Indexing Web 2.0 Applications

by

Sumana Mohan

A PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of

the requirements for the degree of

Master of Science

Presented July, 2009

Commencement June, 2010

AN ABSTRACT OF THE PROJECT OF

Sumana Mohan for the degree of Master of Science in Computer Science presented on

Aug 10, 2009.

Title: Indexing Web 2.0 Applications.

Abstract Approved: _____

Dr. Bella Bose

This project aims at implementing Indexing for Web 2.0 applications. Ajax applications consist of a set of states which are generated by the user through actions such as click, focus, blur etc. events. By saving these DOM states we can index information obtained from dynamically generated web content. To prevent indexing of duplicate DOM states, a Tree Edit Distance algorithm known as Fast Match Edit Script has been implemented.

TABLE OF CONTENTS

1 Introduction

With the emergence of Web 2.0 technologies, developers are now incorporating Rich Internet Applications such as Ajax, Flash while developing their websites. Advantages of using Ajax includes

1. Allowing a webpage to request only the required content thus reducing the network bandwidth and load time.
2. By requiring only parts of the webpage to be reloaded, the user interface is more interactive and responsive with the user.

However these advantages pose a problem when it comes to indexing these web pages, as the traditional search engine crawlers (Google, Yahoo, MSN etc) do not execute the underlying JavaScript code. This results in a large portion of the web not being accessible. This is called the Invisible Web or the Deep Web. It consists of websites generated dynamically, which the traditional search engines do not retrieve. As of 2002, the Deep Web contains approximately about 91,000 terabytes of information, in contrast to the surface web which has about 167 terabytes. Traditional search engines most often ignore Rich Internet Applications or produce false positives and false negatives since they retrieve data asynchronously from the server without changing the behavior of the current webpage.

1.1. Existing Systems

Current Search engines, such as Google, try to solve the problem of indexing dynamic content in following ways:

1. Indexing the entire content of the webpage without executing the JavaScript code. As seen before this results in false positives and false negatives.

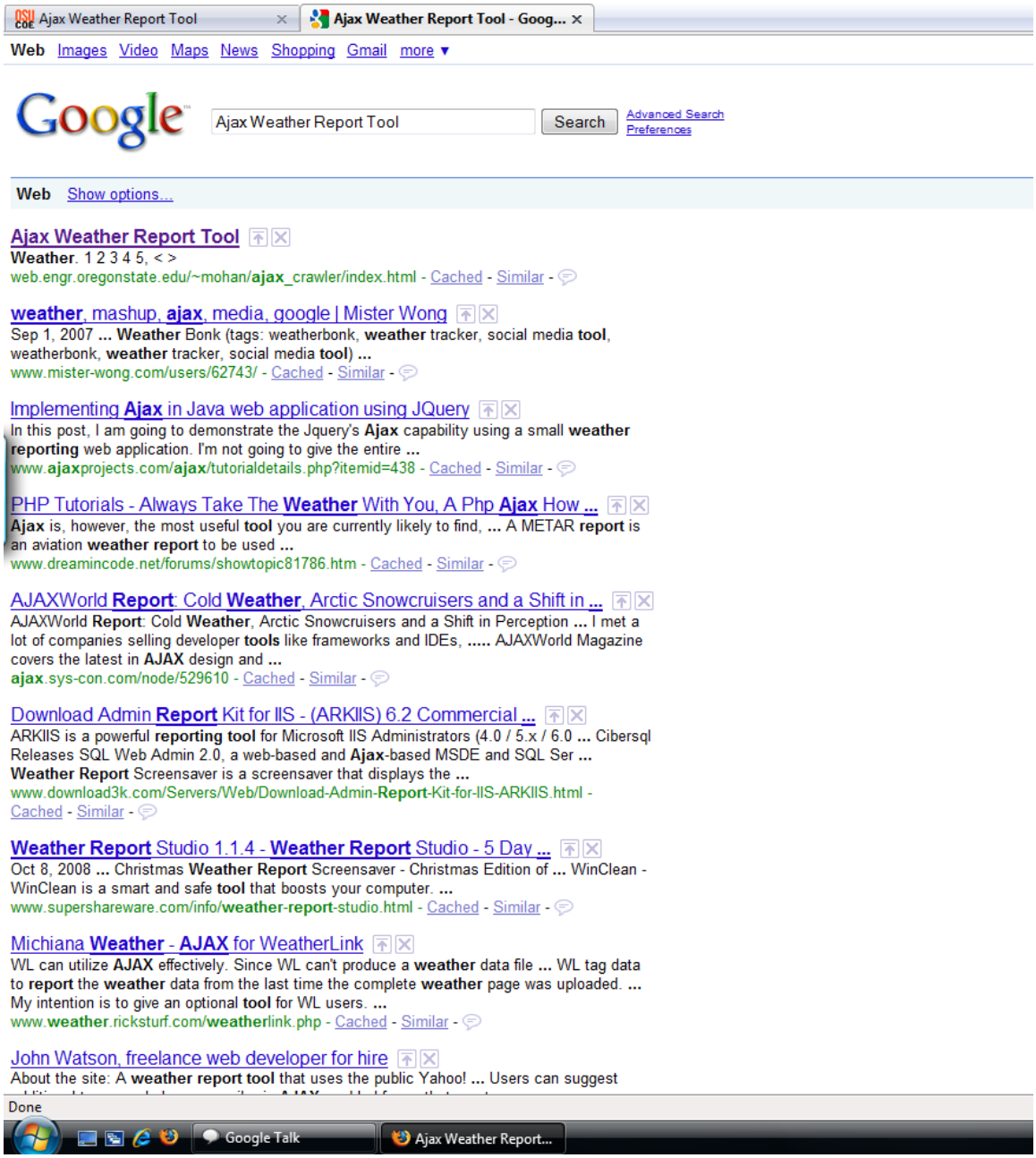


Figure 1.0: Search Results listing 'Ajax Weather Report Tool'

In the above Figure 1.0, when we search for *'Ajax Weather Report Tool'*, we get the result listing at the top of the page.

However when we search for any keyword, say *'Sunnyvale'* which is generated dynamically and hence not indexed by Google, we see the results as seen in Figure 1.1 which does not list the website.

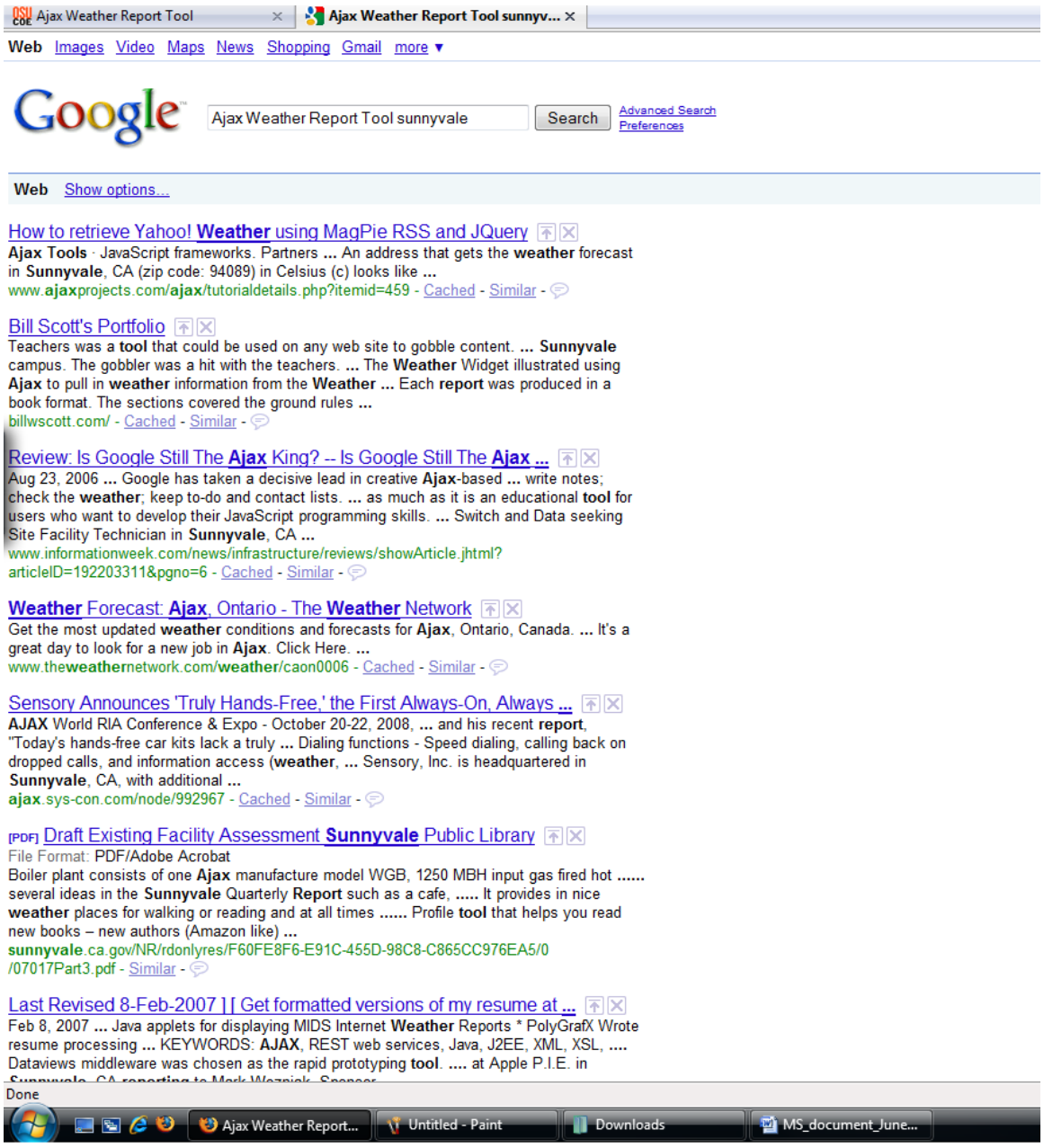


Figure 1.1 : Ex of False Negative - Search Results do no list 'Ajax Weather Report Tool' in search results

2. Exposing data to the search engines directly by hard coding the details. Although this may increase the relevancy of the content being indexed, the granularity becomes too coarse.
3. By embedding the search functionality in each application, which is something small application owners cannot afford.

They also provide alternative methods for indexing documents and accessing content that are otherwise not reachable through surface web. Sitemap protocol and mod_oai are some of them. While the sitemap protocol allows webmasters to inform crawlers which URL's are available for crawling through a defined XML file, mod_oai is an apache module which uses the Open Archives Initiative Protocol for gathering information from digital repositories. However none of these methods guarantee that the information being indexed is part of the Deep Web.

1.2. Motivation

Considering the size of the Deep Web mentioned earlier (91,000 terabytes of data) as opposed to 167 terabytes of the surface web, information which maybe very useful to users is not being indexed by traditional search engines. Part of this information is dynamic content generated using AJAX, Flash, Flex etc. Due to impedance mismatch, what the user sees is different from what the search engine is able to locate.
The user views the dynamic application as a series of states caused by events performed through actions by clicking a tab or a button, link etc. The search engine, on the other hand, views the application as a single state represented by a URL. Since in Web 2.0 applications, most of the actions happen asynchronously by changing states without changing the URL, the web crawler does not index the dynamic content.

1.3. AJAX Search

1.3.1 Ajax Search Model

An AJAX search engine has the following phases:

1.3.1.1 Crawling

The AJAX crawling algorithm first models the client code of the AJAX website.
In our case, the URL for prototype would be - http://web.engr.oregonstate.edu/~mohan/ajax_crawler/index.html.
Since Traditional search engines solve the problem of creating a hyperlink graph, the focus here is to allow the browser instance to read the initial DOM structure of the

webpage and execute the onload event. Crawling starts after this state has been initialized. The algorithm performs a Breadth First Search by identifying the events on the webpage and executing them. In our case only *"onclick"* events are identified. While crawling through an AJAX application it is important to be able to index as many applications states as possible. But since each application state is a DOM tree to which events are attached, the change from one state to another can be so minimal that identifying duplicate DOM states becomes crucial. This is done by using a Tree Edit Distance algorithm. Since the events being executed can infinitely expand when invoked indefinitely, a threshold is set to prevent the situation.

```
Function Crawl()
Goto webpage.
Identify urls on the webpage.
Add urls into urlQueue
   while(urlQueue is not empty)
     Pop first url from queue
    Fetch all clickable elements on
   webpage
   Add clickable elements to
  clickableQueue
    while(clickableQueue not empty)
          affectedElement ← Pop
clickable element
        Fetch onclick attribute of
clickable element
       result ← Execute javascript
event on attribute
   Add result to resultList
   Serialize document newDoc
     for(Document in resultList)
       Begin For
          Pop oldDoc from resultList
        Serialize oldDoc
       Diff(oldDoc, newDoc)
           if(Diff returns true)
              continue
              diff ← true
          else
              diff ← false
             break
      end For
if(diff == true)
storeIndex(newDoc,affectedElement)
```

```
Function StoreIndex(doc,affectedElement)
Begin
 endState ← createstate()
 tokens ← keywordTokenize(doc)
```

```
startState ← prevState
    foreach(keyword in tokens)
      Begin for
            storeKeywordStateInformation(keyword,endState)
     End for
 createTransition(startSate,endState,affectedElement)
End
```

Figure 1.2 : Crawling Algorithm

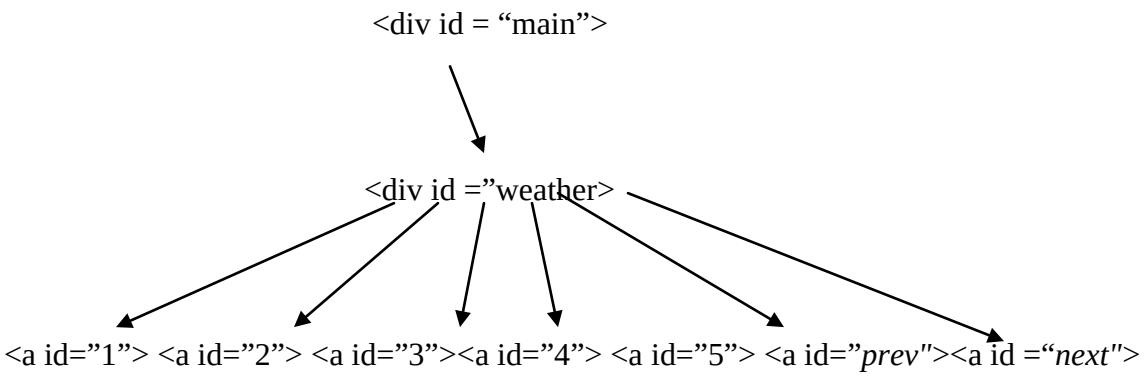


Figure 1.2 : DOM Structure of the AJAX Weather Report Tool

Each of the *'prev'* and *'next'* links are tied to onclick events. The div containing weather, is also tied to onclick events which load the respective content when the action is performed.

The application model represents a view of the application through a graph known as the Transition Graph. The Graph contains nodes and edges where nodes represent the DOM trees (application states). The edges represent a transition which was triggered by an event performed due to an action such as click, focus etc. Figure 1.3 shows the transition graph for the Ajax Weather Tool. Modeling the website, 'State 1' in the graph may represent the weather information regarding the city 'Sunnyvale', 'State 2' may represent weather information regarding the city 'Corvallis' etc. State 2 can be

reached from State 1 by clicking the 'next' button/element in our case thus representing a *Transition*.

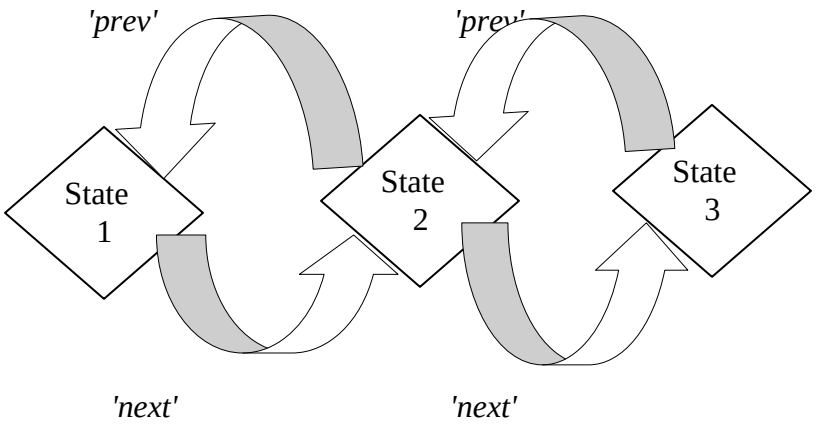


Figure 1.3 : Transition Graph for Ajax Weather Tool

1.3.1.2 Indexing

The indexing process for AJAX search is a little different compared to the traditional indexing techniques. In traditional search, indexing is a way to store documents which contain information (keywords) in the form of an inverted file. In the case of AJAX search, in addition we need to store the URI and the state information associated with it. Since AJAX applications are represented by a single URL, storing the state information along with it gives us access to keywords present at the state. An AJAX inverted file example for the Weather Tool is shown in Figure 1.4

Keyword	URI	State	Element
Sunnyvale	http://web.engr.oregonstate.edu/~mohan/ajax_crawler/index.html	S1	weather
Corvallis	http://web.engr.oregonstate.edu/~mohan/ajax_crawler/index.html	S2	weather
Partly cloudy	http://web.engr.oregonstate.edu/~mohan/ajax_crawler/index.html	S2	weather

Figure 1.4 Inverted Index for Ajax Weather Tool

1.3.1.3 Keyword Processing

While indexing, it's important to be able to tokenize keywords based on some rule. If keywords are tokenized using "period" as the separators, decimal numbers can be stored wrong. Hence in this project keywords are tokenized by eliminating common words known as stop words ('a', 'is', 'the' etc) which don't carry any meaning by themselves. Hence the indexes created contain only meaningful words.

1.3.1.4 Ranking

Ranking Ajax applications is a little different from the traditional ranking techniques being used today. Ex: PageRank system developed and used currently by Google.

This is because, unlike traditional indexing where crawling pages are the norm, indexing Ajax applications consists of crawling states. A few studies have shown why a new ranking system known as AjaxRank might be required and research is still ongoing in this area.

In this project, the ranking function tf-idf (Term Frequency- Inverse Document Frequency) has been computed by summing the tf-idf for each query term. tf-idf is a weight or statistical measure used to evaluate how important a word is to a document or a corpus. The importance of the word increases proportionally to the number of times it appears in the document but is offset by the frequency of the word in the corpus.

Term Frequency is defined as follows:

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$ where $n_{i,j}$ is the number of occurrences of the considered term (t_i) in document d_j, and the denominator is the sum of number of occurrences of all terms in document d_j.

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \in d\}|$: the number of documents where the term t_i appears

$$(\mathrm{tf\text{-}idf})_{\mathrm{i,j}} = \mathrm{tf_{i,j}} \times \mathrm{idf_i}$$

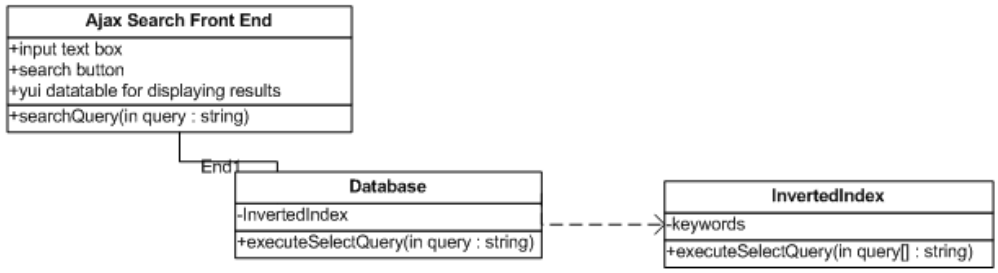
A high weight in tf–idf is reached by a high term frequency in the given document and a low document frequency of the term in the whole collection of documents. The weights hence tend to filter out common terms. The tf-idf value for a term will always be greater than or equal to zero.

1.3.1.5 Query Processing and Result Aggregation

Before we aggregate the results, it is important to process the queries, such that the states where the keywords appear or the DOM element which contains the elements can be presented to the user.

As shown in Figure 1.4, a query to fetch a keyword returns the URI, state information, element which contains it and an associated rank. When processing a group of keywords, the results returned would contain all the states and elements which contain it. Hence ranking algorithm is applied to sort results based on relevancy.

1.3.1.6 UML Diagram for Ajax Search Model



The results aggregated would contain the following information:

1. Real-time version of the state presented by a URL
2. The sequence of events to perform to reach the state
3. The resulting state containing highlighted keywords.

In this project, with the user point of view, in order to recreate a particular state, the real application is opened in an Iframe, and JavaScript methods are invoked on the corresponding document until we reach the required state.

1.4. HtmlUnit

In order to be able to execute events on any HTML page, we need to be able to simulate a browser. HtmlUnit provides the capability to simulate browser events such as click, blur, focus etc primarily for testing purposes. It is a GUI less browser for Java programs. It provides us with the common basic functionality such as invoking events, filling out forms, the same stuff we would do with a real browser. It's has good JavaScript support and is also able to work with quite complex AJAX libraries simulating either Firefox or IE.

Some other features are listed:

1. Support for the HTTP and HTTPS protocols, and cookies.
2. Allows you to specify how responses from the server should be handled.
3. Support for POST and GET.

4. Wrapper for HTML pages that provides easy access to all information contained inside them
5. Support for submitting forms, invoking events, walking the DOM model of the HTML document, basic and NTLM authentication.

2 Design and Implementation

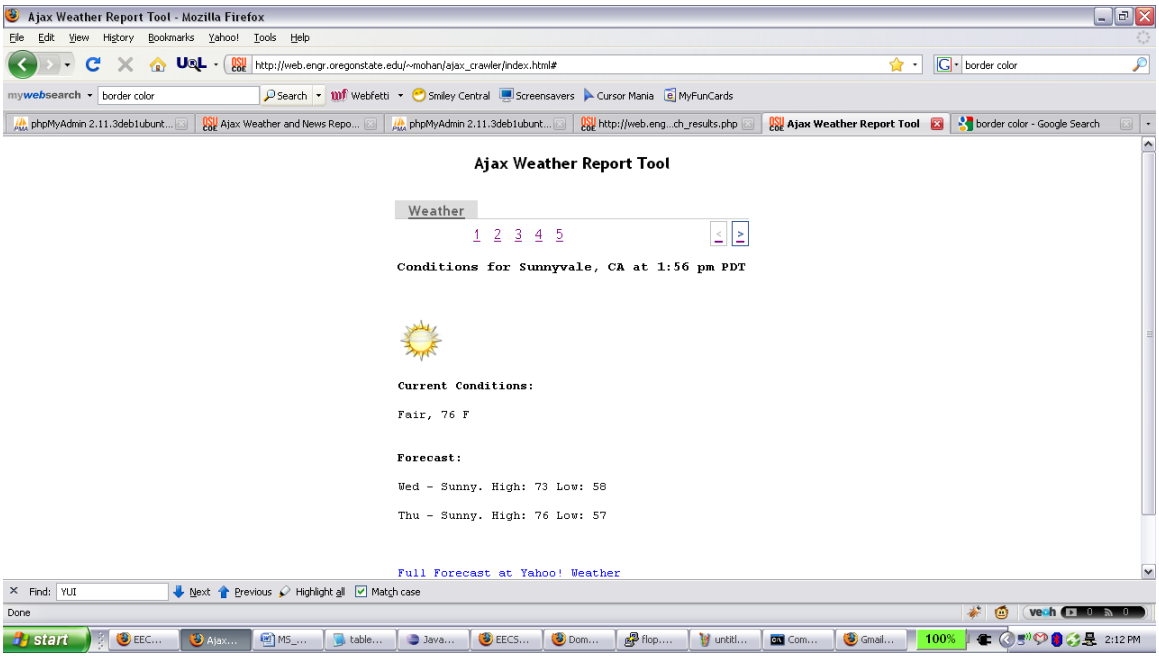


Figure 2.1: Ajax Weather Report Tool

'Ajax Weather Report Tool' shown in Figure 2.1 has been used to demonstrate the Ajax Search functionality. The Weather tool consists of a set of states which can be encountered while using either the '*next*' and '*prev*' buttons (shown as '<' and '>' in Figure 2.1) or using the numbered links 1,2,3,4,5. Each state is an Ajax call which retrieves weather information related to a particular city.

When the crawler first crawls the application, it identifies all of the clickable elements present within a particular state. In this project, only links are identified(represented by <a href> attribute associated with an onclick event). All of the clickable elements are added to a queue. And as long as the queue is not empty, each clickable element is removed from the queue and its corresponding onclick attribute is extracted and the JavaScript event is invoked on the element.

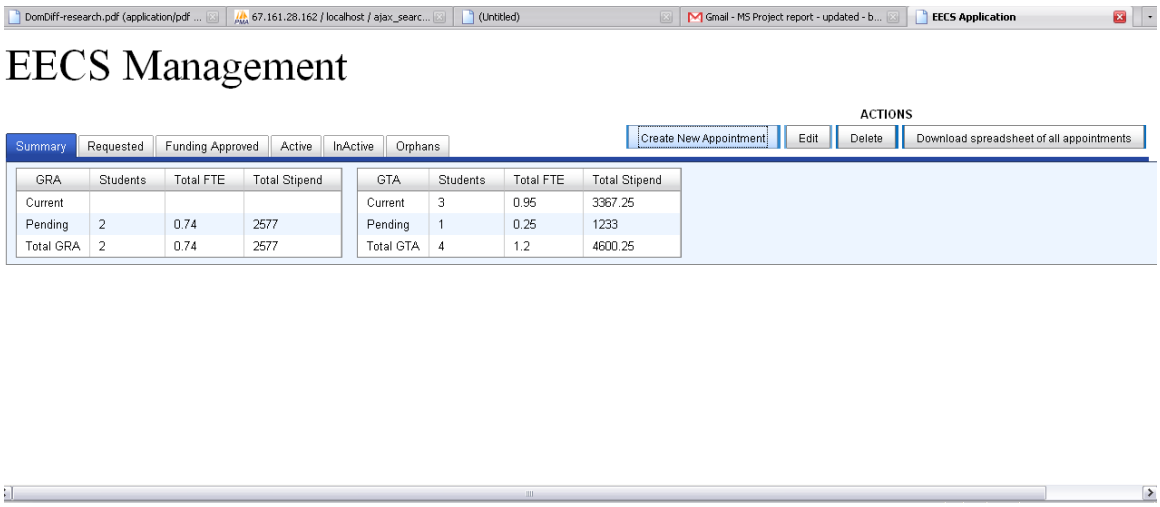
The resulting new htmlpage is extracted and added to a result queue. The above operation continues and each time a new html document is generated, it is compared to all of the existing html documents in the result queue. In order to be able to compare the html

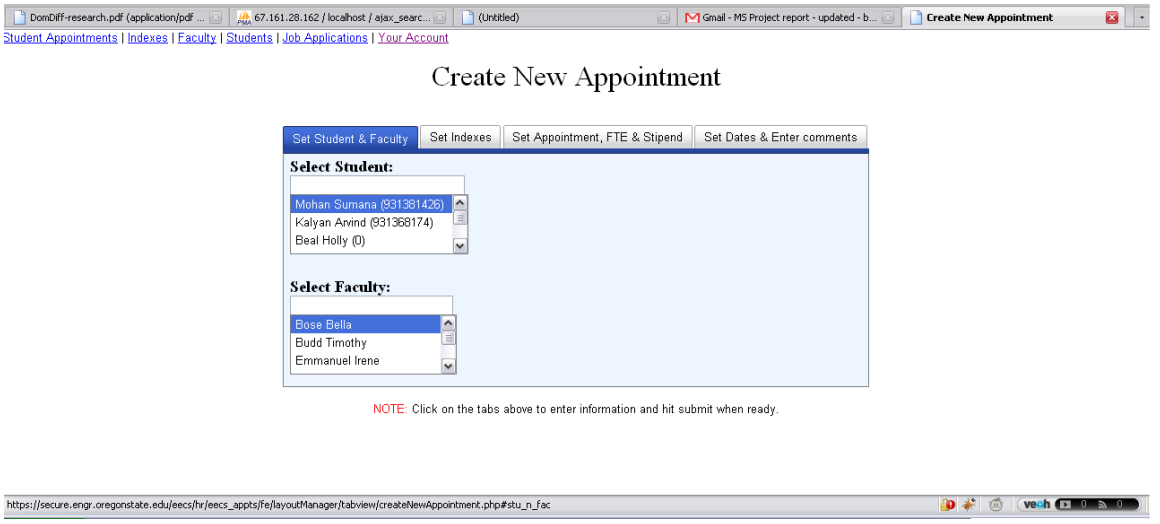
documents (or DOM elements) and store only unique DOM states, the tree edit distance algorithm known as the Fast Match Edit Script algorithm has been implemented. (Please see section 2.3 for more details on FMES algorithm)

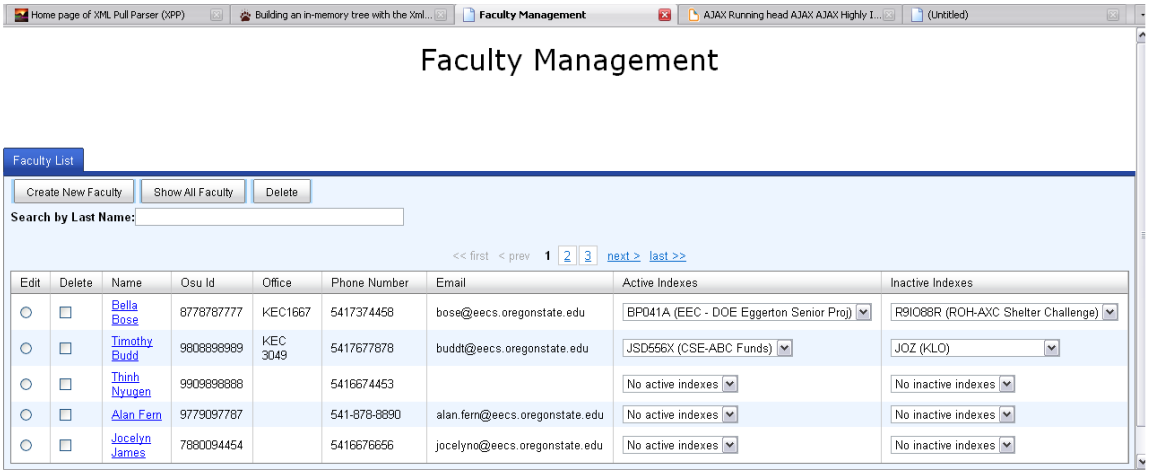
To illustrate with an example, we can reach the same DOM state in the weather application by clicking on the '*next*' button when in *state 1*, or by clicking on the numbered link *2*. (i.e performing any of the above actions brings us to the state containing weather information about *'Corvallis'*). Hence to avoid storing the same state twice, we need a diff algorithm which compares two tree documents, in this case being XML documents.

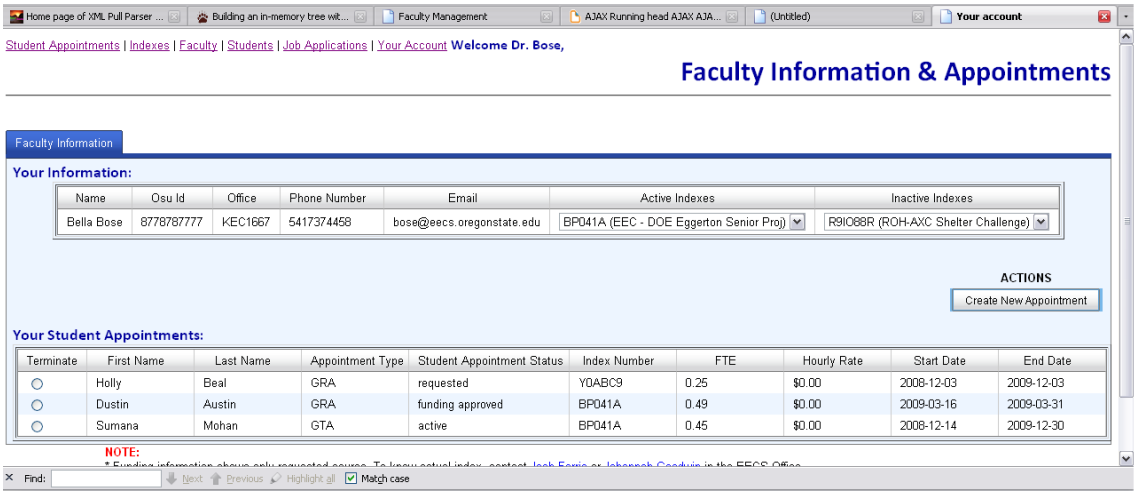
Finally, each resulting new DOM state is stored in the database. An inverted index is created in the database which stores the keyword information along with its state and the number of occurrences of the keyword in that state. Every state created also stores the respective DOM that was generated. Every time a new DOM is generated, a transition is created in the database to record the start state, end state, and the element which was affected during the transition.

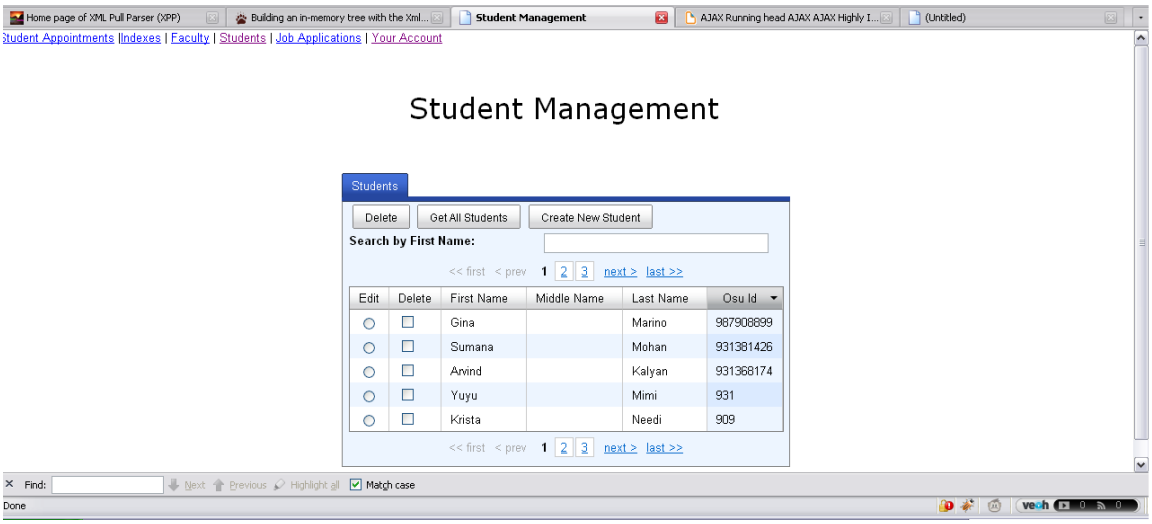
2.2 Ajax Web Application for EECS






Home page of XML Pull Parser (XPP)
Building an in-memory tree with the Xml...
Faculty Management
AJAX Running head AJAX AJAX Highly I...
(Untitled)
Faculty Management
Faculty List
Create New Faculty
Show All Faculty
Delete
Search by Last Name:
<< first < prev 1 2 3 next > last >>
Edit Delete Name Osu Id Office Phone Number Email Active Indexes Inactive Indexes
Bella Bose 8778787777 KEC1667 5417374458 bose@eecs.oregonstate.edu BP041A (EEC - DOE Eggerton Senior Proj) R9IO88R (ROH-AXC Shelter Challenge)
Timothy Budd 9808898989 KEC 3049 5417677878 buddt@eecs.oregonstate.edu JSD556X (CSE-ABC Funds) JOZ (KLO)
Thinh Nyugen 9909898888 5416674453 No active indexes No inactive indexes
Alan Fern 9779097787 541-878-8890 alan.fern@eecs.oregonstate.edu No active indexes No inactive indexes
Jocelyn James 7880094454 5416676656 jocelyno@eecs.oregonstate.edu No active indexes No inactive indexes


Home page of XML Pull Parser ...
Building an in-memory tree wit...
Faculty Management
AJAX Running head AJAX AJA...
(Untitled)
Your account
Student Appointments | Indexes | Faculty | Students | Job Applications | Your Account Welcome Dr. Bose,
Faculty Information & Appointments
Faculty Information
Your Information:
Name Osu Id Office Phone Number Email Active Indexes Inactive Indexes
Bella Bose 8778787777 KEC1667 5417374458 bose@eecs.oregonstate.edu BP041A (EEC - DOE Eggerton Senior Proj) R9IO88R (ROH-AXC Shelter Challenge)
ACTIONS
Create New Appointment
Your Student Appointments:
Terminate First Name Last Name Appointment Type Student Appointment Status Index Number FTE Hourly Rate Start Date End Date
Holly Beal GRA requested Y0ABC9 0.25 $0.00 2008-12-03 2009-12-03
Dustin Austin GRA funding approved BP041A 0.49 $0.00 2009-03-16 2009-03-31
Sumana Mohan GTA active BP041A 0.45 $0.00 2008-12-14 2009-12-30
NOTE:
Find: Next Previous Highlight all Match case


Home page of XML Pull Parser (XPP)
Building an in-memory tree with the Xml...
Student Management
AJAX Running head AJAX AJAX Highly I...
(Untitled)
Student Appointments Indexes | Faculty | Students | Job Applications | Your Account
Student Management
Students
Delete
Get All Students
Create New Student
Search by First Name:
<< first < prev 1 2 3 next > last >>
Edit Delete First Name Middle Name Last Name Osu Id
Gina Marino 987908899
Sumana Mohan 931381426
Arvind Kalyan 931368174
Yuyu Mimi 931
Krista Needi 909
Find: Next Previous Highlight all Match case
Done

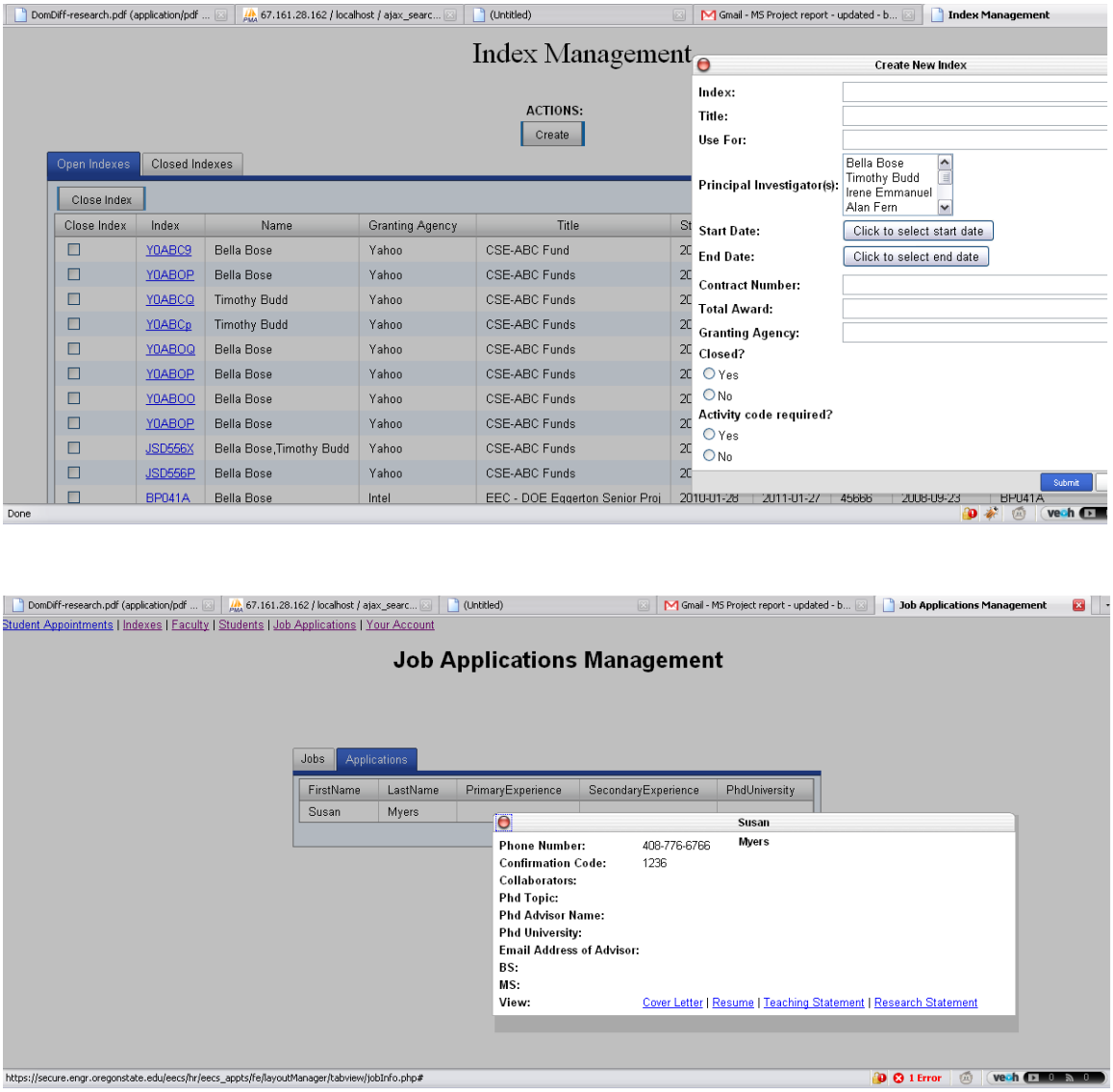


Figure 2.2: Screen shots for EECS Management, Create New Appointment, Faculty Management, Faculty Information & Appointments, Student Management, Index Management and Job Applications Management

2.2.1 Design

The EECS HR Application shown in Figure 2.2 was built using Yahoo User Interface components to not only enhance user experience by providing a rich user friendly interface but also improve the existing database design. The application consists of 6 different Ajax applications:

1. **Student Appointments -** The HR use this tool to identify what status the student appointments are in. i.e. whether they have just been requested, or under funding approved, active, inactive or under orphans (in which case there is no student, faculty member relationship).
2. **Create New Appointment -** Faculty members send request for appointing student for GRA/GTA/HOURLY, dates of appointment. They can also check if such an appointment already exists for the student.
3. **Faculty Management –** Lists faculty members along with their personal information and indexes. The admin can create, edit and delete any faculty. Clicking on any particular faculty member shows information regarding that faculty/ shows per faculty view. The faculty can edit their information and request student appointments.
4. **Index Management –** Indexes can be managed by the admin, they can be closed or opened depending on the requirement. New indexes can also be added, modified or deleted.
5. **Job Applications Management –** The job management tool allows the EECS Faculty members to view the job applications submitted to the school by applicants applying for faculty positions within EECS. Clicking on any job lists the applicants under the job. Jobs can be created, edited and deleted by the admin.
6. **Student Management –** The student management tool allows management of EECS student information. Students can be created upon joining the school, edited or deleted.

2.2.2 Implementation

The application has been built using Object Oriented PHP and JavaScript. The modules are divided into:

i. **Front end layer** – holds the front end HTML/JavaScript code
ii. **Web Services** – holds the web service classes required for fetching data using REST (Representational State Transfer). The MIME type of the data being used is JSON with the set of operations such as GET, POST being supported.

iii. **Database layer** – The database layer consists of classes which actually connect to the database and perform (select, update) query operations.

2.2.3 Tools

2.2.3.1 Event Driven Programming using Yahoo User Interface Components

Yahoo User Interface Components commonly known as YUI is an open source JavaScript library used for building richly interactive Web Applications using Web 2.0 technologies such as AJAX, DHTML and DOM scripting. It features the YUI Core, Utilities, Controls, CSS resources, Developer Tools and Build Tools.

The YUI Core consists of

a. Yahoo Global Object
b. Dom Collection
c. Event Utility.

The Yahoo Global object provides a single global namespace which contains the language utilities, base infrastructure for the YUI. Hence it is needed in every page which utilizes the library. The Dom Collection simplifies the common DOM scripting tasks such as CSS management and positioning of HTML elements while providing support for cross browser inconsistencies. The Event Utility allows developers to create event driven applications by giving a simplified interface for subscribing the DOM events. Using these custom events allows you to publish them in your code so that other components can subscribe to these events and respond to them.

The Yahoo Utlities and Controls consists of a large number of YUI components. Some of them include Animation facility, Browser History Manager, Connection Manager, Cookie Utility, DataSource, Drag and Drop, ImageLoader, JSON Utility, Selector, Button, Calendar, DataTable, Container, Paginator, Menu, Tabview, Rich Text Editor etc.

The Developer Tools aim to assist the developers to log messages to an on-screen console such as FireBug using the Logger or using the YUI test framework to test browser based JavaScript solutions.

2.2.3.2 Components

In this project, for the EECS HR Application shown in Figure 2.2 the following major YUI components were applied.

2.2.3.2.1 TabView Control

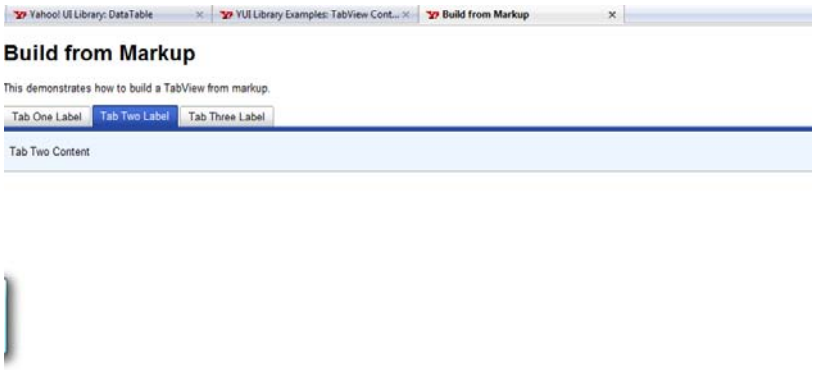


Figure 2.3 : YUI TabView Control Layout

The TabView Control applied in this project looks similar to the one shown in Figure 2.3. This control has been utilized as a layout feature for all of the applications shown in Figure 2.2

2.2.3.2.2 DataTable Control

The DataTable Control provides a convenient API which allows us to display screen-reader accessible tabular data on a web page. Some features of the DataTable applied in this project include:

a. Sortable columns

b. Pagination

c. Scrolling

d. Row selection

e. Resizeable columns

f. Inline cell editing.

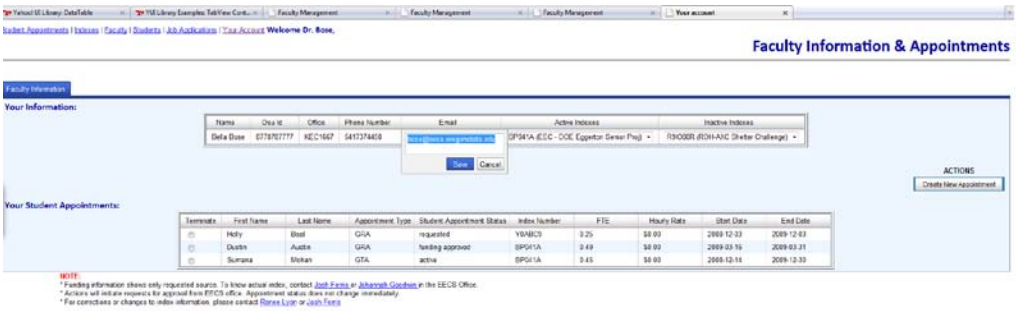


Figure 2.4 : DataTable Inline cell editing

In Figure 2.4, we can see the *'Email'* column of the DataTable can be edited and updated by simply editing the cell and clicking on *'Save'*. Updating inline allows us to improve the user experience as the user does not have to wait for the page to submit or load while the action takes place.

Most of the columns are also sortable. However in some special cases, (please see Figure 2.4) columns *'Active Indexes'* and *'Inactive Indexes'* are not sortable since they internally hold dropdown boxes.

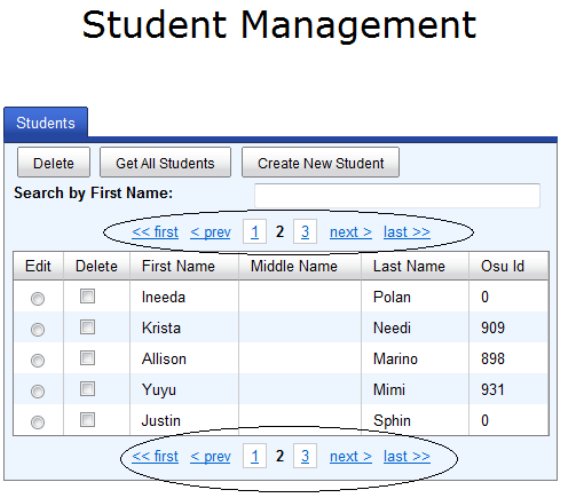


Figure 2.5 : DataTable Pagination

Figure 2.5 shows client side pagination with DataTable.

2.2.3.2.3 YUI AutoComplete

Using YUI AutoComplete feature with the DataTable allows the user to search for faculty members, students thereby updating the table in real-time. i.e. The AutoComplete queries the DataSource and presents the results back to the user. Figure 2.6 shows the AutoComplete Feature at work.

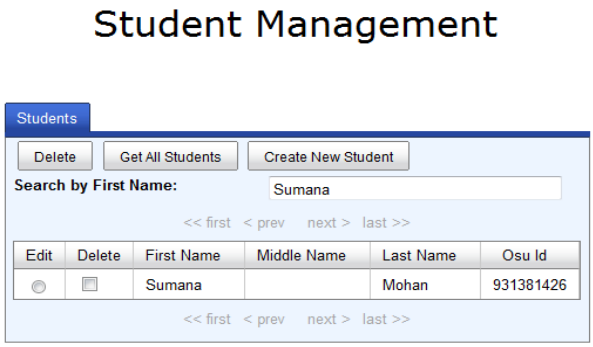


Figure 2.6 : AutoComplete listing search result

2.3 Fast Match Edit Script Algorithm

As explained earlier, in order to be able to store unique DOM states, a tree edit distance algorithm is needed. One such algorithm known as the FMES algorithm is used to describe the difference between two versions of hierarchical data. The FMES for two trees T1 and T2 is defined using node insert, node delete, node update and subtree move as the basic edit operations.
The algorithms presented below have been taken from the paper referenced in [2].

The algorithm has the following steps:

2.3.1 Finding a good matching

Find good matching's between trees T1 and T2. A good matching M is defined as all of the nodes in tree T1 which have corresponding matching nodes in T2. Any node which does not have such a match is not included in the matching M.

For ex: Consider a DOCUMENT_ELEMENT node x in T1, if there is a corresponding DOCUMENT_ELEMENT x node in T2 with the same set of values as in T1 (i.e say <x>some value</x>), then we can say we have a matching between them.

Figure 2.7 shows the *'Fast Match'* algorithm which is used for matching purposes

1. M ← null
2. For each leaf label l do
(a) S1 ← chainT1 (l).
(b) S2 ← chainT2 (l).
(c) Find LCS(S1, S2, equal).
(d) For each pair of nodes (x, y) belonging LCS, add (x, y) to M.
(e) For each unmatched node x belonging to S1, if there is an unmatched node y which belongs to S2 such that equal (x,y) then
A. Add (x, y) to M.

B. Mark x and y as "matched."

3. Repeat steps 2a - 2e for each internal node label l.

Figure 2.7 : Matching Algorithm

2.3.2 Generating the Edit Script

Find sequence of change operations (insert, delete, update, move) that transforms T1 to T1' where T1' is isomorphic to T2. It is isomorphic in the sense that T1' is similar to T2 except for the fact that they have different object identifiers. The sequence of change operations is referred to as an edit script.

Figure 2.8 : Edit Script Algorithm

1. E ← null, M ← M'
2. Visit the nodes of T2 in breadth-first order
/ combines the update, insert, align, and move phases */*
(a) Let x be the current node in the breadth first search of T2 and let y = p(x). Let z be the partner of y in M'.
(b) If x has no partner in M'
i. k ← FindPos(x)
ii. Append ins((w, a, v(x)), z, k) to E, for a new identifier w.
iii. Add (w,x) to M' and apply ins((w, a, v(x)), z, k) to T1.
(c) else if x is not the root / x has a partner in M' */*
i. Let w be the partner of x in M', and let v = p(w) in T1.
ii. If v(w) != v(x)
A. Append upd(w, v(x)) to E.
B. Apply upd(w, v(x)) to T1.
iii. If (y, v) does not belong to M'
A. Let z be the partner of y in M'.
B. k ← FindPos(x)
C. Append mov(w, z, k) to E.
D. Apply mov(w, z, k) to T1.
(d) AlignChildren(w, x)
3. Do a post-order traversal of T1. / the delete phase */*
(a) Let w be the current node in the post-order traversal of T1.
(b) If w has no partner in M' then append del(w) to E and apply del(w) to T1.
4. E is the edit script containing the change operations, M' is a total matching, and T1 is isomorphic to T2.

While generating the edit script, the tree T1 gets transformed to T2.

2.3.3 Align Children

The function AlignChildren takes into account that some children in the tree T1 maybe misaligned.

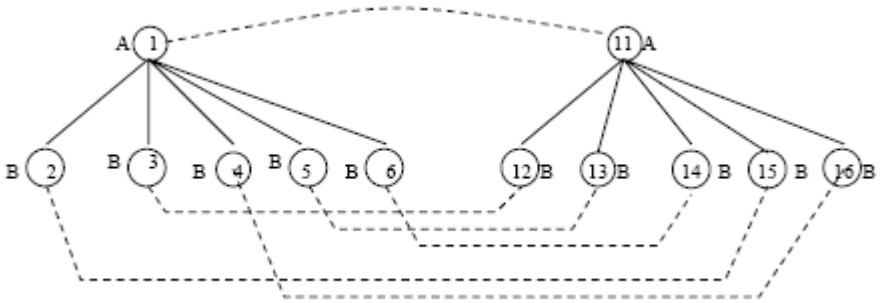


Figure 2.9 Trees depicting alignment of nodes

In the figure above, there are two ways of aligning the children of nodes 1 and 11. We can either move nodes 2 and 4 to the right of node 6 or move the nodes 3,5,6 to the left of node 2. Although both give us the same end result, it is more likely that moving fewer nodes is better. In order to ensure this case, the Longest Common Subsequence algorithm is used to find the shortest sequence of moves.

Figure 2.1.0 : Align Children Algorithm

Function AlignChildren(w, x)

1. Mark all children of w and all children of x as "out of order."

2. Let S1 be the sequence of children of w whose partners are children of x and

Let S2 be the sequence of children of x whose partners are children of w.

3. Define the function equal (a, b) to be true if and only if (a, b) belongs to M'.

4. Let S ← LCS (S1,S2,equal).

5. For each (a, b) belonging to S, mark nodes a and b as "in order."

6. For each a belonging to S1, b belonging to S2 such that (a, b) belongs to M but (a, b) does not belong to S

(a) k ← FindPos(b).

(b) Append mov(a,w, k) to E and apply mov(a,w, k) to T1.

(c) Mark a and b as "in order."

2.3.4 Find Position

The function FindPos is used to locate the position of the given node.

Figure 2.1.1 : Algorithm FindPos

1. Let y = p(x) in T2 and let w be the partner of x (x belongs to T1).

2. If x is the leftmost child of y that is marked "in order", return 1.

3. Find v belonging to T2 where v is the rightmost sibling of x that is to the left of x and is marked "in order."

4. Let u be the partner of v in T1.

5. Suppose u is the i^{th} child of its parent (counting from left to right) that is marked "in order." Return i + 1.

2.3.5 Example

To illustrate the FMES algorithm with an example: Suppose we have the following:

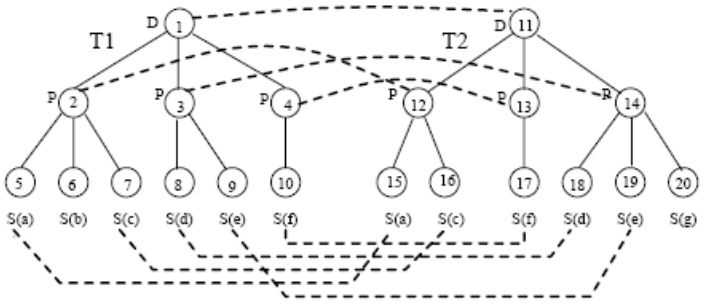


Figure 2.1.1 : Trees T1 and T2 depicting corresponding nodes with dashed lines

In Figure 2.1.1 we see two trees T1 and T2. Ignore the dashed lines at the moment. When running the Fast Match Edit Script algorithm against the two trees, T1 would be the tree the operations would be performed on to convert T1 to T1' where T1' is isomorphic to T2. The letters (D, P and S) denote Document, Paragraph, and Sentence respectively. In the case of XML documents, they could represent (<html><head><body>). The sequence of edit operations such as insert, update, subtree move and delete required to transform T1 to T2 is known as an *edit script*.

2.3.5.1 Running the Matching Algorithm

Before the edit script is generated, we need to run the matching algorithm to find nodes in T1 which correspond to nodes in T2. So for each leaf label, we need to find S1 such that S1 contains a chain of T1 nodes from left to right with the same label 'l'. i.e only those leaf nodes in our case marked with the label 'S' Similarly S2 contains the chain of T2 nodes from left to right with the label 'l'. The Longest Common Subsequence algorithm is then calculated with the 'equal' function providing the criteria for matching.

For leaf nodes: *equal(x, y)* is true if and only if *l(x) = l(y)* and *compare(v(x),v(y)) < =f,* where *'f'* is a parameter valued between 0 and 1. (Note: *x* and *y* are nodes in T1 and T2 respectively. *l(x)* denotes *label(x)* and *v(x)* denotes *value(x))* The compare function is used to compare the values of *x* and *y*. If their values are less than a certain parameter *'f'*, then they are considered to be equal. i.e. E.g.: The sentences *"Where is the city?"* and *"Where is the city located?"* are somewhat equal, in which case the compare function would return a value *<=f*.

For internal nodes: *equal (x, y)* is true if and only if *l(x) = l(y)* and *|common(x, y)|/max (|x|,|y|) > t,* where t > 0.5 is a parameter. The common function returns the child nodes that are common to both x and y and the max function returns the max of (number of child nodes of x , number of child nodes of y). So for internal nodes 2 in T1 and 12 in T2, common(x,y) would yield nodes (5,7) or in other words 2 nodes are

common between them, and max(x,y) would give us 3. Hence 2/3 = 0.6 > 0.5 and thus 2 and 12 are considered to be matching nodes.

The result got from LCS(x, y, equal) contains the pair of nodes (x,y) which match. After marking these nodes as 'matched', if any unmatched nodes are left, then for every unmatched node *x E S1*, if there is an unmatched node *y E S2* such that *equal (x, y)* returns true, then the pair is also added to the matching. The process is repeated for internal nodes 2,3,4 in T1 with nodes 12,13,14 in T2. The dashed lines in Figure 2.1.1 indicate matchings between corresponding nodes in T1 and T2.

2.3.5.2 Generating the Edit Script

An empty edit script is first created. Each node in tree T2 is visited in level order also known as Breadth First Order.

E.g.: Suppose we are currently at *node 12* in tree T2, *p(12) = 11*. (where *p* denotes *parent*). Since 12 has a partner in the matching M'(denoted by *w*), w = 2 and *v = p(2) = 1*.

Now, if v(2) != v(12), (i.e. value(2) != value(12)) then an update operation is performed, to update v(2) = v(12).

The values of the nodes 2 and 12 are not shown in the figure. The update operation is also added to the edit script. The children of 2 and 12 are subsequently aligned if necessary. The process is repeated for each node in tree T2. After traversing all of the nodes, a post order traversal of the tree T1 is performed to delete any nodes still unmatched and a delete operation is appended to the edit script.

Hence at the end of the algorithm, we have an edit script where M' is the total matching and T1 -> T1' which is isomorphic to T2.

If the edit script is empty, then we can confirm that the trees T1 and T2 do not differ.

2.4 Search Results

2.4.1 Search Results

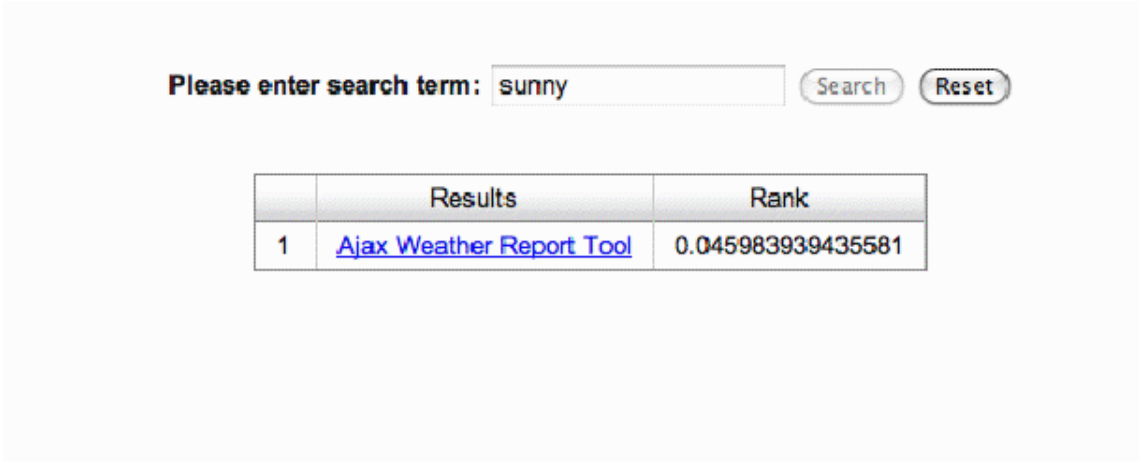


Figure 2.5 – Searching for keyword ‘sunny’

Figure 2.6 – IFrame displaying result after invoking JavaScript to reach state

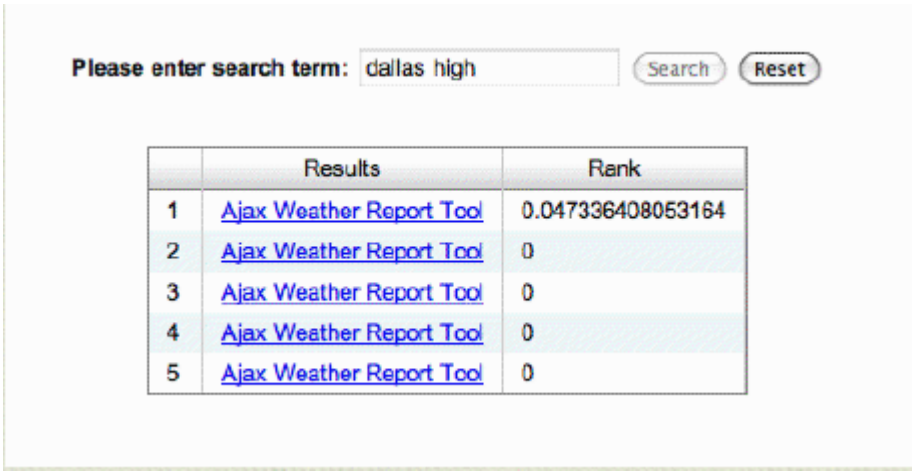


Figure 2.7 – Searching for term ‘dallas high’

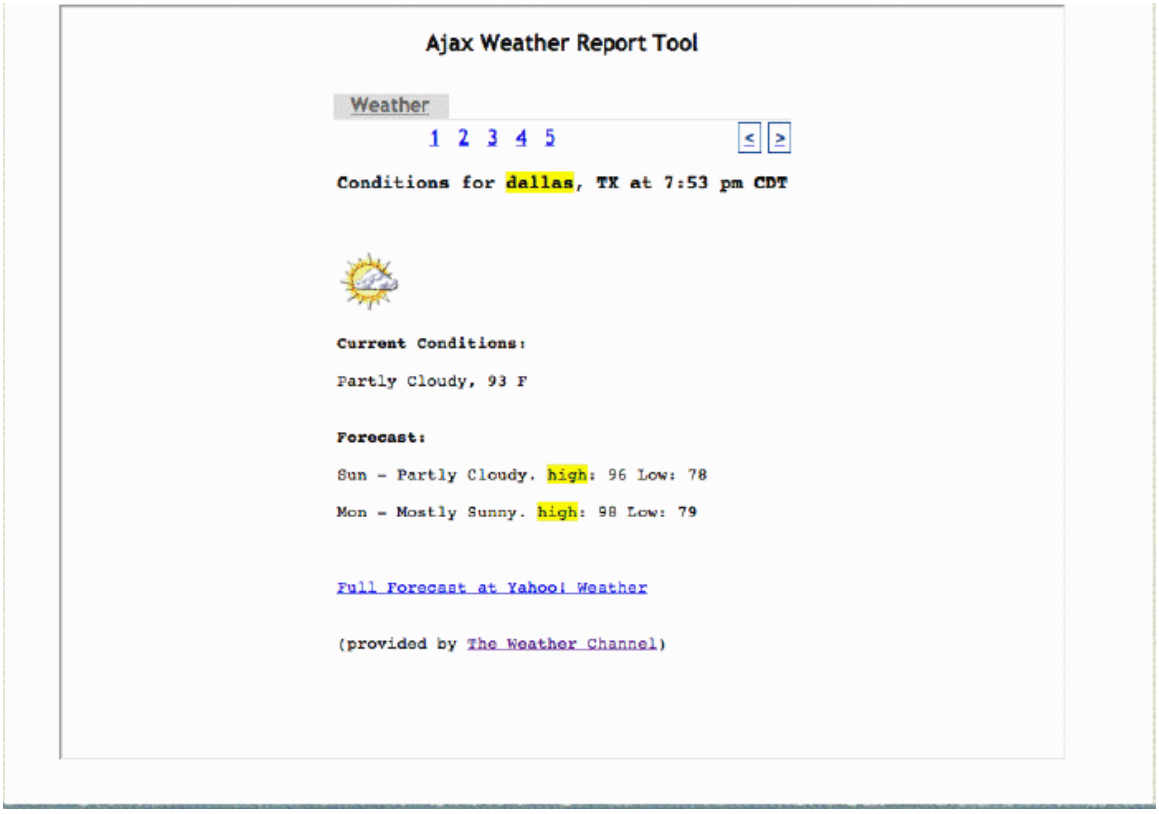


Figure 2.8 – IFrame displaying result for search term ‘dallas high’

2.5 Limitations

Due to the inherent nature of the AJAX pages the DOM states are updated immediately. While in some cases it could take a few seconds, there are also possible cases where the DOM might never get initialized completely due to network failure , server side issues or page unavailability.

Hence in some cases, the crawler was unable to successfully crawl all states within certain applications such as the other example I have built, namely the EECS HR application using HtmlUnit. Though the EECS HR application was built using standard JavaScript libraries from Yahoo YUI, the behavior of the application was not predictable and was inconsistent between pages and the various states within a page.

These are some of the drawbacks when it comes to indexing complex Ajax Web applications such as the EECS HR application. As developers use complex libraries today, such as (DOJO, YUI, GWT, JQuery, Prototype etc.) it would require much more complex crawlers and GUI less browser libraries to be built which can understand the underlying JavaScript code, execute and wait for the results successfully.

3 References

1. Sudarshan S. Chawathe , Anand Rajaraman , Hector Garcia-Molina , Jennifer Widom, *Change detection in hierarchically structured information,* ACM SIGMOD Record, v.25 n.2, p.493-504, June 1996

2. Ali Mesbah , Engin Bozdag , Arie van Deursen, *Crawling AJAX by Inferring User Interface State Changes,* Proceedings of the 2008 Eighth International Conference on Web Engineering, p.122-134, July 14-18, 2008

3. Cristian Duda , Gianni Frey , Donald Kossmann , Chong Zhou, AJAXSearch: *Crawling, Indexing and Searching web 2.0 applications,* Proceedings of the VLDB Endowment, v.1 n.2, August 2008

4. *Term-Frequency-Inverse Document Frequency*
http://en.wikipedia.org/wiki/tf-idf