search for 'notepad'

Finally, by issuing the query 'notepad', we search for a keyword that is only contained in the GUI. As the reader may anticipate, Yahoo! Mail returns no results, while AJAXSearch returns a group of all states as one result item (Table 8.13).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo! Mail</td>
<td>AJAXSearch</td>
</tr>
<tr>
<td></td>
<td>0, 1, 2, ...</td>
</tr>
</tbody>
</table>

Table 8.13: Result for 'notepad'

It is again unclear which results is correct. As Yahoo! Mail search is intended for messages only, its result is certainly correct and does not have a false negative. On the other hand, AJAXSearch is meant for AJAX Application with unknown structure, and therefore, presenting the GUI as one result is correct as well.

8.8.5 Presenting the results

Yahoo! Mail has a clear advantage when it comes to presenting the results of a query (Figure 8.11) over AJAXSearch (Figure 8.12), mainly because Yahoo! is aware of the structure and can therefore display metadata to the messages. In AJAXSearch, this is not possible because we also need to display non-message results. We therefore have to find a general representation for search results.
Figure 8.12: Presenting the search results in AJAXSearch
Chapter 9

Tools

This section describes the tools that were used to build the crawler and AJAXSearch applications. All programming was done in Java, using Eclipse as IDE.

9.1 Cobra

Cobra is a HTML renderer and DOM parser supporting HTML 4, Javascript and CSS 2, written completely in Java. For the crawler application, we only need the DOM parser along with the Javascript and CSS packages.

The Cobra packages allowed us to quickly implement a fully Javascript-aware HTML parser which we could extend with the necessary functionality to execute Javascript and listen to DOM changes. However, due to a lot of missing features compared to browsers such as Internet Explorer, Firefox and Safari, we invested many hours to add the required functionality to achieve basic compliance with current web browser.

9.2 Rhino

Rhino is an open-source implementation of JavaScript written entirely in Java. Rhino converts Javascript scripts into Java classes and is used by Cobra to evaluate the Javascript and reflect the changes on the DOM.

9.3 Xerces

Xerces is a software package for parsing and manipulating XML, implementing the DOM. We use Xerces in the crawler to provide XPath functionality and XML manipulation.

9.4 Firebug

Firebug is a collection of web development tools, such as a DOM inspector, Javascript debugger and network monitor. We used Firebug to inspect the Javascript in the Yahoo! Mail application in order to make the crawler run on Yahoo! Mail. Yahoo! Mail uses obfuscated Javascript (Figure 9.1), which made it very hard to follow the call stack. Using the Javascript profiler of Firebug 9.2, we could obtain a list of all Javascript functions
involved in a certain action, for example, loading a message. Then, using breakpoints, we were able to obtain a stack trace and inspect the values of the variables.

Figure 9.1: One of the Javascript source files

![One of the Javascript source files](image)

Figure 9.2: Firebug displaying Javascript functions called for a search in Yahoo! Mail

![Firebug displaying Javascript functions called for a search in Yahoo! Mail](image)
Chapter 10

Conclusion

We presented an approach to automatically index AJAX applications. This is an area in which current search engines are far from competitive. Our approach is a first but powerful solution applied to existing complex AJAX applications. AJAXSearch and AJAXRank offer search capabilities over such applications. They bridge the gap between traditional search and application search: the results of the search are application-specific states, not only coarse pages. There are several avenues for future work. First of all, to apply it on any complex AJAX application, the crawler needs to be improved and probably built upon existing tools such as WebKit [21], and the crawler memory footprint can be improved: currently, redundant lookups may be performed, since identical states cannot always be identified. Incremental indexing can contribute to this. Secondly, an enhanced analysis of the content and of the quality of the transitions is necessary: not all states are equally important and not all application parts are relevant for search. Ranking and Correlations partially solved this issue, but additional collaboration of the application developer can help create an application model. Third, the framework assumes there is no connection to the server side. Especially in case of Enterprise Search, where internal data (such as E-mails or entire databases) is accessible for crawling and indexing, this knowledge can be combined with the client-side search engine in order to obtain fast access to data and better understand all interaction. Fourth: The ideas of lazy evaluation (as proposed for example in [1]) can be applied to this approach by adding lazy evaluation and expansion of the application model. Furthermore, constructing the scoring scheme can also be done lazily by using Top-k Techniques. From the practical perspective, a browser-embedded search function, which can correctly add search functionality to any loaded dynamic web application, is the low-scale version of this tool that can be implemented without great effort, by making use of the browser’s capabilities. On another level, the information generated by research or commercial Web Development Tools [22], [8], Frameworks such as JSF [16], Struts [19], or APIs such as the Google Web Toolkit [14] can be used to create an application model automatically. AJAX applications also start applying design patterns in the spirit of GoF [10]. Special search strategies relying on extracting relevant entities in the page (Information Extraction) can be applied in order to return even more specific results to the user. Last but not least, indexing more complex applications such as Google Maps [12] or using the process the model supplied by the application developer or produced by the developing environment with which the AJAX application was created. The goal is: better search results with minimal intervention.
Bibliography


[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, 1994.


