AJAX Crawl: Making AJAX Applications Searchable

Cristian Duda #1, Gianni Frey #2, Donald Kossmann #3, Reto Matter #4, Chong Zhou *5

#*ETH Zurich, Switzerland*
[1]cristian.duda@inf.ethz.ch
[2]freygi@student.ethz.ch
[3]donald.kossmann@inf.ethz.ch
[4]reto.matter@student.ethz.ch

**Huazhong University of Science and Technology*
[5]chong.zhou@inf.ethz.ch

***Abstract*— Current search engines such as Google and Yahoo! are prevalent for searching the Web. Search on dynamic client-side Web pages is, however, either inexistent or far from perfect, and not addressed by existing work, for example on Deep Web. This is a real impediment since AJAX and Rich Internet Applications are already very common in the Web. AJAX applications are composed of *states* which can be seen by the user, but not by the search engine, and changed by the user using client-side *events*. Current search engines either ignore AJAX applications or produce false negatives. The reason is that crawling client-side code is a difficult problem that cannot be solved naïvely by invoking user events. The challenges are: lack of caching, duplicate states detection, very granular events, reducing the number of AJAX calls and infinite event invocation. This paper sets the stage for this new search challenge and proposes a solution: it shows how an AJAX Web application can be crawled in the granularity of the application states. A model of AJAX Web sites is presented. An AJAX Crawler and optimizations for caching and duplicate elimination are defined, and finally, the gain in search result quality and corresponding performance price are evaluated on YouTube, a real AJAX application.**

I. INTRODUCTION

Google Mail[14], Yahoo! Mail[29], Google Maps[15] are well-known AJAX applications. Their goal is to enhance the user experience by running client code in the browser instead of always refreshing the Web page, and minimizing server traffic. Traditional Web Applications such as Amazon and YouTube! also start including AJAX Content in traditional pages in order to offer higher interactivity to the user. An AJAX Application is a dynamic Web application at a given URL, usually based on Javascript, which presents to the user different states changed by the user through UI events. Taken to the extreme, an AJAX application can present all the states of the application seamlessly, without changing the URL. This causes a mismatch with the current search model of Google-like search engines, where the presented results are uniquely identified by a URL.

Current search engines ignore AJAX content since crawling AJAX is a difficult problem, due to the following challenges: *(i) No Caching/pre-crawl*. Current search engines pre-cache the Web and crawl locally. Events, however cannot be cached. *(ii) Duplicate states*. Several events can lead to the same state,

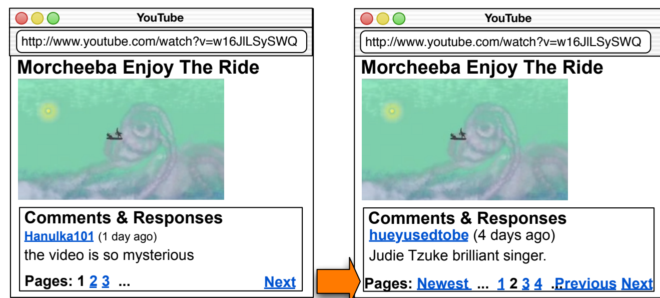


Fig. 1. YouTube: Comments load using AJAX.

because the same underlying Javascript function is used to render the content. This is crucial in optimizing and reducing the size of the application view.
(iii) Very granular events. This can lead to a large set of very similar states.
(iv) Infinite event invocation. Since AJAX applications consist of events, the application view may never lead to a closure. We will address these issues as we will show further.

Because of these challenges, current search engines do not crawl AJAX and provide the workarounds that are showed below. However, they are not generic, and second, not accurate enough or may require the collaboration of the application provider:
Hand-Coded Web Pages. Special Web pages can be set up in order to include an alternate view of the dynamic content. This page is hand-coded, less rich in content than the initial page and causes therefore a loss of information. Currently, Google finds this page and indexes it. Our goal is to avoid hand-coded solutions and be generic.
Custom Search Engines. Applications such as YouTube provide their own search engine. This engine does not usually have access to all dynamic content, therefore it is limited. Furthermore, implementing custom search engines is something small application providers cannot afford.
Exposing Data to Search Engines. Bigger Web Sites can agree to give search engines direct access to the data with

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.90

IEEE computer society

generic credentials. This provides better accuracy, but may prove too coarse-grained since it might provide no specific data. We avoid this and we propose a generic solution for AJAX Crawl.

All approaches of the state-of-the-art are however insufficiently precise, as shown in the following example:

A. Motivating Example: YouTube Comments

YouTube[30] is one Web site which has changed from a traditional, Web-page-based interface, to one which includes AJAX parts. This is a trend also followed by Amazon, for example, which dynamically includes excerpts from books or suggesting related products.

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

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

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

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

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

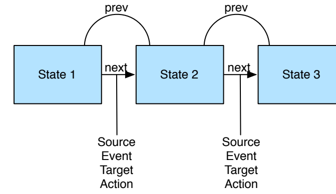


Fig. 2. Model of an AJAX Web page.

.

contributions:

- **Modeling AJAX Sites.** We propose a model of an AJAX Web Site. We address text-based AJAX Applications without user input (i.e., no forms).
- **AJAX Crawler.** We propose an AJAX Crawler which crawls based on user events. We provide an optimization to the problems of caching and duplicate elimination of states.
- **Evaluating the gain in result quality.** We evaluate the improved recall of the AJAX search over traditional search. Results are obtained on a YouTube subset.
- **Evaluating performance tradeoff.** We evaluate the performance price payed for the improved search results on the YouTube subset and custom AJAX Application.

The rest of the paper is organized as follows: Section II models AJAX Web Sites, pages and events. Sections III and Section IV are the main contribution of this paper and describe the crawling algorithm and a solution to the problem of detecting duplicates and caching in AJAX crawling. Section V describes the overall architecture of a search engine, including parallelization. Experimental results are shown in Section VI, Section VII discusses the inclusion of AJAX Search in current search engines, while Sections VIII and Section IX contain conclusions and future work.

II. Modeling AJAX

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

A. Event Model

When Javascript is used, the application reacts to user events: *click*, *doubleClick*, *mouseover*, etc. An example of such an event is displayed in Figure 3. The $onClick$ *event* triggered on the $\langle div\ id\ =\ "next" \rangle$ HTML element *source*, applied to the $recent_comments$ element (*target*). The content of the targets changes using the action: $recent_comments.innerHTML$=.... We will use these notations throughout this work.

Fig. 3. Event Structure in Javascript.

B. AJAX Page Model

Section I presented an AJAX application as not only a simple page identified by an URL, but also as a series of states, events and transitions. This is the main difference between traditional search and AJAX Search and we model it correspondingly. The AJAX Page Model is a view of all states in a page (e.g., all comment pages). In particular it is an automaton, a **Transition Graph**. The Transition Graph contains all application entities (states, events, transitions) as annotations. It is defined by:

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

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

C. Modeling AJAX Web sites

As opposed to traditional Web, an AJAX Web site contains both static and dynamic content. Furthermore, each page contains hyperlinks to other web pages as shown in Figure 4.

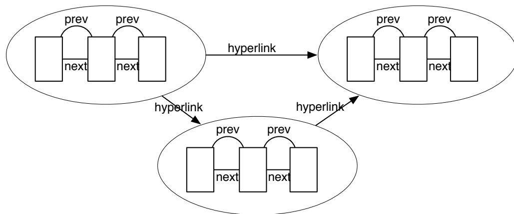


Fig. 4. Model of an AJAX Web Site: AJAX Pages, hyperlinks and AJAX states.

The difference to the traditional Web is that the user may trigger events in the same page (such as *next* and *prev*) which generate new application states. In order to complete the comparison, the transitions caused by the events may be called AJAX links.

As opposed to this, traditional Web Sites are characterized just by a graph of *pages*, connected by hyperlinks. Crawling the AJAX part of a Web site leads to an increase in search quality, but also to an overhead that needs to be addressed in terms of performance.

III. AJAX Crawling

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

A. Crawling Algorithm

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

We detail the crawling algorithm for AJAX applications in Algorithm III.1.

Algorithm III.1 Breadth-First AJAX Crawling Algorithm

```
1: Function init(url)
2: dom = readDocument(url)
3: dom.executeFunction(body.onLoad) {AJAX Specific}
4: appModel.add(dom) {Add first state to the App. Model}
5: crawl(dom)
6: end Function
7: Function crawl(State s)
8: for all Event e ∈ s do
9:    dom.executeFunction(e.function)
10:   if dom.hasChanged() then
11:      State newState = new State(dom)
12:      if appModel.contains(newState) then
13:         newState = appMode.get(newState)
14:      end if
15:      Transition t = new Transition(e.source, e.trigger, e.action*,
         e.modif*)
16:      appModel.add(t, s, newState)
17:      appModel.rollback(t)
18:   end if
19: end for
20: for all Transition t ∈ (s, s1) do
21:    Crawl s1 {Breadth-first traversal of reachable states}
22: end for
23: end Function
```

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

Event	Start State	End State	Source	Event	Target	Action
e1	s1	s2	"next"	onClick	recent_comments	innerHTML
e1	s1	s2	"page 2"	onClick	recent_comments	innerHTML
e2	s2	s3	"next"	onClick	recent_comments	innerHTML
e2	s2	s3	"page 3"	onClick	recent_comments	innerHTML
e3	s3	s2	"prev"	onClick	recent_comments	innerHTML
e3	s3	s2	"page 2"	onClick	recent_comments	innerHTML
e4	s2	s1	"prev"	onClick	recent_comments	innerHTML
e4	s2	s1	"page 1"	onClick	recent_comments	innerHTML

TABLE I
ANNOTATION FOR THE TRANSITION GRAPH OF AN AJAX APPLICATION.

triggered, the immediately adjacent states in the application graph can be crawled.

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

B. Addressing Crawling Challenges.

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

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

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

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

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

Number of AJAX Calls. We identified that the biggest challenge in the performance of the AJAX Crawler is the high

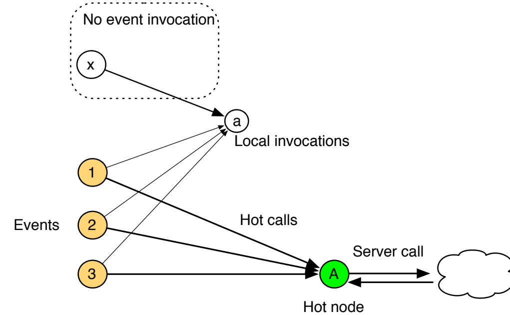


Fig. 5. Javascript Invocation Graph.

amount of calls to the server, especially since this can lead to the same state. We developed a special technique for solving this problem, mentioned below.

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

IV. A HEURISTIC CRAWLING POLICY FOR AJAX APPLICATIONS

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

This section addresses this problem and uses the following observation. In the crawled AJAX applications, the structure of the application is usually stable, and contains for example a menu, present in all states, and a dynamic part. The YouTube Page for example contains for each page several links to the next page or direct jump to one of the immediately consecutive pages. This leads to a frequent situation: using the menu items reload the same content from the server, and leads to a state that has already been reached. A crawler can identify that the same state has been reached just when it compares the two states, after the content has again been retrieved from the server. We deal with this problem by identifying the same state but *without* fetching the content.

Event #	Functionality	Event Type
1	“init page”	*onload*
2	“next page” (from page 1)	*onclick*
3	“prev page” (from page 2)	*onclick*
4	“jump to page 2”	*onclick*
5	“jump to page 3”	*onclick*
6	...	*onmousedown*
7	...	*onmouseover*
...	...	...

TABLE II

EVENTS AND FUNCTIONALITIES IN THE JAVASCRIPT INVOCATION GRAPH.

ID	Function
a	$showLoading(ID)$
b	$togglePanel(ID)$
c	$urchinTracker(ID)$
d	...
A	$getURLXMLResponseAndFillDiv(URL, ID)$

TABLE III

FUNCTIONS IN THE JAVASCRIPT INVOCATION GRAPH ON YOUTUBE PAGE.

A. Javascript Invocation Graph.

The heuristic we use is based on the runtime analysis of the Javascript invocation graph. This structure contains a node for each Javascript function in the program and its dependencies (i.e., invoked functions). The invocation Graph for the YouTube Web site is depicted in Figure 5.

The nodes in the Javascript invocation graph are Javascript functions. The functionality of an AJAX page is expressed through events. In case of YouTube, these are *next*, *prev*, *jump to page*, as shown in Table II. Functions in the Javascript code can be invoked either directly by event triggers (event invocations) or indirectly by other functions (local invocations). Table III lists some functions appearing in the YouTube page. In order to fulfill their functionality in an AJAX application the events call, directly or indirectly, other functions which fetch content from the server. The dependencies in the code are listed in Figure IV. We call the functions that fetch content from the server **Hot Nodes** and a call to a hot node **Hot Call**. A single function fetches content from the server, i.e., $getURLXMLResponseAndFillDiv$. In AJAX, the same function can be invoked in order to fetch the same content from the server from different comment pages, as shown in Figure IV. For example, both events “Next page (from page 1)” and “Jump to Page 2” lead to the same server invocation, for page 2. In this approach we detect this situation and we avoid invoking the same function twice, as shown below.

B. Optimized Crawling Algorithm.

We solve the problem of caching in AJAX applications and detecting duplicate states by identifying and reusing the result of server calls. Just as in traditional I/O analysis in databases, we tend to minimize the number of the most expensive operations, i.e., the *Hot Calls*, invocations which generate AJAX calls to the server. The new Crawler with heuristics can be summarized in Algorithm IV.1. The main points of the algorithm are:

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

Event #	Functionality	Function
1	“init page”	$init()$
2	“next page” (from page 1)	$showLoading('recent_comments')$
2	“next page” (from page 1)	$getURLXMLResponseAndFillDiv($ $\ldots' action_get_comments = 1\&p = 2' \ldots)$
2	“next page” (from page 1)	$urchinTracker(...)$
3	“prev page” (from page 2)	$showLoading('recent_comments')$
3	“prev page” (from page 2)	$getURLXMLResponseAndFillDiv($ $\ldots' action_get_comments = 1\&p = 1' \ldots)$
3	“prev page” (from page 2)	$urchinTracker(...)$
4	“jump to page 2”	$showLoading('recent_comments')$
4	“jump to page 2”	$getURLXMLResponseAndFillDiv($ $\ldots' action_get_comments = 1\&p = 2' \ldots)$
4	“jump to page 2”	$urchinTracker(...)$
...	...	

TABLE IV

FUNCTIONS CORRESPONDING TO EVENTS IN THE JAVASCRIPT INVOCATION GRAPH ON YOUTUBE PAGE.

Hot Node	Parameters	content
A	P_1	...
A	P_2	...
B	P_1	...

TABLE V

THE HOT NODE CACHE

Step 2: Building Hot Node Cache. The crawler builds a table containing all *Hot Node* invocations, the actual parameters used in the call and the results returned by the server (line 34-53). This step uses the current runtime stack trace.

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

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

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

C. Complexity Analysis

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

Algorithm IV.1 Breadth-first Heuristic AJAX Crawling Algorithm

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

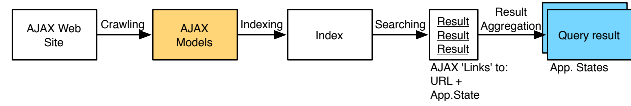


Fig. 6. Architecture of an AJAX Search Engine.

D. Simplifying Assumptions

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

- *Snapshot Isolation*: we assume that an application does not change during crawling. This is realistic since crawling must anyway be done regularly, but not continuously, even for the most updated sites. In YouTube, it is not relevant to crawl comments every second.
- *No Forms*: a lot of AJAX applications (such as Google Suggest [16]) use forms to infer actions from the user, dynamically. We do not deal with AJAX parts that require user inputting data in forms.
- *No update events.* We explicitly avoid triggering update events, such as Delete buttons. In case of crawling an authenticated user's Yahoo! Mail or GMail (well-known) AJAX clients, this could mean deleting E-mails from the user's Inbox.
- *State explosion.* Google Maps for example has an infinite amount of states (i.e., as many as there are pixels on the map). Still, an automated crawling is viable, by limiting the amount of automatically indexed states, and this is also the approach that we take. We predict that in the future, AJAX Web Sites will provide a *robots.txt* file with information on the possible granularity of search on their pages.
- *No Image-based retrieval.* The states in applications such as Google Maps [15] are not text-based, but image-based. We limit ourselves to text-based retrieval.

V. AN AJAX SEARCH ENGINE

The purpose of this paper is to crawl AJAX content with the goal of improved search results. In order to be able to provide this functionality, the crawler is integrated in a complete architecture similar to that of a traditional search engine, as shown in Figure 6. The components are described below.

A. AJAX Crawler

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

B. Indexing

Indexing is an operation which starts from the model of the AJAX Site and builds the physical inverted file. In traditional information retrieval, an index is an inverted file[2] containing information about the documents in which the keywords

occur. The result of the Indexing will be used during query processing, in order to return results. As opposed to traditional index processing, in our case a result is an URI and a state. As an example, the inverted file for the YouTube Application is presented in Table VI. The enhanced inverted file contains a link to the Web page containing the words (in this case, the URLs are two videos of Morcheeba), and to the state (i.e., the comment page) containing the word. The score is computed based on the number of occurrences of the word in the state.

Word	URI	State	Score
morcheeba	www.youtube.com/watch?v=w16JlLSySWQ	s1	1
morcheeba	www.youtube.com/watch?v=w16JlLSySWQ	s2	1
morcheeba	*www.youtube.com/watch?v=Iv5JXxME0js*	s1	2
mysterious	www.youtube.com/watch?v=w16JlLSySWQ	s1	1
singer	www.youtube.com/watch?v=w16JlLSySWQ	s2	1
...	...	...	...

TABLE VI

INVERTED FILE FOR YOUTUBE AJAX WEB SITE.

C. Query Processing

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

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

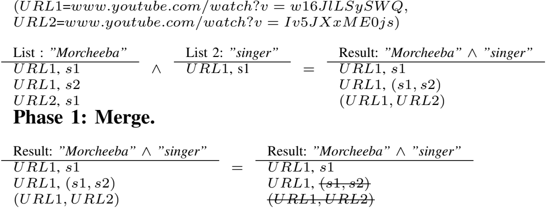
Query	URI	State	Rank
morcheeba	*www.youtube.com/watch?v=Iv5JXxME0js*	s1	0.9
	www.youtube.com/watch?v=w16JlLSySWQ	s1	0.7
	www.youtube.com/watch?v=w16JlLSySWQ	s2	0.7
	...	...	...

TABLE VII

QUERY PROCESSING ON THE AJAX NEWS APPLICATION.

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

As an example, shown in Figure 7 let's take the query *Q3* from the Introduction: *Morcheeba singer*. This will result in the conjunction between two posting lists. The posting lists of keyword *Morcheeba* (already shown in Table VII) and of keyword *singer* are presented and merged in the first row of Figure 7. The second row indicates the second phase of Processing Conjunctions, and shows how incompatible URIs, as well as incompatible states under the same URI are eliminated from the result. The result of *Q3* is the tuple $\langle URL1, s2\rangle$, corresponding to the second page of comments of the video presented in Section I.



Phase 2: Eliminate Incompatibilities.

Fig. 7. Processing Conjunctions.

3) Ranking: This work focused on boolean retrieval, with the purpose of increasing Recall. Therefore, ranking is not the main focus of this work. We mention briefly that we implemented a ranking algorithm based on both traditional ranking mechanisms (i.e., *tf/idf*), positional information, and application based information (i.e., distance in the transition graph). More information can be found in [12].

D. Result Aggregation

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

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

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

E. Parallelization

Because crawling AJAX faces the difficulty of not being able to really cache dynamic Web content, except for the heuristics discussed in Section IV, network connections must continuously be created. This drastically increases the crawling time, as we will also show in Section VI. The situation can however be improved through parallelization. Figure 8 shows the parallel architecture of the AJAX Search Engine.

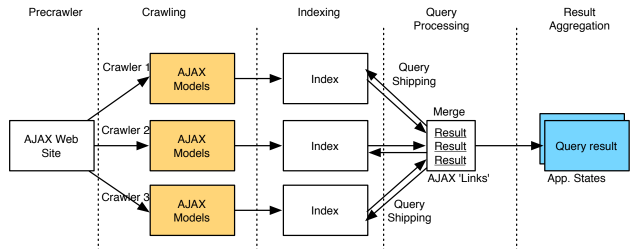


Fig. 8. Parallel AJAX Search architecture.

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

VI. Experimental Results

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

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

A. Experimental setup

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

1) YouTube Datasets.: The experiments were performed on YouTube [30], and in particular on a subset of 10000-page videos of YouTube (called YouTube10000). The videos have been chosen by starting the crawling on the video of Section I and continuing on the related videos until the desired number has been reached. Each video and its comments have been crawled using AJAX. For performance reasons, we restricted the number of comment pages that we retrieved to ten (e.g., ten comment pages per video). The characteristics of the data are displayed in Table VIII.

Parameter	Value
No. of pages	10000
No. of states	41572
No. of events	187980
No. of events per page	avg. 4

TABLE VIII
YOUTUBE STATISTICS.

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

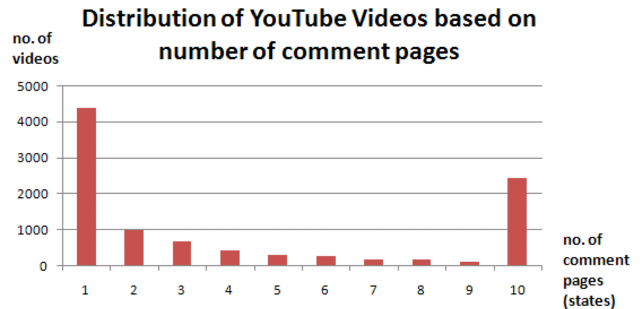


Fig. 9. Distribution of YouTube videos based on number of comment pages.

Figure 9 shows the distribution of videos with a given number of comment pages (i.e., AJAX states). Most videos have, indeed, just a single page of comments. However, there are enough videos with a lot more than one page, and crawling them leads to better search results, as we will show below. This is also what motivated this work in the first place.

The number of additional pages themselves shows just one dimension on which the amount of crawled content increases. The processing time of any crawler will be influenced by the number of actual events that must be invoked on the page in order to fetch all the AJAX content, and which result in network connection times. Figure 10 shows that especially the number of events grows at least polynomially, and mostly affects the overhead that needs to be handled by an AJAX Crawler in YouTube.

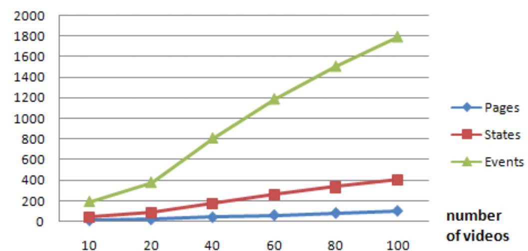


Fig. 10. Number of Pages, States and Events in YouTube.

In any YouTube page, there is an average of four events that can be invoked by a user (*next*, *prev* and direct “jumps” to the immediately few previous and next pages), as it was shown in Figure 1. This leads to a lot more events than pages, including duplicate events. They will be avoided using the *Hot Node* policy as we will show below.

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

1. Traditional Crawling. We configured the AJAXSearch to read just the first state of each YouTube video. In case of YouTube, this means the first comment page (i.e., 10

comments). Traditional crawling reads the same content that is obtained when JavaScript is disabled in a user's browser.

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

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

4. AJAX Parallel. We parallelized the AJAX Crawler on four nodes, as shown in Section V-E. While all previous configurations ran in sequential mode (Figure 6), this version of the crawler runs in parallel mode (Figure 8) and uses the *Hot Node* approach for better performance.

When search capabilities are used, we implemented the Indexing and Query Processing capabilities of Section V. The Crawling process(es) stored the AJAX model and disk. The Index was constructed incrementally [6] from the application model, and is fully maintained in memory. We ran the experiments on a Intel Xeon 3050 2.13GHz, 2 MB L2 Cache, Dualcore with 4 GB RAM (with ECC, DDR-2 533 Mhz), 1 x 250 GB S-ATA 7200 rpm hard disk, 1 x 500 GB S-ATA 7200 rpm hard disk running Windows 2003 Server. More capabilities of the framework, which go beyond the scope of this paper, are described in [12]. Javascript code was analyzed using the COBRA Toolkit [10] toolkit.

B. Crawling Time

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

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

AJAX Crawling is Expensive. The Crawling Time for 10000 pages with AJAX is in the range of about 13 hours.

AJAX Crawl induces an overhead over Traditional Crawl. AJAX Crawling non-optimized takes about twice as much as Traditional Crawling. AJAX Cached however is just about 1.33 slower than Traditional AJAX which just indexes one state.

Network Time is predominant. It can be seen that most of the time needed during Crawling is taken by Network Time (78%). This also underlines the importance of applying the *Hot Node* Optimization. Furthermore, just the network of Non-Cached version is a factor of 1.41 higher than in the AJAX Cached version of the Crawler. This results in an important slice of the time being spent on Network Time, as shown in Figure 11, when an increasing number of videos is crawled.

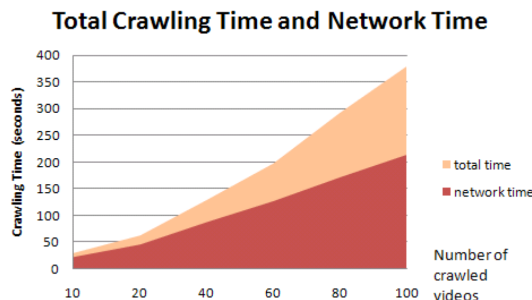


Fig. 11. Crawling and Network Time for increasing number of crawled videos.

The Hot Node Heuristics is effective. The heuristic approach of the Hot Nodes causes a 1.29 factor of improvement in crawling time as opposed to the Non-Cached Approach. This is due to the reduced number of AJAX Calls per page as opposed to the Non-Cached Approach. In case of YouTube, this results in a lower number of clicks on *next* or *prev* links, if a direct click on a link which leads to the same page number has already been done in the crawler. Figure 12 displays the number of AJAX events from the crawled pages for which a network call was needed for an increasing number of crawled videos (one to one hundred) and shows that using the Hot Node approach causes a reduction with a factor of four of the needed network calls.

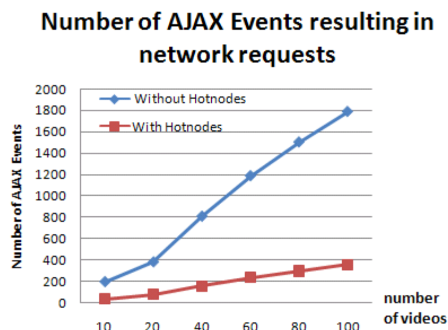


Fig. 12. Number of AJAX Events resulting in AJAX Calls with and without caching policy.

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

	Trad.	AJAX) Non-cached	AJAX Cached	AJAX Parallel	AJAX Cached/ Trad.	AJAX Non-Cached/ AJAX Cached	AJAX Cached/ AJAX Parallel
Total time (secs.)	28093.45	47933.27	37291.61	9750	x1.33	x1.29	x3.82
Network time (secs.)	21561.22	37261.19	26501.21	6496.89	x1.23	x1.41	x4.08

TABLE IX

CRAWLING TIME (SECS.) FOR TRAD. AND AJAX CRAWLING.

	Trad.	AJAX Non-cached	AJAX Cached	AJAX Parallel	AJAX Cached/ Trad.	AJAX Non-Cached/ AJAX Cached	AJAX Cached/ AJAX Parallel
Pages/second	0.36	0.21	0.27	1.53	x0.75	x0.78	x0.26
States/second	0.36	0.87	1.11	4.26	x3.13	x0.78	x0.26

TABLE X

CRAWLING THROUGHPUT (STATES AND PAGES/SECOND) FOR TRAD. AND AJAX CRAWLING.

AJAX pages.

Crawling Throughput. It is interesting to compute the crawling throughput in pages per second and states per second. In particular, it is important to mention the overhead of AJAX Crawling in terms of number of indexed pages. Table X shows the page and state throughput for all crawling methods. It can be seen that the throughput of AJAX Crawling of Youtube10000 is a factor of 0.75 slower than the throughput of Traditional Crawling. It translates in a time of 3.7 seconds per page for AJAX, as opposed to 2.7 seconds per traditional page. The basic crawling time in both cases is caused by the parsing libraries we used, and can be optimized.

C. Application Model Size.

As shown in Section II, a contribution of this paper is the new model of AJAX Web Sites. The model is an annotated automaton, useful for describing and restoring transitions in the AJAX Web site. Crawling AJAX Applications is therefore influenced by the maintenance and size of the application model. Table XI shows the size of the application model in case of traditional crawling and of the AJAX Crawling, after is has been serialized on disk. Of course, this shows a clear overhead in terms of space as opposed to traditional AJAX Sites. The Size of the AJAX Model is mostly influenced by the size of the DOM which is 692MB in case of AJAX Crawl.

	Trad.	AJAX (all methods)
App. Model Size (MB)	858	1154

TABLE XI

APPLICATION MODEL SIZE (MB) FOR TRAD. AND AJAX CRAWLING.

The model just needs to be maintained during crawling, is used after query processing in order to restore a certain state. Since the model is not updated after crawling, but needs to be quickly read, compression techniques are left as an avenue for future work.

D. Query Processing Time

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

1) Queries: The queries we performed are taken from the most popular YouTube queries, retrieved from [31]. A snippet of these queries is shown in Table XII. There are 100 queries in total. The Table shows a sample of nine queries in order of cardinality. We specify the number of videos where the query appear in the first comment page, and the total number of comments where the keywords appear. [1].

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

TABLE XII

YOUTUBE QUERIES.

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

ID	Query	Query Time Trad.	Query Time AJAX
Q1	youtube	1984.3	69054.9
Q2	with you	1009.4	38196.1
Q3	wow	0	14
Q4	funny	1.5	14.1
Q5	dance	1.6	39
Q6	kiss	0	6.2
Q7	low	1.5	3.2
Q8	fight	0	1.6
Q9	akon	0	7.8

TABLE XIII

QUERY PROCESSING TIMES ON YOUTUBE.

The throughput varies much for individual queries, as shown in Figure 13, but generally traditional search offers better page throughput, although for a much smaller number of results.

E. Recall

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

[1]for the numbers we used a subset of the first 500 videos

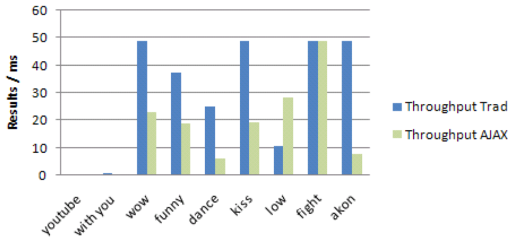


Fig. 13. Throughput of popular YouTube queries in Traditional and AJAX Search.

	Trad.	AJAX (all methods)
Recall	0.14	1

TABLE XIV

RECALL FOR TRAD. AND AJAX CRAWLING.

F. Partial Crawling. Setting Crawling Threshold.

Crawling AJAX brings better results, with the price of performance. The performance that directly affects the user is however not the crawling performance, but the query performance and in particular *query throughput*. For the purpose of establishing this tradeoff, we executed a partial crawling of the YouTube Website, by keeping the number of pages constant (i.e., 10000), and varied the maximum amount of crawled AJAX states per page from one to ten.

Query Throughput. Partial Crawling allows to study when the throughput of AJAX Search exceeds an acceptable limit. Figure 14 shows the throughput for Traditional and AJAX Crawl when one to ten AJAX states per video are crawled. This gives a first hint on where implementers of an AJAX crawl engine can decide the best tradeoff between the amount of AJAX content that is indexed and the query performance. If the threshold is set so that a factor of two occurs in decreasing the query throughput, crawling limit can be set to four states per video.

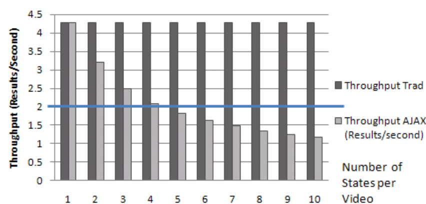


Fig. 14. Throughput for Traditional and AJAX Search in case of Partial Crawling.

Recall. Partial Crawling has the effect that the more AJAX States are retrieved (i.e., comment pages in YouTube), the worse the computed recall of traditional search gets. Another hint on setting the crawling threshold is to crawl as many states per AJAX page until the recall difference between traditional and AJAX Crawl reaches a "good enough" level. We consider all results returned by AJAX Search as the relevant results (i.e., the desired ones) and we assume boolean retrieval. Figure 15 shows the recall of traditional search and AJAX search (which is constant, equal to one). Also in this case, crawling between a maximum of four to five states is already enough to gain a 0.7 increase in Recall over Traditional Search. This result is of most importance for current search engine implementors, since it shows that it is enough to crawl a relative small amount of AJAX content in order to get an important increase in Recall.

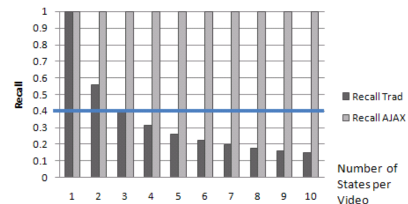


Fig. 15. Recall for Traditional and AJAX Search in case of Partial Crawling.

VII. DISCUSSION: INTEGRATING AJAX CRAWL INTO EXISTING SEARCH ENGINES

Currently search engines do not use AJAX Crawling because naïve crawling is expensive. We have shown that AJAX Crawling is possible although it does induce a real overhead over traditional search. There are however three points which encourage existing search engines to incorporate AJAX Search: First, the AJAX model extends current Web model. The existing model of the Web is maintained "as is" and is just enhanced in AJAX Crawl. The extensions can also be dropped, reducing the model to that of traditional Web. Second, indexing is based on traditional methods. The index structures that we used are also extensions of traditional inverted files. This makes the implementation into existing search engines straightforward. Third, a Performance/Quality benefit can be obtained. The stronger assumptions of commercial search engines is that crawling AJAX is expensive. We have shown that the *Hot Node* approach improves the naïve approach through the process of memoization [13]. Furthermore, we have experimentally shown that a very good performance/quality benefit can be achieved by just partially Crawling the AJAX Content. This tradeoff, along with the parallelization optimzation successfully addresses the obvious demand for *scalability*. Finally, the techniques used for AJAX can also be applied for other types of Rich Internet Application such as Flash, minimally addressed by commercial search engines.

VIII. RELATED WORK

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

AJAX Crawling. We are the first paper to list an automatic AJAX Crawling Algorithm and optimization. A state machine was used for modelling AJAX in [22] and constructed using a diff-based approach, but manual modeling was used by the authors in extensions such as [21]. Software engineering studies the extreme problem of a Web site being merged into

a single AJAX page [23]. Giving hints to the AJAX crawler of our paper is also possible, as shown in [12]. The Hot Node approach is a memoization technique, also encountered in the context of XQuery in [13].

Duplicate elimination. There are two kinds of duplicate cases during crawling: syntactic and semantic duplicates. The syntactic duplicates are more than one page linking to the same page with the same URL. Traditional crawlers extract and store all static URLs in a look-up table used to avoids repeated access to the same URL. For the semantic duplicates, several algorithms exist to detect changes in tree structures: [8] or near-duplicate Web pages: [17], [20], Broder et al.'s shingling algorithm[5] and Charikar's [7], [20] random projection. DustBuster (Different URLs with Similar Text) [3] tries to detect duplicate URLs using mining. This is not possible for AJAX application crawling. Traditional Web Crawling has been addressed in works such as [4]. Shah [25] crawls YouTube but for Data Mining purposes.

Javascript Analysis. The work of [19] analyzes Javascript and constructs a control flow, for testing Javascript applications. However, this model cannot help the crawler identify duplicate states in AJAX applications. Most Javascript works, such as [9], deal with detecting common spam redirects on Web pages.

Application Models. Colorful XML [18] encodes multiple views over the data in the same data structure; we are similar in the approach (i.e., multiple states in a single-page application), but their model lacks the notion of transition. The Transition Graph bears similarities with ActiveXML [1], a dynamic XML structure enhanced with dynamic calls. Active XML views are however always evolving, and offer just snapshots of current states as opposed to AJAX Crawling.

Deep Web. Because AJAX applications interact with the server using user input, crawling AJAX is related to Hidden Web [11], [24], [26], [27], [28]. AJAX Crawl does more than a hidden Web crawler, since it focuses on very granular interactions with the application and since it escapes the page paradigm commonly used in search.

IX. Conclusions and Future Work

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

There are several avenues for future work, most address ways to decrease the number of AJAX content that needs to be crawled. The first one is crawling more current AJAX applications, such as Google Maps. A second one is to address forms in AJAX applications. Most AJAX applications allow user input. Combining AJAX Search and work on Deep Web can provide insight on which content is relevant for crawling. The next one is crawling and personalization, i.e., focusing on a specific user's or group's interaction with the server. This would mean that only relevant states and events are crawled, improving both quality and performance. Finally, crawling AJAX can also be seen as a repetitive process, which can reduce the number of crawled events, by ignoring events which did not cause large changes in previous crawling sessions.

References

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In SIGMOD, 2004.
[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. Modern Information Retrieval. ACM Press / Addison-Wesley, 1999.
[3] Z. Bar-Yossef, I. Keidar, and U. Schonfeld. Do not Crawl in the Dust: Different URLs with Similar Text. In WWW, 2007.
[4] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. Computer Networks and ISDN Systems, 1998.
[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic Clustering of the Web. In WWW, 1997.
[6] E. W. Brown, J. P. Callan, and W. B. Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In VLDB, 1994.
[7] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In STOC, 2002.
[8] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In SIGMOD, 1996.
[9] K. Chellapilla and A. Maykov. A Taxonomy of JavaScript Redirection Spam. In AIRWeb, 2007.
[10] COBRA Toolkit. http://html.xamjwg.org/cobra.jsp.
[11] A. de Carvalho Fontes and F. S. Silva. SmartCrawl: A New Strategy for the Exploration of the Hidden Web. In WIDM, 2004.
[12] C. Duda, G. Frey, D. Kossmann, and C. Zhou. AJAXSearch: Crawling, Indexing and Searching Web 2.0 Applications. In VLDB, 2008.
[13] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In VLDB, 2003.
[14] Google Mail. http://www.gmail.com.
[15] Google Maps. http://maps.google.com.
[16] Google Suggest. http://www.google.com/webhp?complete=1&hl=en.
[17] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In SIGIR, 2006.
[18] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: One Hierarchy Isn't Enough. In SIGMOD, 2004.
[19] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. Structural Testing of Web Applications. In ISSRE, 2000.
[20] G. S. Manku, A. Jain, and A. D. Sarma. Detecting Near-Duplicates for Web Crawling. In WWW, 2007.
[21] A. Marchetto, P. Tonella, and F. Ricca. State-Based Testing of Ajax Web Applications. In ICST, 2008.
[22] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by Inferring User Interface State Changes. In ICWE, 2008.
[23] A. Mesbah and A. van Deursen. Migrating Multi-page Web Applications to Single-page AJAX Interfaces. In CSMR, 2007.
[24] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In VLDB, 2001.
[25] C. Shah. YouTube Crawling: A VidArch Year in Retrospect. http://www.ils.unc.edu/vidarch/vidarch-annrep-2008.pdf.
[26] J. Wang, J.-R. Wen, F. Lochovsky, and W.-Y. Ma. Instance-based Schema Matching for Web Databases by Domain-Specific Query Probing. In VLDB, 2004.
[27] W. Wu, A. Doan, and C. Yu. WebIQ: Learning from the Web to Match Deep-Web Query Interfaces. In ICDE, 2006.
[28] W. Wu, C. Yu, A. Doan, and W. Meng. An Interactive Clustering-based Approach to Integrating Source Query Interfaces on the Deep Web. In SIGMOD, 2004.
[29] Yahoo! Mail. http://mail.yahoo.com.
[30] YouTube. http://www.youtube.com.
[31] YouTube Video Keyword Research and Characteristics of Popular YouTube Queries . http://www.brysonmeunier.com/youtube-video-keyword-research-and-characteristics-of-popular-youtube-queries.