

AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications *

Cristian Duda
ETH Zurich
Switzerland
dudac@inf.ethz.ch

Gianni Frey
ETH Zurich
Switzerland
freygi@student.ethz.ch

Donald Kossmann
ETH Zurich
Switzerland
donaldk@inf.ethz.ch

Chong Zhou[†]
Honghuan Univ. of Science
and Technology, China
zhouc@inf.ethz.ch

ABSTRACT

Current search engines such as Google and Yahoo! are prevalent for searching the Web. Search in dynamic pages, however, is either inexistent or far from perfect. AJAX and Rich Internet Application are such applications. They are increasingly frequent on the Web (in YouTube, Amazon, GMail, Yahoo!Mail) or mobile devices and are offering a high degree of interactivity to the user, by seamlessly loading content from the server without the need to refresh the page. Current search engines cannot correctly index AJAX applications. This produces false positives and false negatives, because search engines do not understand the application logic that loads content dynamically. Crawling an AJAX application is a difficult problem. Since the user invokes events on the page, crawling must identify the different *application states* generated by the client-side logic. This demo sets the stage for this new type of search and shows that a search engine for AJAX can be built. Among others, the challenges, as opposed to traditional search engines, are: automatically identifying states by triggering events, efficiently crawling application states, avoiding the invocation of potentially very numerous events, scalability in the number of events, duplicate elimination of states, result presentation and aggregation, ranking. The demo presents the AJAX search engine: crawler, indexer and query processor, applied on a real application and showcases challenges and solutions.

1. INTRODUCTION

Currently, Google and other search engines are the usual way to search the World Wide Web. A big part of the Web pages can be indexed and retrieved with good quality. However, the Web is changing. More and more applications are dynamic by nature and include a lot of client-side and client-server interactivity: Javascript applications, AJAX applications, Rich Internet Applications are al-

*The work presented in this paper was partly supported by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss National Science Foundation - grant number 5005-67322.

[†]Work performed at ETH Zurich, Switzerland, supported by China Scholarship Council.

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.

Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212)869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

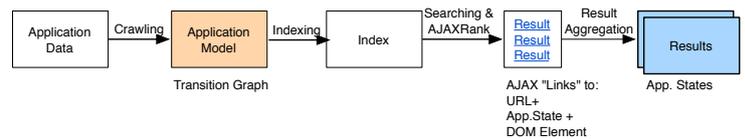


Figure 1: Architecture of AJAXSearch.

ready handling much of the data on the web and on portable devices, providing a high degree of interactivity to the user.

Current search engines fail to index these applications correctly. AJAX Applications run partly on the client, embedding a lot of functionality in a single page, under the same URL. Current search engines do not index these pages since they do not understand the application logic: the application has *states*, and events cause *transitions* between states. Furthermore, all states are identified by the single URL of the page, and this is incompatible with the traditional search model of the web. Current search engines will produce false positives by considering all applications as a single page, or false negatives by ignoring the parts exposed only through client-side scripting. Currently, search in AJAX applications is done by custom search engines developed by the application provider or by exposing the data to the traditional search engines, based on agreements. If search is possible, it is hard-coded and expensive to implement. Small providers cannot afford this luxury.

We address this problem: we implement AJAXSearch: an AJAX-aware search engine. Just as a traditional search engine, it contains a crawler, indexer and query processor, as shown in Figure 1, but the components are adapted to AJAX. Our first implementation focuses on AJAX sites without user input, thus avoiding the already studied domain of the "hidden web".

Technical challenges: The challenges to address when building an AJAX search engine are:

- (i) generating application states by invoking numerous events.
- (ii) identifying duplicate states.
- (iii) maintaining context information; reconstructing application states.
- (iv) ranking results.
- (v) state explosion in case of too numerous events.

The demo shows each component of the AJAX search architecture. The main aspects are highlighted for two applications: a news reader and Yahoo!Mail, a well-known AJAX application.

2. ARCHITECTURE

As mentioned in Section 1, an enhanced search engine capable of reading AJAX applications has the architecture seen in Figure 1.

The AJAX Crawler has the role of identifying application states. First, events are invoked automatically on the initial page. Events are extracted from each page, in code such as *onclick=...*, the JavaScript code is invoked, and a new state is subsequently created.

Crawling Model. The result of crawling is a special **application model**. This is an annotated automaton, containing all states and transitions, as shown in Figure 2. Each node is an application state and each edge is a transition. The annotations mark the source (i.e., the element that triggers the event, e.g., “<div id=1>”), the event (e.g., ‘dblclick’), the target(s) (i.e., all affected elements) and action (i.e., the type of change, e.g., target.innerHTML=newcontent).

Crawling Challenges. The crawler uses a breadth-first approach. The first challenges are completeness and closeness of the application model. Usually, there is a limited number of different events and different states that can be invoked. Otherwise, a maximum depth limit can be set. The second challenge is to decide if two generated states are identical. Currently, we create a new state whenever a transition generates a new DOM tree and we compare the content with that of existing states in order to detect duplicates. The next challenge is maintaining context information. During crawling, all Javascript variables are maintained as part of the state. The last challenge is crawling personalized content (such as Yahoo!Mail). We require here minimum intervention from the user (user and password) in order to access her content. This scenario is realistic for enterprise-only crawling or user-specific crawling.

Crawling Optimizations. We are currently exploring more optimizations, in particular, caching, avoiding states that do not affect caching, and more similarity measures. The *Crawling* scenario of the demo reflects this component and allows to inspect the DOM tree, the variables, and to invoke events on the current state.

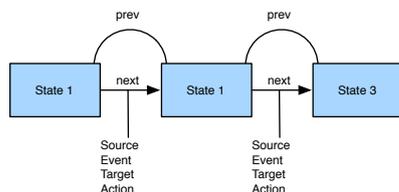


Figure 2: Application Model of an AJAX application.

The **Indexer** reads the application model and builds an enhanced inverted file with the AJAX-specific information on the state and element containing each keyword. The Query Processor uses the index to process keyword queries. Results are returned in the granularity of states: i.e., URL, state, and element containing the desired keywords. We will use the extended element information also for ranking purposes. The *Indexing* scenario showcases this.

The **Query Processor**. The index that was previously built can be queried. Keyword queries can be processed, and they return *application states*. In particular, we also return the element (i.e., the part of the page) where the keyword occurred. The *AJAXSearch* scenario of the demo reflects the search component and how search actually delivers better results than Google does. Furthermore, results are ranked according to their participation in the application and to the importance that the user gives to this state.

Result Aggregation. An additional challenge for search is how to present results to the user, especially if they are too granular. Another one is how to reconstruct the application state, including context information. Therefore, several result presentation strategies are explored, in particular grouping similar results based on the occurrence of keywords in multiple states, in the same element. The demo presents this in the *Result Aggregation* scenario.

Personalization. One can argue that an automatic approach can lead to indexing a too large number of states, some of which might be irrelevant. In particular, independent events may lead to a large number of states with many similarities. We made a first step towards defining rules, in the idea of the *robots.txt* provided by web site developers for guiding search engines. The demo shows how

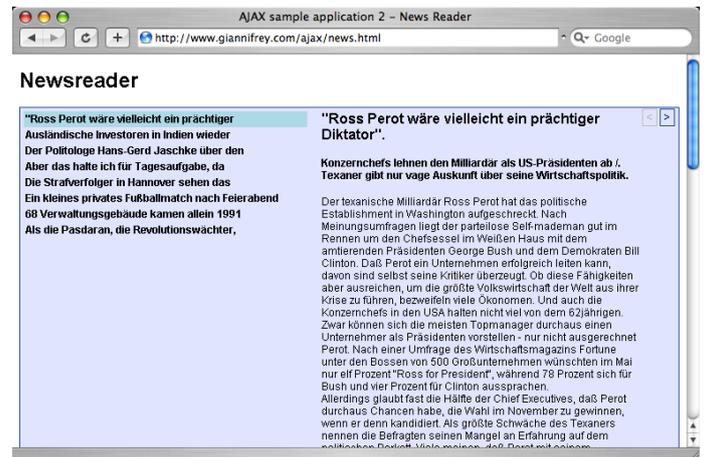


Figure 3: AJAXNews: What does a traditional engine see?

the number of indexed states can decrease when application logic is specified declaratively, in the *Personalization* scenario.

3. DEMONSTRATION

We crawl, index and search AJAX Applications in order to show the advantage of a full AJAX Indexing approach.

3.1 The AJAX Applications

In our demo we use two applications described below. **Application 1: AJAXNews.** The news reader AJAXNews is a custom-built application residing at [3]. It has several pieces of news loaded from the server and changed through two buttons (next and previous) or through a side-menu as shown in Figure 3. This application can be used for testing (since it can be customized), and for showing the quality of ranking. Furthermore it illustrates how grouping strategies for the result can be used in order to avoid generating a large number of superfluous states.

Application 2: Yahoo! Mail. We chose Yahoo! Mail [12] as the first test on a real AJAX Application. Yahoo! Mail is an E-mail client. Since it uses text, it is easier to approach than, for example, Google Maps. The E-mail client also displays the calendar of the user and news along with the messages, using AJAX. Yahoo! Mail has a custom search engine but only on messages. Crawling this application is, however, cumbersome because of frequent application changes, and because of heavy use of Javascript, requiring us to extend the openly available [9] framework that we used for implementation. Yahoo! Mail will be used in order to show the behavior of an AJAXSearch on a complex application. The possibility for the application developer to enhance search result quality using personalized rules will also be demonstrated.

3.2 Demo Scenarios

We showcase AJAXSearch on the two AJAX applications.

Scenario 1: Crawling.

We crawl the AJAXNews Application. During crawling, the user can visualize how the different states are retrieved. This is shown in Figure 4. On the left, the DOM tree of the current application state can be inspected. On top, the individual Javascript events can also be invoked manually by the user to generate states. The Javascript variables can also be seen at each state. At the end, the Transition Graph is displayed in a textual format and can be stored on disk for later retrieval. During crawling, similar states are detecting using a hash on the content of the DOM. An additional point is that

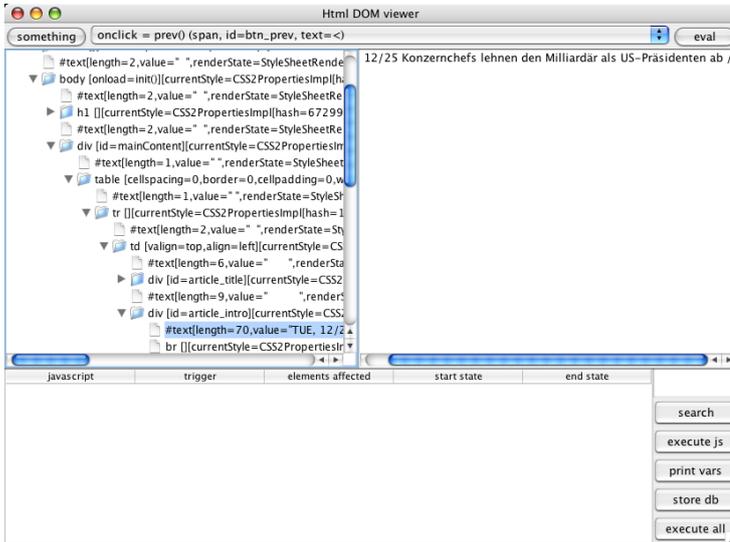


Figure 4: Crawling an AJAX Application. Displaying the DOM.

the application model can grow indefinitely if events can always be invoked. However, the AJAXNews application always ends. Otherwise, a maximum limit on the number of states can be set.

Scenario 2: Indexing.

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

Scenario 3: AJAXSearch.

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

In this scenario, the user can see how results are returned in terms of tuples $\langle URL, state, element \rangle$ and not just in terms of $\langle URL \rangle$ as in traditional search. If two keywords occur in the same state but in different elements, the Least Common Ancestor (LCA) of the two elements is used. The application in action is shown in Figure 5. The user has several options:

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

The first two steps are also common in traditional web search. However, the last one is specific to AJAXSearch. An AJAX application has a context defined by Javascript variables. Our framework is able to recreate the state by traversing the application model built during the crawling phase, as shown in Figure 2. However, the time needed to fetch the result increases for each of the above options. Reconstituting the application state is the most expensive option, but the advantage for the user is that she can continue the navigation from that point.

Scenario 4: Traditional Search vs. AJAXSearch.

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

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

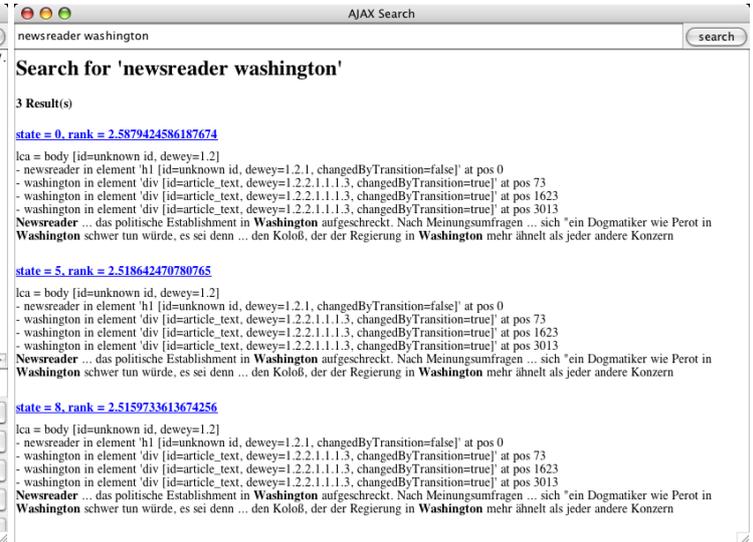


Figure 5: AJAXSearch in action. Results are links to application states.

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

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

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

Scenario 5: Result Aggregation.

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

No group. This is the default option. All occurrences of $\langle URL, state, element, position \rangle$ are returned.

Group by state. More occurrences of the keywords can appear in the same state, in different parts of the page structure. Returning the elements is relevant because some parts may be more or less important in the applications. For usability however, they can be aggregated and only the state is returned.

Group by location. In case of occurrences in many states but in the same visual location, it is advisable to return an aggregated result based on this common location. For example, the news header appears in all states. Returning all states may be daunting for the user. This allows the user to see which part of the page contains most occurrences.

Scenario 6: State Explosion.

We show an example when crawling states automatically leads to an explosion in the number of states.

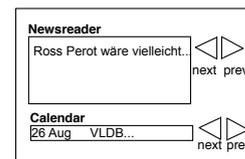


Figure 6: Application with two sets of independent events.

In applications such as Yahoo!Mail, several independent events may be invoked, such as those for displaying E-mails and those

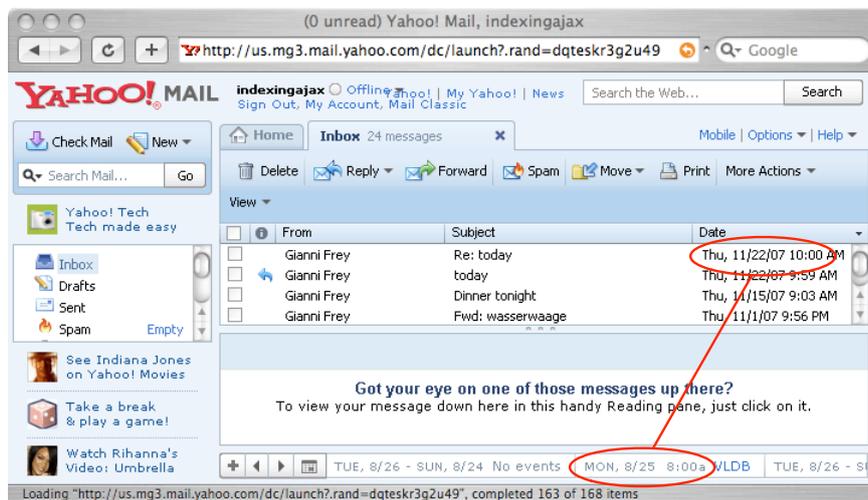


Figure 7: Personalization: Correlations in the application model increase search precision and crawling performance.

for showing the user's calendar. Both are displayed using independent next/previous events. We can also illustrate this on the AJAX news application shown in Figure 6, enhanced with calendar events (application at [2]). Yahoo! Mail is shown in Figure 7. The total number of states is then a cartesian product, as in Figure 8.

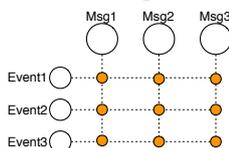


Figure 8: Independent events cause a state explosion.

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

Scenario 7: Personalization.

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

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

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

specifies that the date of the E-mail message is the same as the date of the event in the calendar. The effect in functionality is that the user can search for messages on a given date (e.g., the messages during "VLDB"). This is also shown in Figure 7. In this scenario, the user can crawl, index and search the extended AJAXNews application and/or Yahoo!Mail. The index will be smaller and only the matching states are returned at query processing time.

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

4. RELATED WORK

Searching AJAX applications has not been studied on its own until now. There are however, several connections with literature

from the DB and IR communities. Transitions and states are a part of movie retrieval [6]. Searching and Ranking based on structural properties are addressed by XRank [10] or [5] in the context of XML. Searching and grouping structural results are especially relevant for XQuery [8] and XQuery Full-Text [4] but not included in the final recommendation. [11], [1], [7] address indexing databases for keyword search, based on given queries or constraints (joins).

5. REFERENCES

- [1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: Enabling Keyword Search over Relational Databases. In *SIGMOD*, 2002.
- [2] AJAX News with calendar. <http://www.giannifrey.com/ajax/news.cfm?showCalendar=true>.
- [3] Sample AJAX News Application. <http://www.giannifrey.com/ajax/news.html>.
- [4] S. Amer-Yahia, C. Botev, S. Buxon, P. Case, J. Doerre, D. McBeath, M. Rys, and J. Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text, W3C Working Draft, 4 April 2005. <http://www.w3.org/TR/2005/WD-xquery-full-text-20050404/>.
- [5] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient XML search with complex full-text predicates. In *SIGMOD*, 2006.
- [6] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.
- [7] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.
- [8] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language W3C Candidate Recommendation, 3 November 2005. <http://www.w3.org/TR/2005/CR-xquery-200511033>.
- [9] COBRA Toolkit. <http://html.xamjwg.org/cobra.jsp>.
- [10] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRank: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.
- [11] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, 2002.
- [12] Yahoo! Mail. <http://mail.yahoo.com>.