Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval

Held in Portland, Oregon, USA,
16th August 2012

Edited by
Andrew Trotman,
Charles L. A. Clarke,
Iadh Ounis,
J. Shane Culpepper,
Marc-Allen Cartright,
and
Shlomo Geva.

Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval.

Held in Portland, Oregon, USA,
16th August 2012.

Published by:
Department of Computer Science,
University of Otago,
PO Box 56,
Dunedin,
New Zealand.

Editors:
Andrew Trotman,
Charles L. A. Clarke,
Iadh Ounis,
J. Shane Culpepper,
Marc-Allen Cartright,
and
Shlomo Geva.

Paper Edition: ISBN 978-0-473-22025-9
Electronic Edition: ISBN 978-0-473-22026-6

Preface

These proceedings contain the papers of the SIGIR 2012 Workshop on Open Source Information Retrieval held in held in Portland, Oregon, USA, on the 16th of August 2012. Six full papers and six short papers were selected by the program committee from thirteen submissions (92% acceptance rate). Each paper was reviewed by three members of the international program committee. In addition to these selected papers, invited talks were given by Grant Ingersoll "OpenSearchLab and the Lucene Ecosystem" and Jamie Callan "The Lemur Project and its ClueWeb12 Dataset". We thank them for their special contributions.

When reading this volume it is necessary to keep in mind that these papers represent the opinions of the authors (who are trying to stimulate debate). It is the combination of these papers and the debate that will make the workshop a success. We would like to thank the ACM and SIGIR for hosting us. Thanks also go to the program committee, the paper authors, and the participants, for without these people there would be no workshop.

Andrew Trotman,
Charles L. A. Clarke,
Iadh Ounis,
J. Shane Culpepper,
Marc-Allen Cartright,
and
Shlomo Geva

Workshop Organisation

Program Chairs

Andrew Trotman
Charles L. A. Clarke
Iadh Ounis
J. Shane Culpepper
Marc-Allen Cartright
Shlomo Geva

Program Committee

Andrew Trotman
C. Lee Giles
Charles L. A. Clarke
Claudia Hauff
Craig Macdonald
David Hawking
Djoerd Hiemstra
Giambattista Amati
Iadh Ounis
J. Shane Culpepper
Jamie Callan
Marc Cartright
Michael Stack
Michel Beigbeder
Richard Boulton
Samuel Huston
Shlomo Geva

Table of Contents

WikiQuery -- An Interactive Collaboration Interface for Creating, Storing and Sharing Effective CNF Queries

Le Zhao
Carnegie Mellon University
lezhao@cs.cmu.edu

Xiaozhong Liu
Indiana University, Bloomington
liu237@indiana.edu

Jamie Callan
Carnegie Mellon University
callan@cs.cmu.edu

ABSTRACT

Boolean Conjunctive Normal Form (CNF) expansion can effectively address the vocabulary mismatch problem, a problem that current retrieval techniques have very limited ability to solve. Meanwhile, expert searchers are found to spend large amounts of time carefully creating manual CNF queries. These CNF queries are highly effective, and can outperform bag of word queries by a large margin. However, not many effective tools exist that can facilitate the efficient manual creation of effective CNF queries.

We describe such a publicly available search tool, WikiQuery, which can efficiently assist the users to create CNF queries through easy query editing and immediate access to search results. Experiments show that ordinary search users, with limited prior knowledge of Boolean queries, can use this intuitive tool to create effective CNF queries. We argue that tools like WikiQuery can attract and retain certain users from the commercial Web search engines, and may be a good starting point to build a research Web search engine.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]:

General Terms

Theory, Experimentation, Measurement

Keywords

Wiki for queries, conjunctive normal form (CNF) queries, query refinement, user interactions

1. INTRODUCTION

One particular goal of the Open Source Information Retrieval workshop is to build 'an open source, live and functioning, online web search engine for research purposes'. A key factor necessary for the success of such an effort is to *attract* and *retain* users.

In order to attract users, the search engine needs to have a *distinct* and *useful* feature that is not offered by the current search engines. As a somewhat negative example, the Lemur community query log project did not collect enough query log data perhaps due to the lack of any additional benefit provided by the query log toolbar[1]. Compared to the toolbar, a full scale open source search engine is even more likely to fail, as the quality of the results from such an academic search engine is likely to be much worse than that from the commercial Web search engines.

In order to retain users, it is perhaps necessary that the distinct feature is *unlikely to be copied* by the competitors (the commercial Web search engines).

[1] http://lemurstudy.cs.umass.edu/

This paper describes one such publicly available open source search tool, WikiQuery (http://www.wikiquery.org), which both engages ordinary searchers in effective search interactions, and is unlikely to be adopted by the commercial Web search engines. WikiQuery can provide more effective search interactions than what the current search engines can offer, and is flexible enough to be applied on top of virtually any Web search engine.

Prior research showed that the current retrieval techniques are still very limited in their ability to solve the vocabulary mismatch problem [13]. Users are still frequently frustrated by the current search engines when performing informational searches [5]. Prior research also indicated that high quality manually created Conjunctive Normal Form (CNF) queries offer the opportunity to address this limitation and significantly improve retrieval beyond the traditional bag of word queries [14]. A huge potential of improvement is possible in the scale of 50-300% with carefully manually created CNF queries [14].

The WikiQuery interface is designed to guide and facilitate users to create highly effective CNF queries efficiently through 1) a simple CNF input interface, 2) immediate inspection and interaction with search results from multiple commercial search engines, and 3) collaboration with other users who share related information needs. The created queries are stored, and readily available for future re-finding or refining. The queries are also shared online so that other users may benefit from the queries or query parts. Being a Wiki website, different users can collaborate and improve queries together. This interface is implemented based on the MediaWiki source code, which allows the users to search for pages or information stored on the website, so that it is easy to lookup, share or collaborate on the website.

User studies in this work show that ordinary search users with limited knowledge of Boolean queries have the potential to use the WikiQuery interface to create effective CNF queries.

WikiQuery has the potential to attract and retain users for two reasons. Firstly, the CNF query interface is effective and intuitive, and can appeal to the ordinary search users -- at least the early adopters who are willing to learn a new and effective way to formulate search queries, or the more serious users who care about their searches. Secondly, the commercial Web search engines are very unlikely to adopt the CNF interface, because the change in user experience is large enough to scare away the change-averse users, making it very risky to use for a large Web search engine.

In addition to the added benefit of facilitating search interactions, the resulting crowdsourced CNF queries stored on the WikiQuery website also constitute a detailed context dependent thesaurus for retrieval and other vocabulary tasks.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 describes the WikiQuery website together with its CNF query interface. Section 4 reports the studies showing that ordinary users can create effective CNF queries with the proper tool and guidance. Section 5 concludes the paper.

2. RELATED WORK

This section reviews prior work related to three aspects of this work, the uses of Boolean CNF queries, Boolean user interfaces, and the use of the user generated Boolean queries as a resource for thesaurus building. We also discuss how this research is different from prior efforts.

2.1 Uses of CNF Queries

Prior research on effective uses and formulations of Boolean CNF queries motivates this research. The use of Conjunctive Normal Form (CNF) queries is widespread among librarians [9,6], lawyers [3,2], and other expert searchers [7,4,11].

For example, the query below from TREC 2006 Legal Track [2]

"*sales of tobacco to children*"

is expanded manually into the Boolean CNF query

(*sales* OR *sell* OR *sold*) AND
(*tobacco* OR *cigar* OR *cigarettes*) AND
(*children* OR *child* OR *teen* OR *juvenile* OR *kid* OR *adolescent*)

In the above case, each query term is expanded into one conjunct of the Conjunctive Normal Form query.

Earlier research on Boolean queries examined *unranked* Boolean retrieval, and showed that ranked keyword retrieval is more effective, mainly because presenting retrieval results as a set is both difficult to control and inefficient to examine. Later research compared *ranked* Boolean with keyword retrieval, showing that user created CNF queries can significantly improve over keyword retrieval by simply grouping the query terms of the verbose keyword queries into Conjunctive Normal Form [7,11].

More recent research showed that lawyers and search experts can create highly effective CNF queries that extensively expand the original keyword queries, solving mismatch and improving retrieval 50-300% [14]. These CNF queries with high quality expansion terms were shown to outperform bag of word expansion with the same set of high quality expansion terms.

2.2 Boolean Search Interfaces

Even though carefully created CNF queries are effective, recent research has focused on bag of word queries, and has not seen much development in interfaces that help users create effective CNF queries. Research on Boolean user interfaces happened mostly before mid 1990s. Hearst [1, Chapter 10] cited several textual as well as graphical Boolean interfaces. Hearst referred to CNF queries as *faceted queries*, and described a possible textual input interface for CNF queries, though without a concrete example. In a newer book, Hearst [8] cited the advanced search interface of the Educational Resources Information Center (ERIC)[2], which allows the entry of CNF queries in a one-conjunct-per-line format. This is similar to the CNF interface of WikiQuery except for two differences. Firstly, the ERIC interface is not specifically designed for CNF queries, and allows the user to enter a query in Disjunctive Normal Form. Secondly, the ERIC interface gives no guidance or useful examples to the user on how to create effective Boolean queries.

The lack of research on Boolean interfaces is coupled with a long list of negative results [8, Section 4.4] showing that ordinary users have a difficult time formulating effective Boolean queries. This work, on the contrary, shows that ordinary search users with limited knowledge of Boolean queries have the potential to create effective Boolean CNF queries using the WikiQuery interface. This apparent contradiction is likely because the prior studies did not focus on Boolean CNF queries, and gave novice users the full freedom of free form Boolean queries without proper guidance. This choice leaves the creation of effective Boolean queries to the chances, and is likely to lead to ineffective Boolean queries. Our results point at a promising direction of designing search interfaces that guide and facilitate users to formulate effective Boolean queries in CNF form.

2.3 Online Thesaurus Building

The resulting CNF queries created by users and stored in WikiQuery can serve as a thesaurus for future users. In particular, each conjunct in the CNF queries contains synonyms or related terms that are dependent on the context of the query. Compared to existing thesauri like WordNet, the WikiQuery synonyms depend on the specific uses of a term in a query, while WordNet is still a static semantic resource without regard to word use.

The thesaurus building aspect of the WikiQuery website is similar to an earlier system that builds a growing thesaurus based on users' Boolean retrieval interactions [12]. The main difference is the emphasis on CNF queries by WikiQuery. WikiQuery also treats individual queries as valuable resources, and as units for storage and retrieval. This is a fairly lazy and ad hoc treatment for a thesaurus. Later more general treatments can build on top of the queries stored on WikiQuery, when it becomes clear what kinds of general treatments are most appropriate.

3. THE WIKIQUERY WEBSITE

The search tool described in this work is a public Wiki website based on the same source code that supports Wikipedia etc. sites.

On the WikiQuery website, each Wiki page stores all the information about one particular user information need, including possibly a description of the information need, the corresponding CNF query (or several related CNF queries), possible relevant results (together with descriptions) identified through the search interactions, or other related information.

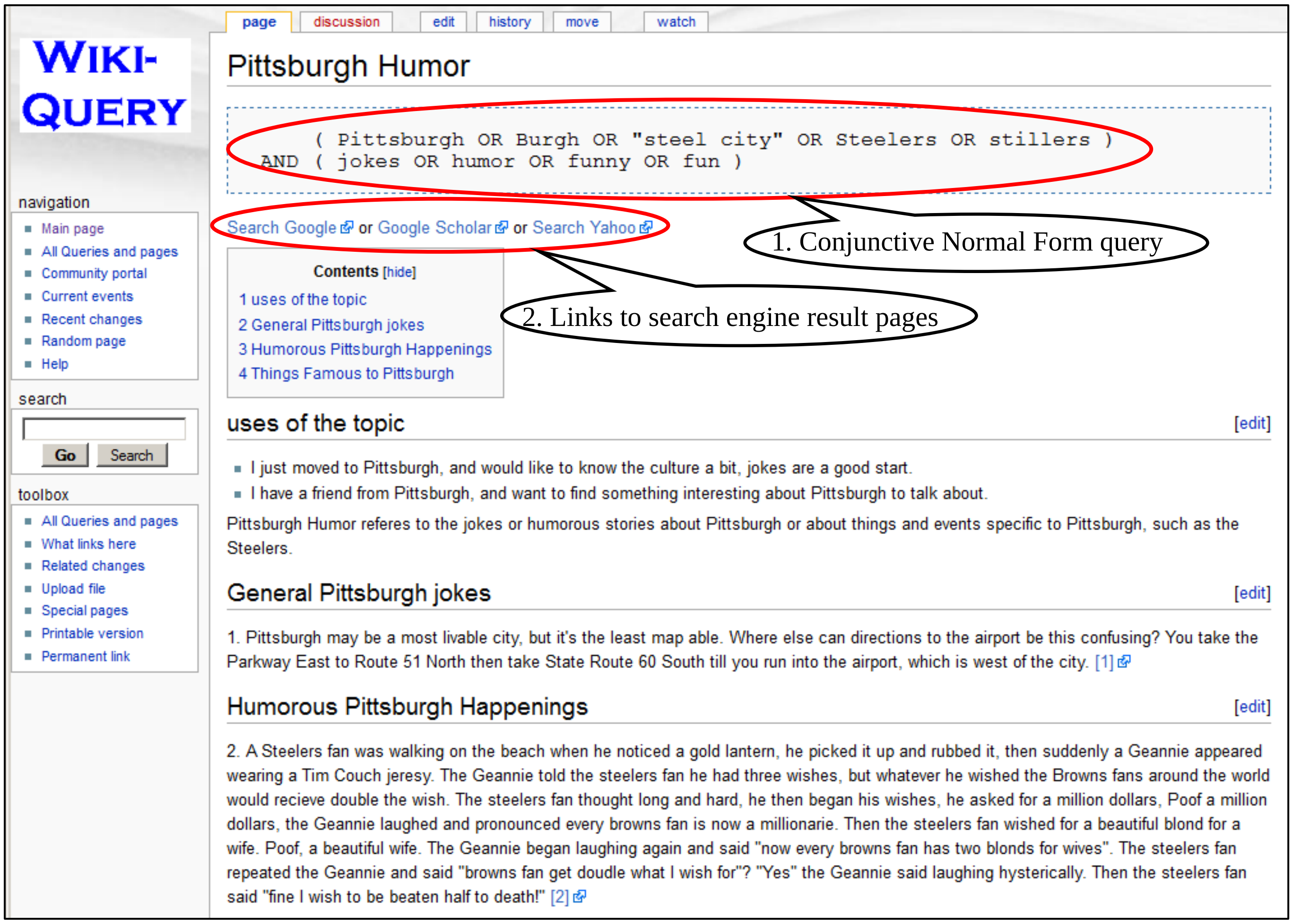
An example WikiQuery page is shown in Figure 1. The main CNF query of the page and the links to the search engine result pages from multiple search engines are circled out.

The open source MediaWiki code (http://www.mediawiki.org) offers the standard set of features used in popular Wiki websites. One useful function allows the users to search for pages or information stored on the website through entering a search query in the search box. In addition, being based on MediaWiki version 1.17, the Wiki website automatically suggests existing WikiQuery pages as the user types into the search box. Other features include history tracking of all the user edits of pages and users, subscribing to a page to monitor changes made to the page, and opportunity of discussion among contributors of a Wiki page.

Several simple customizations were made to accommodate the special user needs for the WikiQuery website, including 1) a simple textual interface for CNF query editing, 2) an automatic client side script to display the query and store it in the Wiki page, and 3) an automatic script that translates the CNF query into the formats accepted by common Web search engines, allowing immediate inspection of the retrieval results produced by the CNF queries. The rest of this section covers these customizations in more detail.

3.1 Interface for CNF Query Editing

[2] http://www.eric.ed.gov/ERICWebPortal/search/extended.jsp accessed on June 1st, 2012.



Pittsburgh Humor

```
    ( Pittsburgh OR Burgh OR "steel city" OR Steelers OR stillers )
AND ( jokes OR humor OR funny OR fun )
```

Search Google or Google Scholar or Search Yahoo

Contents [hide]

1 uses of the topic
2 General Pittsburgh jokes
3 Humorous Pittsburgh Happenings
4 Things Famous to Pittsburgh

uses of the topic [edit]

- I just moved to Pittsburgh, and would like to know the culture a bit, jokes are a good start.
- I have a friend from Pittsburgh, and want to find something interesting about Pittsburgh to talk about.

Pittsburgh Humor referes to the jokes or humorous stories about Pittsburgh or about things and events specific to Pittsburgh, such as the Steelers.

General Pittsburgh jokes [edit]

1. Pittsburgh may be a most livable city, but it's the least map able. Where else can directions to the airport be this confusing? You take the Parkway East to Route 51 North then take State Route 60 South till you run into the airport, which is west of the city. [1]

Humorous Pittsburgh Happenings [edit]

2. A Steelers fan was walking on the beach when he noticed a gold lantern, he picked it up and rubbed it, then suddenly a Geannie appeared wearing a Tim Couch jeresy. The Geannie told the steelers fan he had three wishes, but whatever he wished the Browns fans around the world would recieve double the wish. The steelers fan thought long and hard, he then began his wishes, he asked for a million dollars, Poof a million dollars, the Geannie laughed and pronounced every browns fan is now a millionarie. Then the steelers fan wished for a beautiful blond for a wife. Poof, a beautiful wife. The Geannie began laughing again and said "now every browns fan has two blonds for wives". The steelers fan repeated the Geannie and said "browns fan get doudle what I wish for"? "Yes" the Geannie said laughing hysterically. Then the steelers fan said "fine I wish to be beaten half to death!" [2]

Figure 1. An example Wiki page from the WikiQuery Website. Circled out are the Conjunctive Normal Form (CNF) query of this Wiki page and the links to the search engine result pages for the CNF query.

The CNF query interface is presented to the user when the user edits a Wiki page. It guides the user and allows the user to easily and efficiently create or edit CNF queries. As shown in Figure 2, this interface is consisted of several input bars, each corresponding to one conjunct in the CNF query. The user has to determine how many and what concepts (conjuncts) are necessary for the particular information need. Then the user has to enter in each input bar the search terms that can be used to describe the concept, and join them with the Boolean OR operator. Prior research [14] indicated that including more high quality expansion terms in each conjunct yields a higher likelihood for the conjunct to match the relevant documents of the query, and leads to a higher retrieval accuracy.

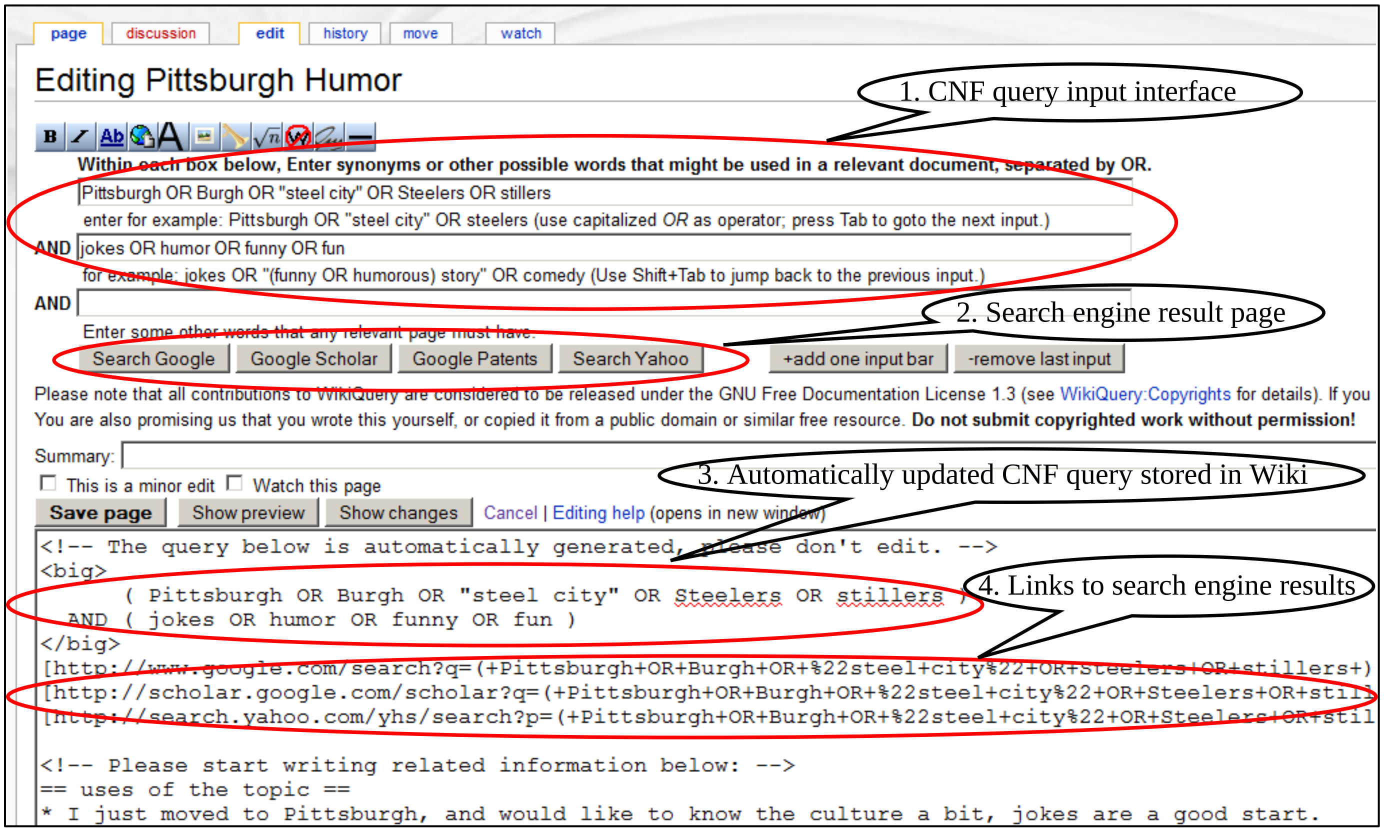
The CNF queries are stored on the wiki to allow users to revisit existing queries, and to further improve the queries. Because refinding tasks are fairly common in Web search, a user might frequently find the stored queries to be helpful at a future time.

Whenever the user edits an existing WikiQuery page that already contains a full CNF query, the CNF input bars are automatically populated with the content of the CNF query, so that the user does not have to enter the query into the input bars again.

The collaborative nature of the Wiki website also allows different users to collaborate and edit the same WikiQuery page of common interest to these users. For popular information needs, collaborations across multiple users are likely to improve the quality of the CNF queries beyond what a single user may achieve. Because high quality CNF queries can take lots of effort to create, collaboration offers the possibility to break down the difficulty through sharing it among a group of users.

This Boolean CNF interface is different from the typical interfaces used in advanced searches in libraries or by lawyers in legal discovery. The advanced search interfaces in libraries (e.g. Library of Congress) allow a restricted Boolean query of the form: Term1 Op1 Term2 Op2 …, where a Term can be a word or a phrase, but *cannot* be a Boolean clause, and an Op is a Boolean operator (e.g. AND, OR, XOR, and NOT). The WikiQuery CNF interface is more powerful than the library Boolean interfaces because any Boolean query can be expressed as a CNF query.

The Boolean interface used by lawyers is usually just one large text box. They are flexible and allow free form Boolean queries to be entered into a, typically large, text box. However, the lawyers typically create CNF-like queries [2], and have to enter the whole query by themselves, having to make sure that the parentheses match and the form is correct. The WikiQuery CNF interface facilitates simpler and more efficient manual creation of CNF-like queries. It breaks down each query by allowing the user to enter each conjunct into one input box. This way, the user does not need to enter the CNF skeleton, nor the conjunct level parentheses. Although this CNF interface suggests the use of CNF-like queries, it does not require that. The user still has the freedom to


page
discussion
edit
history
move
watch
1. CNF query input interface
2. Search engine result page
3. Automatically updated CNF query stored in Wiki
4. Links to search engine results
Search Google
Google Scholar
Google Patents
Search Yahoo
+add one input bar
-remove last input
Save page
Show preview
Show changes
Cancel | Editing help (opens in new window)

Editing Pittsburgh Humor

Within each box below, Enter synonyms or other possible words that might be used in a relevant document, separated by OR.

Pittsburgh OR Burgh OR "steel city" OR Steelers OR stillers

enter for example: Pittsburgh OR "steel city" OR steelers (use capitalized *OR* as operator; press Tab to goto the next input.)

AND jokes OR humor OR funny OR fun

for example: jokes OR "(funny OR humorous) story" OR comedy (Use Shift+Tab to jump back to the previous input.)

AND

Enter some other words that any relevant page must have.

Please note that all contributions to WikiQuery are considered to be released under the GNU Free Documentation License 1.3 (see WikiQuery:Copyrights for details). If you

You are also promising us that you wrote this yourself, or copied it from a public domain or similar free resource. **Do not submit copyrighted work without permission!**

Summary:

☐ This is a minor edit ☐ Watch this page

```
<!-- The query below is automatically generated, please don't edit. -->
<big>
      ( Pittsburgh OR Burgh OR "steel city" OR Steelers OR stillers )
  AND ( jokes OR humor OR funny OR fun )
</big>
[http://www.google.com/search?q=(+Pittsburgh+OR+Burgh+OR+%22steel+city%22+OR+Steelers+OR+stillers+)
[http://scholar.google.com/scholar?q=(+Pittsburgh+OR+Burgh+OR+%22steel+city%22+OR+Steelers+OR+still
[http://search.yahoo.com/yhs/search?p=(+Pittsburgh+OR+Burgh+OR+%22steel+city%22+OR+Steelers+OR+stil

<!-- Please start writing related information below: -->
== uses of the topic ==
* I just moved to Pittsburgh, and would like to know the culture a bit, jokes are a good start.
```

Figure 2. The editing interface of WikiQuery. It includes a Conjunctive Normal Form (CNF) query editing interface and buttons to access search engine result pages. The full CNF query and the HTTP links to the search engine result pages for the query are automatically generated and stored in the page.

enter a free form Boolean query into one input bar in the WikiQuery interface, although this usage is generally discouraged.

3.2 Query Storage and Display

The query that the user enters into the CNF editing interface (Figure 2) is automatically translated into the full form CNF query that is stored and displayed on each Wiki page (as seen in Figure 1). Whenever the user makes a change in one of the input bars in the CNF edit interface, a short Javascript is automatically triggered to translate the contents of the input bars into the full query as part of the content of the Wiki page to be stored.

Because the editing interface uses the document content of the Wiki page to store and display CNF queries, all the benefits from MediaWiki for maintaining the Wiki contents automatically apply to the stored CNF queries, change tracking and searching.

3.3 Querying Search Engines

The WikiQuery website allows the user to access search engine result pages for the CNF queries very easily, during page viewing and editing. Access to the result pages during page view allows viewers of the WikiQuery website to easily check the search engine results for a CNF query. Access to the result pages in the editing interface allows the user to monitor the quality of the search engine results after making changes to the CNF queries, ensuring the quality of the resulting CNF queries.

Figure 1 shows the links displayed on the stored WikiQuery pages during page viewing. Figure 2 shows the buttons that would open a new window to allow the user to navigate to the search engine result pages for the query that the user is editing.

Multiple search engines are supported. These CNF queries can work on any search engine that supports Boolean query operators. Currently it employs Google, Google Scholar, Google Patents and Yahoo (Altavista), but can be easily extended to use other search engines like Bing which uses a slightly different query language.

4. USER STUDY

Prior research already confirmed the effectiveness of the manual CNF queries, and that expert searchers can create effective CNF queries [14]. The study in this section aims to verify the hypothesis that ordinary users with no or limited prior knowledge of Boolean queries can create effective Boolean CNF queries using the WikiQuery CNF interface.

6 users participated in this preliminary user study, each responsible for 2 information needs. Users typically proposed a series of Boolean queries. We report the retrieval performance of the Boolean queries against the baseline keyword queries.

4.1 Experiment Setup

This subsection describes the details of this experiment, including user selection, information needs, evaluation details, relevance judgments and evaluation methodology.

4.1.1 Classroom Users

Users for this study come from an IR class in an information school. A total of 6 students participated in the study. We purposely launched the study at the beginning of the semester when students were not fully exposed to the professional CNF queries.

As participants had little knowledge of Boolean queries or Wiki page editing, a 10-minute session was given before the study, which included an example information need with a walk through the CNF query creation and editing process on WikiQuery. Because this study counts as one homework assignment for the students, participants were highly motivated to spend time and do well in creating effective CNF queries. Users were also asked to document the detailed query formulation and retrieval experience.

4.1.2 Information Needs

12 topics from TREC Ad hoc and Terabyte tracks were selected as candidate topics. Topics were selected to be somewhat interesting for the students, and reasonably difficult so that the keyword queries were unlikely to return perfect results. Each student was randomly assigned two topics, for which the student would assume the role of the searcher and create queries.

Each TREC topic contains a short title and a long description of the information need. The students used all the available information of the topic to grasp the intent behind the topic. They generated queries for each topic and were encouraged to interact with the search results to improve the proposed queries.

One reason for using standard TREC topics is to use the existing relevance judgments on these datasets to evaluate the user queries.

4.1.3 Baseline Keyword Queries

The keyword queries were directly taken from the TREC topic titles and descriptions. The topic titles are shorter, usually 2 to 4 terms long, and the descriptions are much longer, around 5 to 10 terms long. These two types of keyword queries correspond to the two baselines: keyword *title* and keyword *desc*.

4.1.4 User Created Boolean Queries

Users were asked to create Boolean CNF queries using the WikiQuery interface, which allows them to enter the query, to examine the results returned by the search engines for the query, and to improve the query. On average, a user spent around 40 minutes on each information need, based on the recorded history of changes of the WikiQuery pages. For each information need, the users created several queries or improved the CNF query many times.

The users were asked to submit two versions of Boolean queries for each information need, an initial Boolean query and a final version of the Boolean query. The initial Boolean query represents a very first try by the user, and the final Boolean query is usually the result of fine tuning the CNF query based on interactions with the retrieval results of all the queries the user tested.

4.1.5 Evaluating on TREC Datasets

Since the information needs come from official TREC Ad hoc track and Terabyte track topics, existing relevance judgments from TREC can be used to evaluate the effectiveness of the user proposed queries. The advantage of using the TREC relevance judgments is that these judgments are fairly reusable, and more complete than just evaluating several top results returned by a search engine. Thus, one can evaluate the result lists returned on the TREC datasets to much deeper levels. TREC standard evaluation metrics report retrieval accuracy of the top 1000 results. One may question why such deep level of assessment is necessary for Web search where users typically only look at top several results. We argue that the deeper level metrics are more sensitive to retrieval algorithm differences. Certain retrieval algorithm changes may not surface as top rank result changes on a small set of test topics, but may change the rank list more dramatically at deeper levels for these topics. These deeper level changes may show up at the top ranks on a small subset of topics of a much larger test topic set. At the very least, the deeper level metrics provide an additional perspective to the effectiveness of the rank lists.

The TREC relevance judgments only exist on the TREC document collections, so the queries need to be run against the smaller TREC collections, instead of a Web search engine.

We used the Indri search engine of the Lemur toolkit version 4.10 to execute the Boolean queries on TREC document collections. Indri supports a fairly comprehensive query language. The backend model is language model with Dirichlet smoothing. The Boolean OR operator is implemented as the Indri *#syn* operator. #syn counts term frequency and document frequency of the whole group of synonyms by treating all the synonyms in the group as the same term. The Boolean AND operator is implemented as the Indri *#combine* operator which is the probabilistic AND operator. This Indri implementation of a Boolean query automatically returns a rank list of documents, instead of an unranked set. This is a more effective form of result presentation than an unranked set of documents, and is widely adopted by modern retrieval systems.

Equations (1, 2) show how Indri scores document d with query (a OR b) AND (c OR e). tf(a, d) is the number of times term a appears in document d. μ is the parameter for Dirichlet smoothing, which is set at 900 for the Ad hoc track datasets and 1500 for the Terabyte track datasets.

$$\begin{aligned} &\text{Score}(\ (a \text{ OR } b) \text{ AND } (c \text{ OR } e),\ d) \qquad (1)\\ &= P(\ (a \text{ OR } b) \text{ AND } (c \text{ OR } e) \mid d)\\ &= P(\ (a \text{ OR } b) \mid d) * P(\ (c \text{ OR } e) \mid d)\end{aligned}$$

$$\begin{aligned} &P(\ (a \text{ OR } b) \mid d) \qquad (2)\\ &= (\ \text{tf}(a, d) + \text{tf}(b, d) + \mu * (P(a \mid C) + P(b \mid C))\ ) / (\text{length}(d) + \mu)\\ &= P(a \mid d) + P(b \mid d) \qquad \text{(under Dirichlet smoothing)}\end{aligned}$$

4.1.6 Evaluating on Commercial Search Engines

The WikiQuery website allows the users to run the CNF queries they created on commercial Web search engines, which typically have a much larger collection of documents, and the retrieval algorithms are typically more effective than the experimental systems used by researchers. This section tries to evaluate the effectiveness of the CNF queries by running them on commercial search engines.

Given a Boolean CNF query as input, the Web search engines return ranked lists of documents as results. The exact ranking formulae of these search engines are difficult to know. Standard ways of producing ranked retrieval results from Boolean queries include probabilistic Boolean retrieval models, quorum-level ranking [8, Section 4.4.2], or simply using keyword retrieval to rank the documents that match the Boolean query. An empirical comparison of some of them can be found in [14].

Relevance judgments are needed to evaluate the effectiveness of the results returned by the search engines. Users who participated in the study did relevance judgments for each other for the top 5 results returned by the Web search engines (Google and Yahoo). The assessors *did not know* what query, search engine or rank a particular result page comes from. These user-provided judgments were obtained at the time of the homework assignment (February to March 2011). One of the authors of this paper verified these user provided relevance judgments for accuracy.

4.1.7 Evaluation Metrics

For TREC judgments, we report Mean Average Precision (MAP) for the top 1000 results. It is a standard measure because it is sensitive to rank list changes and also fairly stable when comparing between systems [10]. We also report Precision at top 5 and 10 as measures of retrieval accuracy at top ranks. For evaluation on the search engines, we report Precision at top 5, and MAP at 5, as deeper relevance judgments are not available.

Table 1. The differences between short keyword and the final Boolean CNF query for each TREC topic. The **types of changes** include *restricting* the query (by including more conjuncts or using phrase operator to require query terms to occur close together) and *expanding* the query terms by including more synonyms (highlighted by shading the topic number).

Topic No./ Type of Change	Query with changes from *short keyword* to *final Boolean*, (**bolded** are insertions and ~~slashed~~ are removals).
352 both	#combine(#syn(**#1(**British Chunnel**)** **#1(Channel Tunnel)**) **#syn(#1(effect on) changes)** #syn(**#1(**#syn(British **UK**) **economy)** **#1(economic #syn(implications changes evaluation))**) ~~impact~~)
354 expand	#combine(#syn(**reporter newswriter** journalist **correspondent**) **#syn(arrested hostage #1(physical attack) killed threatened kidnapped murdered attack shot)** ~~risks~~)
704 restrict	#combine(**#1(**green party**)** **#syn(US #1(united states))** #syn(**#1(**political views**)** **politics**))
751 expand	#combine(scrabble #syn(players **group**) #syn(social **events**))
752 restrict	#combine(**#syn(location places countries)** dam removal Environmental impact **reason**)
753 restrict	#combine(bullying prevention programs **in schools #syn(classes assemblies discipline mediation projects) #syn(Students staff)**)
758 expand	#combine(embryonic stem cells #syn(restrictions **law policy**))
760 both	#combine(statistics **#1(in** America**)** Muslims **#syn(population demographics)** #syn(mosque **#band(Islamic center)**) school)
764 restrict	#combine(measures improve public transportation ~~increase mass transit use~~)
769 restrict	#combine(**#1(**Kroll Associates**)** employee names)
799 restrict	#combine(**type of animals** Alzheimer research))
805 restrict	#combine(identity theft passport **help victims identify establish credit worthiness show #syn(creditors #band(law enforcement))**)

Note: #combine is Indri's probabilistic AND operator, #syn is Boolean OR, #1 is the phrase operator, and #band is the Boolean AND.

4.2 Experiment Results

This section reports the characteristics of the user generated CNF queries, and the effectiveness of these CNF queries compared against the keyword query baselines.

4.2.1 Characteristics of the User CNF Queries

Query characteristics varied a lot across different users: 2 to 6 conjuncts for the *initial* Boolean queries, each conjunct containing 1 to 5 synonyms. The *final* Boolean queries were expanded a bit more, with 2 to 6 conjuncts, each containing 1 to 9 synonyms.

Table 1 shows how the users modified the original short keyword queries into the final Boolean CNF queries.

Users did not always follow the instruction to include expansion terms when formulating CNF queries. Only 5 of the 12 queries included some synonym expansion, while 9 out of the 12 queries were modified to be more restrictive than the keyword query. These queries are less well expanded than the queries created by expert searchers [2,14]. The sections below show how that affects retrieval performance.

4.2.2 Effectiveness – TREC Evaluation

Retrieving on the TREC datasets, the user created CNF queries are fairly effective overall (Table 2). On average, the final Boolean CNF queries perform the best on all three evaluation metrics. These final CNF queries perform significantly better than the long keyword queries both at top ranks (P@5, 10) and at overall accuracy (MAP@1000), and outperform on average the short keyword queries. This result is consistent with prior research, where shorter keyword queries perform better than long queries [13].

Even though CNF queries are better than short keyword queries, the difference is not statistically significant. We look at the individual queries to understand which CNF queries are better and which are worse than the corresponding short keyword query.

For expansion queries in Table 3 (topics 354, 751 and 758), topic 354 is the only one that decreases performance. The reason is that "reporter" is stemmed and matches the word "report", which is a common word in the TREC newswire collection, causing the expanded query to match many false positives.

For the restrictive Boolean queries, the performance gain is less stable. 5 (topics 752, 760, 764, 799, 805) out of the 9 restrictive Boolean queries perform worse than the short keyword query in MAP. Even in top precision, which is usually the users' goal for using more restrictive queries, 4 out of the 9 restrictive Boolean queries perform worse than short keyword queries.

This result on TREC datasets shows that many of the CNF expansion queries and a few of the restrictive Boolean queries created through interacting with a larger dataset (the Web) and very different retrieval algorithms can still effectively retrieve relevant documents on a smaller dataset. For expansion queries, this may be because on the one hand, accurate CNF expansions on larger collections are less likely to match false positives on smaller collections, ensuring precision. On the other hand, the mismatch problem is likely to get worse on the smaller collections with fewer relevant documents, thus, the CNF expansions are more likely to be useful in improving recall on the smaller collections. Overall, in both precision and recall, the CNF expansions created for larger collections may work well on the smaller collections. However, the more restrictive queries that perform well on large Web collections may not perform as well on smaller collections.

4.2.3 Effectiveness – Evaluation on Search Engines

The Boolean queries are clearly more effective than short keywords on the commercial Web search engines at top ranks, as shown in Table 4 (overall) and Table 5 (per topic).

Table 4 shows the evaluation on Google and Yahoo. On Google, final CNF queries significantly outperformed the short keyword queries in retrieval performance at top ranks. On Yahoo, CNF was on average better than short keyword, but the difference was not statistically significant, because 3 Boolean queries were worse than the short keyword baseline. (For topic 352 on Yahoo, the expansion phrase "channel tunnel" is too general and matches many false positives. For topic 354 on Yahoo, many false positives contain "Reporter", "Journalist" and "Correspondent" as titles of publications or newspapers, instead of as a person. For

Table 2. Retrieval performance on **TREC datasets**, averaged over the 12 topics. Bold-faced is the best run in each row.

\Query type Metrics\	Keyword (short)	Keyword (long)	CNF (initial)	CNF (final)
MAP@1000	0.1635^{l}	0.1038	0.1815^{l}	**0.2017**^{l}
P@5	0.5833^{l}	0.3167	0.5333^{l}	**0.6000**^{l}
P@10	0.5333^{l}	0.3000	0.5250^{l}	**0.5500**^{ln}

^{l} means the run is significantly better than the long keyword baseline by a two tailed t-test at $p < 0.05$.
^{n} means significantly better than long keyword by sign test at $p < 0.05$.

Table 3. Per topic retrieval performance of final Boolean CNF query vs. short keyword query on **TREC datasets**. Bold faced is the better result of keyword and CNF queries in the row.

TREC Topic No.	Keyword (short)		CNF (final) (vs. short keyword)			
	MAP	P@5	MAP	change	P@5	change
352	0.0462	**0.8**	**0.1175**	154.3%	0.6	-25.00%
354	**0.0542**	**0.4**	0.0328	-39.48%	0.0	-100.0%
704	0.2167	0.4	**0.3928**	81.26%	**0.8**	100.0%
751	0.1746	1.0	**0.2170**	24.28%	1.0	0.000%
752	**0.2237**	**1.0**	0.1574	-29.64%	0.8	-20.00%
753	0.3472	0.6	**0.4736**	36.41%	**1.0**	66.67%
758	0.3144	1.0	**0.3187**	1.368%	1.0	0.000%
760	**0.1609**	0.8	0.1279	-20.51%	**1.0**	25.00%
764	**0.1999**	**0.6**	0.0180	-91.00%	0.4	-33.33%
769	0.0143	0.0	**0.4588**	3108%	**0.6**	+inf
799	**0.1850**	**0.4**	0.0946	-48.87%	0.0	-100.0%
805	**0.0247**	0.0	0.0108	-56.28%	0.0	0.000%

Table 4. Retrieval performance with **search engine evaluation**, averaged over the 12 topics. Bold-faced are the better run(s) in each row.

\Query type \ & Metrics Search engine\	Keyword (short)		CNF (final)	
	MAP@5	P@5	MAP@5	P@5
Google	0.1586	0.6333	**0.2247**^{sn}	**0.8500**^{sn}
Yahoo	0.1701	**0.7167**	**0.2041**	**0.7167**

^{s} means significantly better than the short keyword baseline by a two tailed t-test at $p < 0.004$.
^{n} means also significant by sign test at $p < 0.004$.

Table 5. Per topic retrieval performance in MAP@5 for final Boolean query vs. short keyword query on **Google and Yahoo**. Bold faced is the better result of keyword and CNF queries.

TREC Topic No.	Keyword (short)		CNF (final)			
	Google	Yahoo	Google	change	Yahoo	change
352	0.1133	**0.1300**	**0.2000**	76.52%	0.0000	-100.0%
354	0.1462	**0.1538**	**0.1923**	31.53%	0.0192	-87.52%
704	0.3800	0.3800	**0.5000**	31.58%	**0.4000**	5.263%
751	0.0467	0.1600	**0.2000**	328.3%	**0.2000**	25.00%
752	0.1368	0.1693	**0.1693**	23.76%	**0.2000**	18.13%
753	0.0883	0.0800	**0.2500**	183.1%	**0.1900**	137.5%
758	0.2083	0.2083	0.2083	0.000%	0.2083	0.000%
760	0.2923	0.2923	**0.3846**	31.58%	**0.3077**	5.269%
764	0.0000	0.0200	**0.0833**	+inf	**0.2750**	1275%
769	0.0000	0.0464	**0.0179**	+inf	**0.1940**	318.1%
799	0.1786	**0.1786**	0.1786	0.000%	0.1429	-19.99%
805	0.3125	0.2219	0.3125	0.000%	**0.3125**	40.83%

topic 799 on Yahoo, the CNF query is only slightly worse, returning the only irrelevant result at rank 5.)

The user created Boolean queries outperform short keyword queries consistently, but different Boolean queries improve over the keyword queries for different reasons. The synonym expansion queries are better than keyword because they can solve the mismatch problems of the individual query terms in the keyword query. Topics 354, 751 and 758 are such examples. The restrictive type of Boolean queries outperforms keyword queries because the short keyword queries may match many false positives on the Web. A slightly more restrictive query can remove these false positives while still match enough relevant documents to fill up the top ranks. Topics 704, 753 and 769 are examples.

4.2.4 Discussion

When comparing CNF queries with short keyword queries, on TREC datasets, the difference is not very significant, however, on the search engines, CNF queries are consistently better.

This difference is likely because of two reasons. Firstly, when the users created the CNF queries, they tuned the queries by observing their retrieval results returned from the search engines. Thus, as long as the user makes a serious effort, the tuned Boolean queries will be better performing than the keyword queries on the search engines that the users tuned their queries on. Secondly, only top rank performance was observed and measured with the search engines. Since results deeper down the rank list were not available to the users, they would tend to create highly restrictive queries that improve top precision. This could explain why many of the Boolean queries were more restrictive versions of the short keyword queries. These restrictive queries would likely increase top precision on the search engines (which searched against very large corpora), but would likely decrease lower rank performance as suggested by the deeper evaluations on the TREC datasets. On the much smaller TREC corpora, these restrictive queries will match much fewer documents, thus could even hurt top precision. Overall, the more restrictive Boolean queries perform unstably at both top rank and lower rank levels on the smaller TREC datasets.

This suggests that even though it may seem to the user that a restrictive query would be better, more often than not, synonym expansion is the more robust strategy of query formulation.

5. CONCLUSIONS

Boolean CNF expansion queries have the potential to significantly outperform keyword queries, leading to much more effective retrieval. This paper investigates whether ordinary search users with limited knowledge of CNF queries can formulate effective CNF queries using the WikiQuery interface.

Evaluations on TREC datasets show that versus lengthening the short keyword queries by adding more keywords, creating a Boolean structured query can be significantly more effective at both top and deeper level retrieval accuracy. These Boolean queries are also better performing than the short keyword queries on average. However this difference is not statistically significant on the TREC datasets. Evaluations of the user created Boolean queries on commercial Web search engines show that these highly precise Boolean queries can consistently and significantly outperform the original short keyword queries in top precision.

Both expansion and restriction query modifications were common when the users created the Boolean queries from the short keyword queries. Some of the expansion queries included many synonyms for each original query term, just like those created by experts [14]. These carefully expanded CNF queries have been shown to outperform keyword queries in precision at all recall levels, because CNF expansion can effectively solve term mismatch, a common problem in retrieval with a large potential [14]. However, even with instructions and the guidance from WikiQuery, users still tended to create less well expanded queries. Users also tended to restrict the original keyword query by introducing phrases or more conjuncts, causing more mismatches between the query and the relevant documents. These restrictive queries might improve top precision, but deeper level evaluation on TREC datasets showed that these restrictive queries do not result in stable improvements at lower rank levels. This tendency to create the less effective restrictive-queries is perhaps one of the reasons why novice users have difficulty creating effective Boolean queries or structured queries.

Use in Text Retrieval

Our results suggest that to improve users' interactions with the search engine, and to facilitate them in creating effective queries, users need to be carefully guided to create CNF expansion queries, and to be explicitly warned against the risky restrictive queries.

Classroom Use

This work used WikiQuery as an educational tool for students with limited knowledge about Boolean queries to learn to create effective Boolean queries in a short time. We observe that most students spent about 40 minutes per topic, trying out new queries and interacting with the search results to find effective formulations. Trial and error using the interactive interface of WikiQuery helped the students quickly and effectively learn the subject.

Open source search tools like WikiQuery and IR education can be mutually beneficial. These tools may become the appropriate playground for educational uses, while classroom uses can also provide a steady stream of traffic for these search tools.

Future Work

The WikiQuery website is still in its early stage. This work as a pilot study can be used to guide and prioritize the development of many new and helpful features for WikiQuery.

Search result presentation needs to be improved to help users quickly grasp why a particular document is returned and what terms in each conjunct of the CNF query are present in the document. Such understandings will allow users to efficiently identify further refinements of the CNF query to improve the results. On a commercial search engine, such user interface changes would be deemed too risky. A research oriented search engine might be the best place to lead the effort.

To further facilitate users in their CNF query creation, synonyms or other related words of each conjunct could be automatically suggested to the user, so that the user only needs to select the highly precise expansion terms out of the suggestions.

To better facilitate users in query refinement, novel interfaces that can automatically extract or highlight candidate expansion terms in result snippets or documents can be useful.

To help users decide whether to include one particular expansion term into a conjunct or not, tools that can compare rank list changes before and after a query change will be useful. In particular, tools that can present deeper rank level changes will enable the user to more accurately gauge the overall retrieval accuracy.

The WikiQuery website may also allow users to subscribe to the result pages of each CNF query, so that whenever a new relevant page appears on the Web, the user will be notified. This is the equivalent of a traditional routing task, and can be easy implemented given that search engines like Google already support user subscription to search engine result pages.

6. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, 1999.

[2] J. R. Baron, D. D. Lewis and D. W. Oard. TREC 2006 Legal Track Overview. In *Proceedings of the fifteenth Text REtrieval Conference (TREC '06)*, 2007.

[3] D. C. Blair and M. E. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Communications of ACM*, 28(3): 289-299. 1985.

[4] C. L. A. Clarke, G. V. Cormack and F. J. Burkowski. Shortest Substring Ranking (MultiText Experiments for TREC-4). In *Proceedings of the Fourth Text REtrieval Conference (TREC-4)*. 1996.

[5] H. A. Feild, J. Allan and R. Jones. Predicting searcher frustration. In *Proceedings of 33rd annual international ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR '10)*. 34-41, 2010.

[6] S. Harter. *Online Information Retrieval: Concepts, Principles, and Techniques*. Academic Press. San Diego, California. 1986.

[7] M. Hearst. Improving full-text precision on short queries using simple constraints. In *Proceedings of the Fifth Annual Symposium on Document Analysis and Information Retrieval (SDAIR '96)*, 1996.

[8] M. Hearst. *Search User Interfaces*. Cambridge University Press. 2009.

[9] W. Lancaster. *Information Retrieval Systems: Characteristics, Testing and Evaluation*. Wiley. New York, New York, USA. 1968.

[10] C. D. Manning, P. Raghavan and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press. 2008.

[11] M. Mitra, A. Singhal and C. Buckley. Improving automatic query expansion. In *Proceedings of 21st annual international ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR '98)*. 206-214, 1998.

[12] P. Reisner. Construction of a growing thesaurus by conversational interaction in a man-machine system. *Proceedings of the American Documentation Institute*, 26th annual meeting, Chicago, Illinois, 1963.

[13] L. Zhao and J. Callan. Term necessity prediction. In Proceedings of the *19th ACM Conference on Information and Knowledge Management*. 2010.

[14] L. Zhao and J. Callan. Automatic term mismatch diagnosis for selective query expansion. To appear *Proceedings of 35th annual international ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR '12)*. 2012.

ezDL: An Interactive Search and Evaluation System

Thomas Beckers
Information Engineering
University of Duisburg-Essen
Duisburg, Germany
thomas.beckers@uni-due.de

Sebastian Dungs
Information Engineering
University of Duisburg-Essen
Duisburg, Germany
sebastian.dungs@uni-due.de

Norbert Fuhr
Information Engineering
University of Duisburg-Essen
Duisburg, Germany
norbert.fuhr@uni-due.de

Matthias Jordan
Information Engineering
University of Duisburg-Essen
Duisburg, Germany
matthias.jordan@uni-due.de

Sascha Kriewel
Information Engineering
University of Duisburg-Essen
Duisburg, Germany
sascha.kriewel@uni-due.de

ABSTRACT

The open-source system *ezDL* is presented. It is an interactive search tool, a development platform for interactive IR systems, and an evaluation system. *ezDL* can be used as a meta-search system for heterogeneous sources or digital libraries, allows organizing and filtering of merged results, offers support for search sessions as well as a personal library for storing different document types. The *ezDL* framework is easy to extend and is based on a service-oriented architecture. In addition, support for performing user studies and eye tracking is provided. *ezDL* has been used as a system in several funded research projects.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Human Factors, Experimentation

Keywords

interactive search system, framework

1. INTRODUCTION

In this paper we present *ezDL*, an open-source[1] software for building highly interactive search user interfaces with strategic support. It builds on the ideas developed and implemented within the Daffodil project from 2000 to 2009 [11, 16, 13], but uses more modern software technologies and interface design methods.

The *ezDL* framework can be characterized by three main purposes. It is foremost *i*) a working interactive tool for searching a heterogeneous collection of digital libraries. In addition to that, it is *ii*) a flexible software platform providing a solid base for writing customized applications as well as *iii*) a system that can be used for many different types of user evaluations.

Today many systems covering one or more aspects of *ezDL* exist but to the best of our knowledge the concept of unifying them into one single framework is unique. In the following paragraphs similar systems related to the different aspects of *ezDL* are presented.

Interactive Search Tools

Querium [9] is an interactive search system featuring a concept that focuses on complex recall oriented searches. It aims at preserving the context of searches and allows relevance feedback to generate alternative result sets. At the moment, the system is limited to two data sources.

Numerous tools exist that focus on storing and managing a personal library or citations. A popular example is Mendeley[2] which also offers different front ends and collaborative features. Other citation tools include CiteULike[3] and Connotea[4]

CoSearch [1] is a collaborative web search tool. It offers a user interface that can simultaneously take input from different users sharing a single machine. Mobile devices can be used to contribute to a collaborative search session. Data is acquired by using a popular web search engine.

Development Platforms

SpidersRUs Digital Library Toolkit [8] is a search engine development tool. The developers strove for a balance between easiness of use and customizability. The toolkit also features

[1] *ezDL* is licensed under GPL v3. Other licenses can be used on request. The main web site for developers and further information can be found here: `http://ezdl.de/developers`

[2] `http://www.mendeley.com/`
[3] `http://www.citeulike.org/`
[4] `http://www.connotea.org/`

SIGIR 2012 Workshop on Open Source Information Retrieval (OSIR 2012)
Portland, Oregon, USA

a GUI for the process of search engine creation. Results presentation follows common standards of popular web search engines. Support for complex search sessions, e.g. a tray or citation management tool are not included.

Evaluation Systems

The Lemur project[5] includes a query log tool bar that can be used to capture usage data. It can collect queries as well as user interaction such as mouse activity and is available as open source.

Bierig et al [7] presented an evaluation and logging framework for user-centered and task-based experiments in interactive information retrieval that focuses on "multidimensional logging to obtain rich behavioural data" of searchers.

2. CONCEPTS

As a re-implementation of the Daffodil project [11], *ezDL* builds on many of the same concepts and principles as Daffodil. Like Daffodil it is a "search system for digital libraries aiming at strategic support during the information search process" [16]. Its primary target group is not that of casual users using a search system for short ad-hoc queries. Instead the software aims to support searchers during complex information tasks by addressing all the steps in the *Digital Library Life Cycle*, as well as integrating search models originally proposed by Marcia Bates [2, 3, 4].

The *Digital Library Life Cycle* divides the information workflow into five phases [18], beginning with the *discovery* of information resources, which in *ezDL* is supported through the Library Choice view. This is followed by the *retrieval* phase of information search, the *collating* of found information using the personal library and tagging, *interpreting* the information, and finally the *re-presenting* phase where new information is generated. In all phases, different so-called tactics or stratagems can be employed by searchers or information workers, which we try to support through *ezDL*.

The notion of tactics and stratagems as higher-level search activities was introduced by Bates [2, 3, 4]. Based on search tactics used by librarians and expert searchers, Bates describes basic *moves*, as well as higher-level *tactics*, *stratagems*, and finally *strategies* that build on lower-level activities.

ezDL already offers direct support for some of those higher-level activities, e.g. through the use of sharing functionalities to support collaborative idea generation, through term suggestions of synonyms or spelling variants, extraction of common results terms, or through icons in the result items that allow easy monitoring of performed activities.

During query formulation, *ezDL* provides term suggestions to the user (e.g. synonyms and related terms). These are an example for the concept of proactive system support. Bates describes "five levels of system involvement (SI) in searching" [4]. The proactive support of *ezDL* belongs to the third level, where a search system (through monitoring of user activities) can react to the search situation without prompting by the user. Users are informed of improvement options for their current move. Jansen and Pooch [12] demonstrated that proactive software agents assisting users during their search can result in improved performance of users. The effectiveness of such suggestions has also been shown for the Daffodil system [20].

[5]`http://www.lemurproject.org/`

Tran [21] implemented a prototype of a support tool for the pearl growing stratagem. The tool shows citation relationships between documents in a graph and allows the user to follow these relationships and keep track of the search progress using document annotations. Figure 1 shows a screenshot of a pearl growing session with some documents marked as relevant. It is planned to include this tool in *ezDL* in the near future.

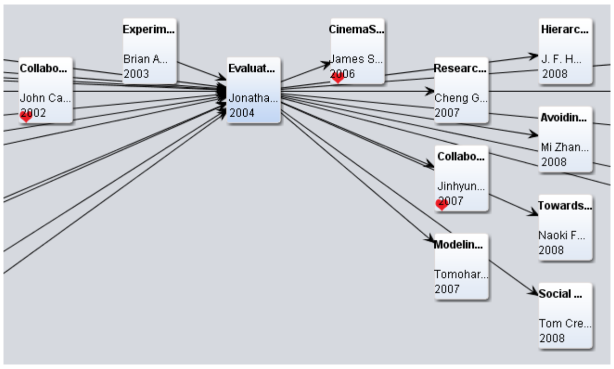

Experim... Brian A... 2003
Collabo... John Ca... 2002
Evaluat... Jonatha... 2004
CinemaS... James S... 2006
Researc... Cheng G... 2007
Collabo... Jinhyun... 2007
Modelin... Tomohar... 2007
Hierarc... J. F. H... 2008
Avoidin... Mi Zhan... 2008
Towards... Naoki F... 2008
Social ... Tom Cre... 2008

Figure 1: A close-up of the pearl growing tool

Proactive support of higher-level activities, such as suggestion of tactics and stratagems for improvable search situations [14, 15] or suggestion of search strategies with scaffolding support, is currently planned and will likely be available for *ezDL* within the next nine months.

3. ARCHITECTURE

ezDL is a continuation of the Daffodil project and therefore shares its main ideas: meta-search in digital libraries and strategic support for users. Its overall architecture likewise has inherited many features from Daffodil. Figure 2 provides a high-level overview of the system.

The system architecture makes extensive use of separation of concerns to keep interdependencies to a minimum and make the system more stable. This is true on the system level where a clear separation exists between clients and backend, but also within the backend itself, where individual "agent" processes handle specific parts of the functionality, and even within these agents. The desktop client, too, is separated into multiple independent components called "tools". *ezDL* is completely written in Java using common frameworks and libraries.

3.1 The Backend

The backend provides a large part of the core functionality of *ezDL*: the meta-search facility, user authorization, a knowledge base about collected documents, as well as wrappers and services that connect to external services. Functionality that provides collaboration support and allows storing of documents and queries in a personal library is also located here.

The right part of Figure 2 shows the structure of the backend. The components of the backend are agents: independent processes that provide a specific functionality to the system. Agents use a common communication bus for transferring messages between each other.

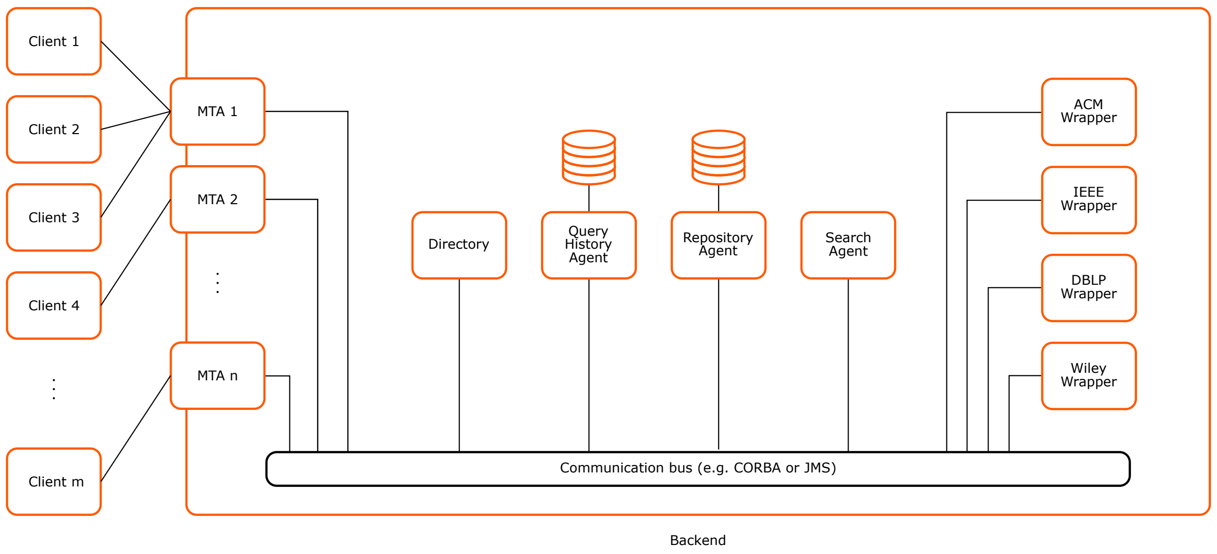


Figure 2: Overall architecture of *ezDL*

The Service-based Agent Infrastructure

Since every kind of functionality is taken care of by different agents, the crash of one agent generally only disrupts this particular functionality. For example, if the search agent crashes, detail requests and the personal library are still working. Also, it is possible to run multiple agents of each kind for load balancing and as a fail-safe mechanism.

Agents are subdivided into the main agent behaviour (registering with the Directory, sending and receiving messages, managing resources) and components that deal with specific requests. These components—the request handlers—are independent and process requests concurrently.

Beginning on the left, an MTA (Message Transfer Agent) is an agent that provides clients with a connection point to the backend. MTAs are responsible for authenticating users and translating requests from clients into messages to certain agents. E.g., if a client requests a search for a given query, the query from the client is translated into a message to the Search Agent. This mechanism creates a clear separation between the client view of the system and the internal workings: the client doesn't have to know how many agents are serving search queries and new search agents could be instantiated as the system load demands. Currently there is only one MTA implementation, which uses a binary protocol over a TCP connection, but it is possible to provide other protocols—e.g. SOAP—by using separate MTA implementations.

The Directory is a special agent that keeps a list of agents and the services they provide. Upon start, each agent registers with the Directory and announces the services it provides.

The connection to remote (or local) search services (e.g., digital libraries or information retrieval systems) is managed by wrapper agents—in Figure 2 the four agents on the right hand side. They translate the internal query representation of *ezDL* into one that the remote service can parse and translates the response of the remote service into an appropriate document representations to be handled by *ezDL*.

Example: Running Search Queries

If a client requests a search, it sends a request to the MTA with a query in *ezDL* notation and a list of remote services that the query should be run on. The request is handled by the MTA which forwards it to the Search agent. The Search agent asks the Directory for the name of agents that provide a connection to the remote services requested by the client. After receiving that list, the Search agent forwards the query to each of these agents. The agents then translate the query into something that the remote service understands and sends the answer of the remote service back to the search agent. The search agent collects all answers from all the remote services, merges duplicates and reranks them. Reranking is either performed by using the original RSVs or by using standard Lucene[6] functionality. The answer set is then sent back to the MTA that requested the search. The MTA sends the answer to the client. The search agent also forwards the collected documents to the repository agent which is responsible for serving requests for details on documents (e.g., if the user wants to see the full text).

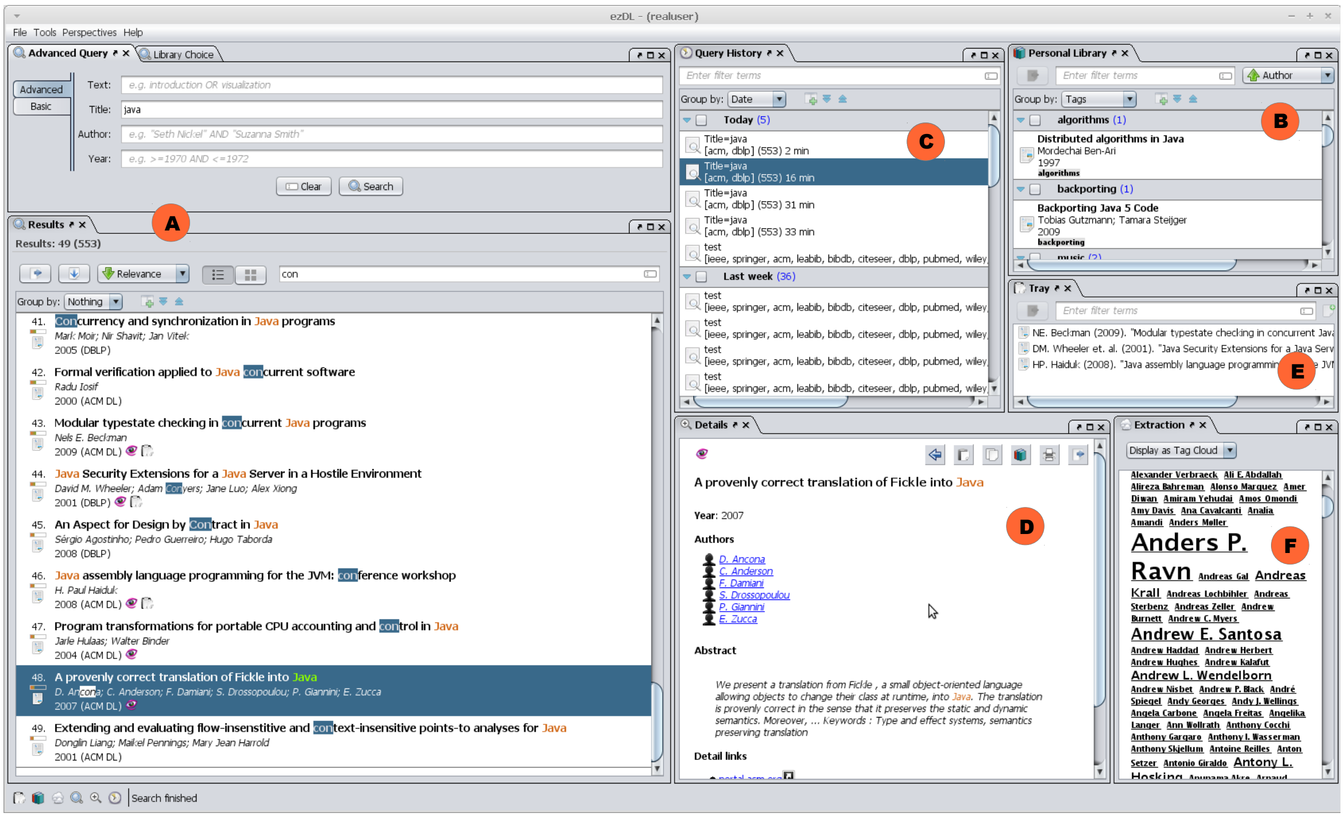
3.2 The Frontend

There are multiple frontends for *ezDL*: among them the basic desktop client and a web client. Specialized frontends exist for various applications (see Use Cases). Clients for iOS and Android tablets are currently being developed. This subsection details the architecture of the desktop client, since this is the main client for *ezDL*.

Tools and Perspectives

A *tool* comprises a set of logically connected functionalities. Each tool has one or more *tool views*, interactive display components that can be placed somewhere on the desktop. A configuration of available tools and the specific layout of their tool views on the desktop is called a *perspective*. Users can modify existing predefined perspectives as well as create custom perspectives. The desktop client already has many

[6]`http://lucene.apache.org/core/`


ezDL - (realuser)

File Tools Perspectives Help

Advanced Query | Library Choice

Advanced / Basic

Text: e.g. introduction OR visualization
Title: java
Author: e.g. "Seth Nickel" AND "Suzanna Smith"
Year: e.g. >=1970 AND <=1972

Clear Search

Results A

Results: 49 (553)

Relevance | con

Group by: Nothing

41. Concurrency and synchronization in Java programs
Mark Moir; Nir Shavit; Jan Vitek
2005 (DBLP)

42. Formal verification applied to Java concurrent software
Radu Iosif
2000 (ACM DL)

43. Modular typestate checking in concurrent Java programs
Nels E. Beckman
2009 (ACM DL)

44. Java Security Extensions for a Java Server in a Hostile Environment
David M. Wheeler; Adam Conyers; Jane Luo; Alex Xiong
2001 (DBLP)

45. An Aspect for Design by Contract in Java
Sérgio Agostinho; Pedro Guerreiro; Hugo Taborda
2008 (DBLP)

46. Java assembly language programming for the JVM: conference workshop
H. Paul Haiduk
2008 (ACM DL)

47. Program transformations for portable CPU accounting and control in Java
Jarle Hulaas; Walter Binder
2004 (ACM DL)

48. A provenly correct translation of Fickle into Java
D. Ancona; C. Anderson; F. Damiani; S. Drossopoulou; P. Giannini; E. Zucca
2007 (ACM DL)

49. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java
Donglin Liang; Maikel Pennings; Mary Jean Harrold
2001 (ACM DL)

Search finished

Query History C

Enter filter terms

Group by: Date

Today (5)
Title=java [acm, dblp] (553) 2 min
Title=java [acm, dblp] (553) 16 min
Title=java [acm, dblp] (553) 31 min
Title=java [acm, dblp] (553) 33 min
test [ieee, springer, acm, leabib, bibdb, citeseer, dblp, pubmed, wiley

Last week (36)
test [ieee, springer, acm, leabib, bibdb, citeseer, dblp, pubmed, wiley
test [ieee, springer, acm, leabib, bibdb, citeseer, dblp, pubmed, wiley
test [ieee, springer, acm, leabib, bibdb, citeseer, dblp, pubmed, wiley
test [ieee, springer, acm, leabib, bibdb, citeseer, dblp, pubmed, wiley

Personal Library B

Enter filter terms | Author

Group by: Tags

algorithms (1)
Distributed algorithms in Java
Mordechai Ben-Ari
1997
algorithms

backporting (1)
Backporting Java 5 Code
Tobias Gutzmann; Tamara Steijger
2009
backporting

music (2)

Tray E

Enter filter terms

NE. Beckman (2009). "Modular typestate checking in concurrent Java
DM. Wheeler et. al. (2001). "Java Security Extensions for a Java Serv
HP. Haiduk: (2008). "Java assembly language programming for the JVM

Details D

A provenly correct translation of Fickle into Java

Year: 2007

Authors
D. Ancona
C. Anderson
F. Damiani
S. Drossopoulou
P. Giannini
E. Zucca

Abstract

We present a translation from Fickle , a small object-oriented language allowing objects to change their class at runtime, into Java. The translation is provenly correct in the sense that it preserves the static and dynamic semantics. Moreover, ... Keywords : Type and effect systems, semantics preserving translation

Detail links

Extraction F

Display as Tag Cloud

Alexander Verbraeck Ali E. Abdallah Alireza Bahreman Alonso Marquez Amer Diwan Amiram Yehudai Amos Omondi Amy Davis Ana Cavalcanti Analia Amandi Anders Moller Anders P. Ravn Andreas Gal Andreas Krall Andreas Lochbihler Andreas Sterbenz Andreas Zeller Andrew Burnett Andrew C. Myers Andrew E. Santosa Andrew Haddad Andrew Herbert Andrew Hughes Andrew Kalafut Andrew L. Wendelborn Andrew Nisbet Andrew P. Black André Spiegel Andy Georges Andy J. Wellings Angela Carbone Angela Freitas Angelika Langer Ann Wollrath Anthony Cocchi Anthony Gargare Anthony I. Wasserman Anthony Skjellum Antoine Reilles Anton Setzer Antonio Giraldo Antony L. Hosking

Figure 4: The Desktop client during a search session

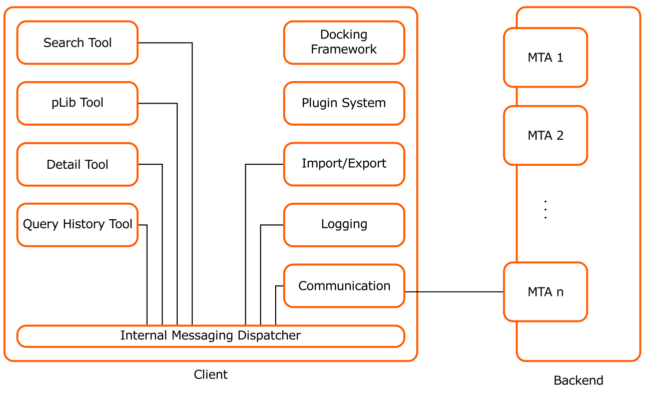


Figure 3: Architecture of the desktop client

built-in tools and functionalities and can be easily extended (see Figure 4):

- The Search Tool (A) offers a variety of query forms for different purposes and views to present the results in list or grid form, as well as a Library Choice view for selecting information sources. Results can be sorted or grouped by different criteria, filtered, and exported. An extraction function (F) can be used to extract frequent terms, authors, or other features from the result and visualize them in form of a list, a bar chart or a term cloud. Grouping criteria or extraction strategies are encapsulated and new ones can easily be added, as can be new renderers for result surrogates.

- The Personal Library (B) allows to store documents or queries persistently for authenticated users. Within their personal collection, users can filter, group and sort (e.g. by date of addition), organize the documents with personal tags, and share them with other users. Additional documents can be imported into the personal library as long as their metadata is available in BibTeX format.

- The Search History (C) lists past queries for re-use and allows grouping by date and filtering.

- The Detail View (D) shows additional details on individual documents, such as thumbnails or short summaries where available, or additional metadata not included in the surrogate that is shown in the result list. A detail link can be provided to retrieve the fulltext.

- A Tray (E) can be used to temporarily collect relevant documents within a search session.

Communication with the Backend

Like the backend, the desktop client uses a messaging infrastructure for communication between otherwise independent components. In Figure 3 a diagram of the components is shown. On the left, four of the available tools can be seen with their connection to the internal communication infrastructure (search, personal library, details, and query

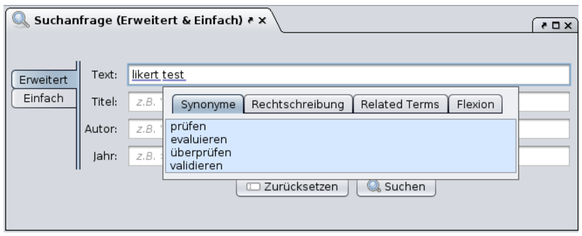


Figure 5: The search form with suggestions

history). On the right hand side, a few subsystems are presented, one of which is the external communication facility that connects the client to the backend.

As an example, if the user enters a query in the search tool and presses the "search" button, an internal message is sent to the communication facility, which transmits the query to the backend. When the answer is received, the communication facility routes the message back to the search tool.

Since from the client's point of the view the backend is hidden behind the MTA, further details are omitted in the backend part of Figure 3.

Query Processing and Proactive Support

The queries that users enter are expressed in a grammar specific to *ezDL* that is quite flexible and allows simple queries like `term1 term2 term3` as well as more complicated ones like `Title=term1 AND (term2 NEAR/2 term3)`. Internally, the query is represented as a tree structure that can also keep images as comparison values so *ezDL* can be used to specify image search queries.[7]

During query formulation, the user's interaction is observed by the system. If the system notices a break in the user's typing, the query is processed by modules of the proactive support subsystem that can either ask the backend for suggestions or calculate them directly in the frontend. The suggestions can replace query terms e.g. by spelling corrections, insert new terms (e.g., for synonym expansion), or tag terms with concepts from an ontology. The ontology items become part of the query so that a query can contain both plain text terms and ontology terms. When suggestions are found for a term, the term is marked by an underline in the query text field and a popup list is shown that presents the suggestions (see Figure 5).

3.3 Extending and Customizing ezDL

Each agent and the desktop client are extensible using a plugin system. Plugins are registered at a central component that can later be asked to return plugin objects of a specific type. As an example, it is possible to add a new proactive suggestion module to the system that implements a new way of retrieving suggestions. Also the popup list that shows the suggestions can be replaced by an alternative. Further uses of the plugin system are export and import modules and modules that extract information from the result list.

Adding a new service is usually done by implementing a new agent. There is an abstract class that takes care of most issues but the actual functionality. This is usually implemented using specialized classes (request handlers) for which there are abstract implementations, too. Thus, developers can concentrate on the business logic.

Connecting to a new collection for searching (a digital library, a local IR system, a BibTeX file, etc.) is accomplished by implementing a wrapper agent. These are agents specialized in translating between *ezDL* and a remote system. Remote systems can be those that provide a stable API like SOAP or SQL but also those that only have a web site and a search form. *ezDL* has built-in support for most common fields (e.g. title, author, publication year, abstract) and data types (e.g. text, numbers, images). There are abstract wrappers available to quickly connect to a Solr[8] server. If required, web pages can be scraped using an elaborate tool kit that is configured by an XML file. Because of this, even digital libraries without a proper API can be connected. Sometimes, digital libraries change their web page layout, breaking scripts that parse their HTML. Configuring the page scraping using an XML file makes automatic repairs of the configuration possible. See [10] for an example implementation based on Daffodil, The approach outlined in this work uses repeated queries to infer the template elements of the web pages and step-wise generalisation to find the location of known information on the page.

There is also a library of code for translating the *ezDL* query representation into other languages.

Agents—and, thus, wrapper agents—announce themselves to the Directory agent when started. The client can ask the backend for a list of known wrapper agents, so there is no need to change any code or configuration outside of the agent. This also enables developers to store the code and put it under version control independent from the main *ezDL* code.

Often, services in the back end are used in the client in an individual tool. One example for this is the search facility, which consists of the search tool in the Swing client and the search agent in the back end. Writing a new tool for the Swing client can be done by implementing an OSGi plugin. The tool code itself is fairly simple since there is an abstract implementation for the glue code. The remaining task is implementing a Swing GUI and communicating with the back end by firing events and listening for an answer.

4. EVALUATING SEARCH SYSTEMS

To support user-centred evaluations, *ezDL* has a builtin evaluation mode that addresses many of the major challenges inherent in setting up evaluation tasks and tracking user activity during the experiments. The following is a brief overview of those functionalities within *ezDL* directly designed to support evaluations.

Logging user actions

For evaluations with actual users all user actions performed with the system should be logged for later inspection and analysis. *ezDL* has a built-in logging facility that stores all the interaction data of the user in a relational database (currently *mysql* is used). A *log session* comprises all *log events* that a user or the system has triggered. A log event has *i*) a unique name identifying this type of event, *ii*) timestamps from the frontend and the backend, *iii*) a sequence number

[7]This mechanism is used in the Khresmoi project (see Section 5) to allow general physician and radiologists to search for medical images.

[8]`http://lucene.apache.org/solr/`

to ensure the correct order, and *iv*) parameters as multiple key/value pairs. For example, when a user performs a search for `information retrieval` in the DBLP and ACM digital library the corresponding log event may look like this:

```
event:
  name: "search"
  clientTimestamp: 1/4/2012 15:26:32,1234
  timestamp: 1/4/2012 15:26:32,3456
  sequenceNumber: 10
  parameters:
      query: "information retrieval"
      sources: dblp, acm
```

The logging facility takes care about allocating activities to sessions and users. If it is required to log some previously unlogged action, this can be simply integrated by sending a corresponding logging message to the backend.

Tracking AOIs

Gaze tracking is a method for user-centred evaluation that has recently gained popularity within the IR field [5, 7]. A challenge for logging highly interactive systems with changing interfaces and moving components is keeping track of their position, so that gaze points or fixation data of users can be aggregated across so-called *Areas of Interest* (AOIs, see Figure 6). This feature has been integrated into *ezDL* with the help of the *AOILog* framework [6].

Fixed layout on screen

The layout of the desktop can be locked to keep UI-related variance low. With a fixed layout it is no longer possible for a test subject to open additional tool views or change the layout of the desktop.

Loading predefined perspectives

Predefined perspectives can be loaded immediately after the system has been started. This allows the evaluator to create custom perspectives that can be used for an evaluation without selecting them manually.

Splash screen for choosing evaluation settings

A splash screen can be enabled that is shown before starting the system. It can be used to choose and set settings for the evaluation session, e.g. a search task description or the system variant when doing a comparison of different UIs or system features.

Several user studies have been performed and experimental systems implemented using *ezDL* as a base system. The next section will present some of them in more detail.

5. USE CASES

ezDL is currently running as a live system, and is being used and extended in a number of projects of various sizes.

The live system[9] features all core functionalities that part of a more specific project. These include a simple and an advanced search function, various result manipulation options, a temporary document store, and exporting of meta information (e.g. in BibTeX format). Registered users can also use a personal library to store, annotate and share found or imported documents. Currently, nine different digital libraries are connected to the system focusing on computer

[9]`http://www.ezdl.de/`

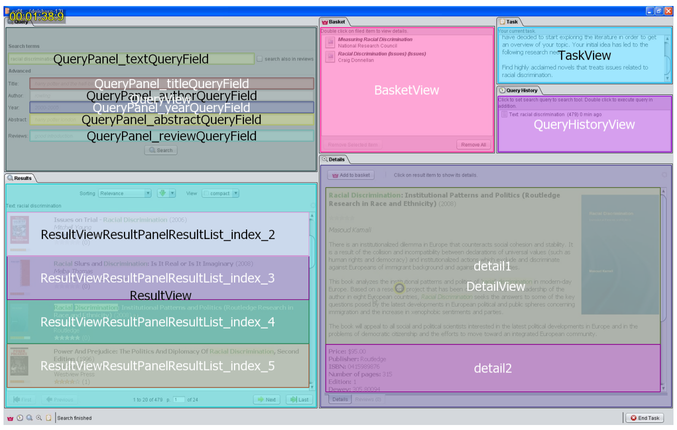

00:01:38.9
QueryPanel_textQueryField
QueryPanel_titleQueryField
QueryPanel_authorQueryField
QueryView
QueryPanel_yearQueryField
QueryPanel_abstractQueryField
QueryPanel_reviewQueryField
BasketView
TaskView
QueryHistoryView
ResultViewResultPanelResultList_index_2
ResultViewResultPanelResultList_index_3
ResultView
ResultViewResultPanelResultList_index_4
ResultViewResultPanelResultList_index_5
detail1
DetailView
detail2

Figure 6: An *ezDL* client overlayed with AOIs for eye tracking (from the HIIR project)

science libraries, but including others like Pubmed and the Amazon catalogue. The system is publicly available, still under constant development and updated regularly.

Khresmoi[10] is a four year project funded by the EC, which aims at building a multilingual and multimodal search system for biomedical documents. The *ezDL* framework is used for all user interfaces developed within the project. These include variations of the stand-alone Swing client, such as an interface for search medical images, including 3D data. Another version of the interface will be customized to the needs of general practitioners. For casual users with health related information needs, an easy to use web based interface (see Figure 7) is under development. From a functional point of view numerous new data sources were made available. The set of searchable data types was extended to cover the specific demands of the medical domain. The system allows the user to specify an image as a query to perform a similarity search. Additional collaborative and social functions will be added to the full client in later versions.

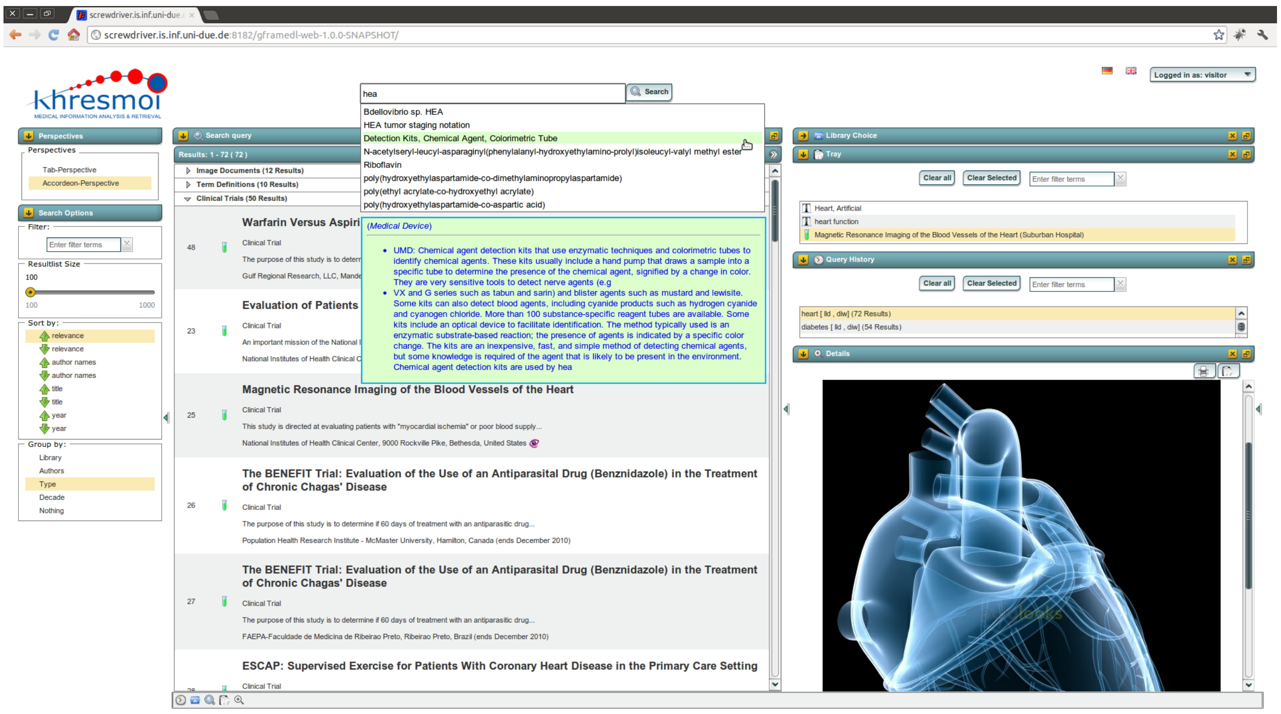
Within the INEX 2010 Interactive Track *ezDL* was used to observe how users act during their search sessions [19]. Valuable insights on user behaviour were gained. An application for viewing the logged data and a questionnaire tool controlling the experiment flow have been implemented.

For the ongoing CAIR[11] project an advanced 2010 INEX *ezDL* version is used. To answer the question whether clustering of results can improve efficiency of searches with vague information needs *ezDL* was expanded by a clustering service and visualization [17] (see Figure 8). New data sources and a browser were added to the system. For the evaluation a task selection and a questionnaire tool were developed. Log data generated by *ezDL* can be analysed automatically.

The AOI logging framework mentioned in Section 4 was implemented as part of the HIIR (Highly Interactive IR) project[12]. The project's goal is improving interaction with the system by considering the users cognitive efforts [23, 22].

6. CONCLUSION AND OUTLOOK

[10]`http://www.khresmoi.eu/`
[11]`http://www.uni-weimar.de/cms/index.php?id=17632`
[12]`http://www.is.inf.uni-due.de/projects/hiir/index.html.en`



Bdellovibrio sp. HEA
HEA tumor staging notation
Detection Kits, Chemical Agent, Colorimetric Tube
N-acetylseryl-leucyl-asparaginyl(phenylalanyl-hydroxyethylamino-prolyl)isoleucyl-valyl methyl ester
Riboflavin
poly(hydroxyethylaspartamide-co-dimethylaminopropylaspartamide)
poly(ethyl acrylate-co-hydroxyethyl acrylate)
poly(hydroxyethylaspartamide-co-aspartic acid)

(Medical Device)

- UMD: Chemical agent detection kits that use enzymatic techniques and colorimetric tubes to identify chemical agents. These kits usually include a hand pump that draws a sample into a specific tube to determine the presence of the chemical agent, signified by a change in color. They are very sensitive tools to detect nerve agents (e.g
- VX and G series such as tabun and sarin) and blister agents such as mustard and lewisite. Some kits can also detect blood agents, including cyanide products such as hydrogen cyanide and cyanogen chloride. More than 100 substance-specific reagent tubes are available. Some kits include an optical device to facilitate identification. The method typically used is an enzymatic substrate-based reaction; the presence of agents is indicated by a specific color change. The kits are an inexpensive, fast, and simple method of detecting chemical agents, but some knowledge is required of the agent that is likely to be present in the environment. Chemical agent detection kits are used by hea

48 Warfarin Versus Aspiri
Clinical Trial
The purpose of this study is to deterr
Gulf Regional Research, LLC, Mande

23 Evaluation of Patients
Clinical Trial
An important mission of the National I
National Institutes of Health Clinical C

25 Magnetic Resonance Imaging of the Blood Vessels of the Heart
Clinical Trial
This study is directed at evaluating patients with "myocardial ischemia" or poor blood supply...
National Institutes of Health Clinical Center, 9000 Rockville Pike, Bethesda, United States

26 The BENEFIT Trial: Evaluation of the Use of an Antiparasital Drug (Benznidazole) in the Treatment of Chronic Chagas' Disease
Clinical Trial
The purpose of this study is to determine if 60 days of treatment with an antiparasitic drug...
Population Health Research Institute - McMaster University, Hamilton, Canada (ends December 2010)

27 The BENEFIT Trial: Evaluation of the Use of an Antiparasital Drug (Benznidazole) in the Treatment of Chronic Chagas' Disease
Clinical Trial
The purpose of this study is to determine if 60 days of treatment with an antiparasitic drug...
FAEPA-Faculdade de Medicina de Ribeirao Preto, Ribeirao Preto, Brazil (ends December 2010)

28 ESCAP: Supervised Exercise for Patients With Coronary Heart Disease in the Primary Care Setting
Clinical Trial

Heart, Artificial
heart function
Magnetic Resonance Imaging of the Blood Vessels of the Heart (Suburban Hospital)

heart [lld , diw] (72 Results)
diabetes [lld , diw] (54 Results)

Figure 7: The web client used in the Khresmoi project

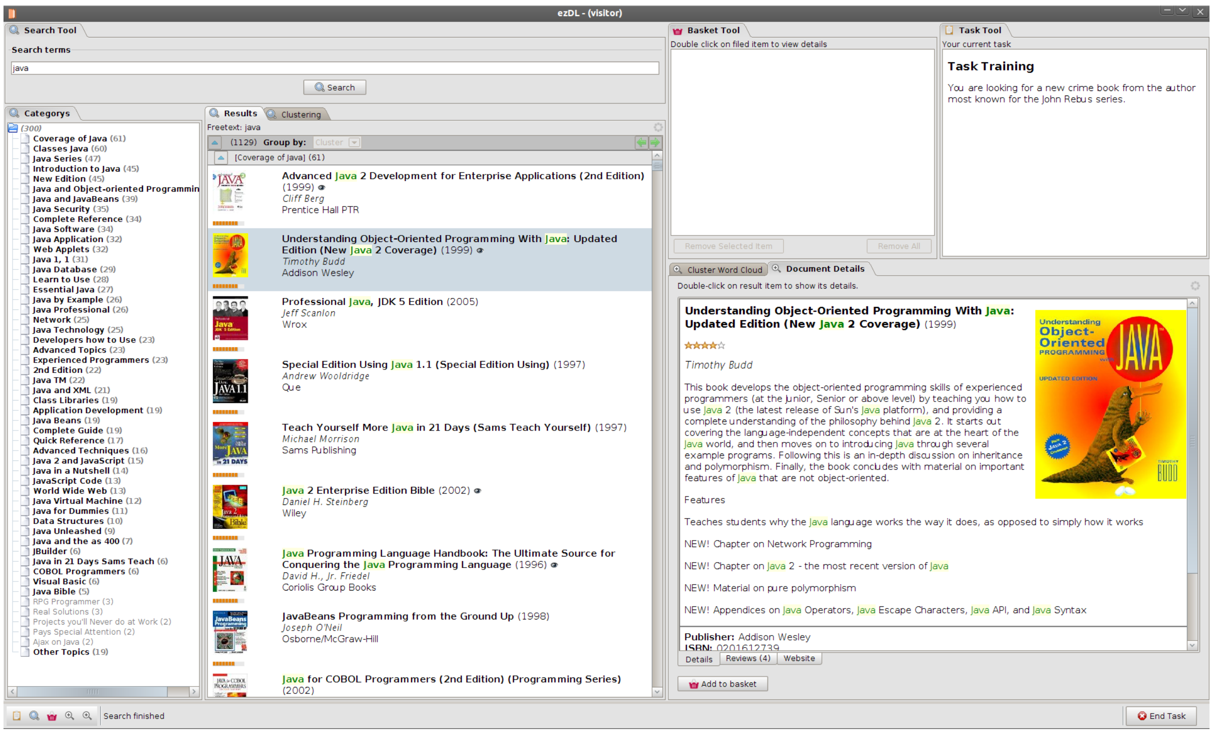
We presented *ezDL*, which is a framework system for interactive retrieval and its evaluation. Building upon state-of-the art interface technology and usability results, *ezDL* can provide an advanced user interface for many IR applications. The system can also be easily extended, at the functionality level as well as at the presentation level; thus, new concepts for the design of IR user interfaces can be integrated into *ezDL* with little effort. Furthermore, the system provides extensive support for performing user-oriented evaluations. In the same way as there are various experimental IR backend systems, there is now an IR frontend system that allows for easy experimentation and application of interactive retrieval.

Acknowlegments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement №257528 (KHRESMOI). Parts of this work were supported by the German Science Foundation under grant nos. FU 205/24-1 (HIIR) and FU 205/22-1 (CAIR).

7. REFERENCES

[1] S. Amershi and M. R. Morris. Cosearch: a system for co-located collaborative web search. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1647–1656, New York, NY, USA, 2008. ACM.
[2] M. J. Bates. Idea tactics. *Journal of the American Society for Information Science*, 30(5):280–289, 1979.
[3] M. J. Bates. Information search tactics. *Journal of the American Society for Information Science*, 30(4):205–214, 1979.
[4] M. J. Bates. Where should the person stop and the search interface start? *Information Processing and Management*, 26(5):575–591, 1990.
[5] T. Beckers and N. Fuhr. User-oriented and eye-tracking-based evaluation of an interactive search system. In *4th Workshop on Human-Computer Interaction and Information Retrieval (HCIR 2010) @ IIiX 2010*, 2010.
[6] T. Beckers and D. Korbar. Using eye-tracking for the evaluation of interactive information retrieval. In *Proceedings of INEX 2010*, pages 236–240, 2011.
[7] R. Bierig, J. Gwizdka, and M. Cole. A User-Centered Experiment and Logging Framework for Interactive Information Retrieval. In *Understanding the user - workshop in conjuction with SIGIR'09*, 2009.
[8] M. Chau and C. H. Wong. Designing the user interface and functions of a search engine development tool. *Decision Support Systems*, 48(2):369 – 382, 2010.
[9] A. Diriye and G. Golovchinsky. Querium: a session-based collaborative search system. In *Proceedings of the 34th European conference on Advances in Information Retrieval*, ECIR'12, pages 583–584, Berlin, Heidelberg, 2012. Springer-Verlag.
[10] A. Ernst-Gerlach. Semiautomatisches Pflegen von Wrappern. Diplomarbeit, Universität Dortmund, FB Informatik, 2004.
[11] N. Fuhr, C.-P. Klas, A. Schaefer, and P. Mutschke. Daffodil: An integrated desktop for supporting high-level search activities in federated digital


ezDL - (visitor)

Search Tool

Search terms

java

Search

Categorys

(300)
Coverage of Java (61)
Classes Java (60)
Java Series (47)
Introduction to Java (45)
New Edition (45)
Java and Object-oriented Programmin
Java and JavaBeans (39)
Java Security (35)
Complete Reference (34)
Java Software (34)
Java Application (32)
Web Applets (32)
Java 1, 1 (31)
Java Database (29)
Learn to Use (28)
Essential Java (27)
Java by Example (26)
Java Professional (26)
Network (25)
Java Technology (25)
Developers how to Use (23)
Advanced Topics (23)
Experienced Programmers (23)
2nd Edition (22)
Java TM (22)
Java and XML (21)
Class Libraries (19)
Application Development (19)
Java Beans (19)
Complete Guide (19)
Quick Reference (17)
Advanced Techniques (16)
Java 2 and JavaScript (15)
Java in a Nutshell (14)
JavaScript Code (13)
World Wide Web (13)
Java Virtual Machine (12)
Java for Dummies (11)
Data Structures (10)
Java Unleashed (9)
Java and the as 400 (7)
JBuilder (6)
Java in 21 Days Sams Teach (6)
COBOL Programmers (6)
Visual Basic (6)
Java Bible (5)
RPG Programmer (3)
Real Solutions (3)
Projects you'll Never do at Work (2)
Pays Special Attention (2)
Ajax on Java (2)
Other Topics (19)

Results | Clustering

Freetext: java

(1129) Group by: Cluster

[Coverage of Java] (61)

Advanced Java 2 Development for Enterprise Applications (2nd Edition) (1999)
Cliff Berg
Prentice Hall PTR

Understanding Object-Oriented Programming With Java: Updated Edition (New Java 2 Coverage) (1999)
Timothy Budd
Addison Wesley

Professional Java, JDK 5 Edition (2005)
Jeff Scanlon
Wrox

Special Edition Using Java 1.1 (Special Edition Using) (1997)
Andrew Wooldridge
Que

Teach Yourself More Java in 21 Days (Sams Teach Yourself) (1997)
Michael Morrison
Sams Publishing

Java 2 Enterprise Edition Bible (2002)
Daniel H. Steinberg
Wiley

Java Programming Language Handbook: The Ultimate Source for Conquering the Java Programming Language (1996)
David H., Jr. Friedel
Coriolis Group Books

JavaBeans Programming from the Ground Up (1998)
Joseph O'Neil
Osborne/McGraw-Hill

Java for COBOL Programmers (2nd Edition) (Programming Series) (2002)

Search finished

Basket Tool

Double click on filed item to view details

Remove Selected Item | Remove All

Task Tool

Your current task

Task Training

You are looking for a new crime book from the author most known for the John Rebus series.

Cluster Word Cloud | Document Details

Double-click on result item to show its details.

Understanding Object-Oriented Programming With Java: Updated Edition (New Java 2 Coverage) (1999)

★★★★☆

Timothy Budd

This book develops the object-oriented programming skills of experienced programmers (at the Junior, Senior or above level) by teaching you how to use Java 2 (the latest release of Sun's Java platform), and providing a complete understanding of the philosophy behind Java 2. It starts out covering the language-independent concepts that are at the heart of the Java world, and then moves on to introducing Java through several example programs. Following this is an in-depth discussion on inheritance and polymorphism. Finally, the book concludes with material on important features of Java that are not object-oriented.

Features

Teaches students why the Java language works the way it does, as opposed to simply how it works

NEW! Chapter on Network Programming

NEW! Chapter on Java 2 - the most recent version of Java

NEW! Material on pure polymorphism

NEW! Appendices on Java Operators, Java Escape Characters, Java API, and Java Syntax

Publisher: Addison Wesley
ISBN: 0201612739

Details | Reviews (4) | Website

Add to basket

End Task

Figure 8: The *ezDL*-based system extended with clustering functionality developed within the CAIR project

libraries. In *Research and Advanced Technology for Digital Libraries. 6th European Conference, ECDL 2002*, pages 597–612, Heidelberg et al., 2002. Springer.

[12] B. J. Jansen and U. Pooch. Assisting the searcher: utilizing software agents for web search systems. *Internet Research: Electronic Networking Applications and Policy*, 14(1):19–33, 2004.

[13] C.-P. Klas, S. Kriewel, and A. Schaefer. Daffodil - Nutzerorientiertes Zugangssystem für heterogene digitale Bibliotheken. *dvs Band*, 2004.

[14] S. Kriewel. *Unterstützung beim Finden und Durchführen von Suchstrategien in Digitalen Bibliotheken*. PhD thesis, University of Duisburg-Essen, 2010.

[15] S. Kriewel and N. Fuhr. An evaluation of an adaptive search suggestion system. In *32nd European Conference on Information Retrieval Research (ECIR 2010)*, 2010.

[16] S. Kriewel, C.-P. Klas, A. Schaefer, and N. Fuhr. Daffodil - strategic support for user-oriented access to heterogeneous digital libraries. *D-Lib Magazine*, 10(6), June 2004. available at `http://www.dlib.org/dlib/june04/kriewel/06kriewel.html`.

[17] M. Lechtenfeld and N. Fuhr. Result clustering supports users with vague information needs. In *Proceedings of the 12th Dutch-Belgian Information Retrieval Workshop 2012, Ghent, Belgium*, February 2012.

[18] A. Paepcke. Digital libraries: Searching is not enough–what we learned on-site. *D-Lib Magazine*, 2(5), May 1996. `http://www.dlib.org/dlib/may96/stanford/05paepcke.html`.

[19] N. Pharo, T. Beckers, R. Nordlie, and N. Fuhr. Overview of the inex 2010 interactive track. In *Proceedings of the 9th International Workshop of the Initiative for the Evaluation of XML Retrieval (INEX 2010)*, 2011.

[20] A. Schaefer, M. Jordan, C.-P. Klas, and N. Fuhr. Active support for query formulation in virtual digital libraries: A case study with DAFFODIL. In A. Rauber, C. Christodoulakis, and A. M. Tjoa, editors, *Research and Advanced Technology for Digital Libraries. Proc. European Conference on Digital Libraries (ECDL 2005)*, Lecture Notes in Computer Science, Heidelberg et al., 2005. Springer.

[21] V. T. Tran. Entwicklung einer Unterstützung für Pearl Growing. Diplomarbeit, Universität Duisburg-Essen, 2011.

[22] V. T. Tran and N. Fuhr. Quantitative analysis of search sessions enhanced by gaze tracking with dynamic areas of interest. In *The International Conference on Theory and Practice of Digital Libraries 2012*. Springer tbp, September 2012.

[23] V. T. Tran and N. Fuhr. Using eye-tracking with dynamic areas of interest for analyzing interactive information retrieval. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM tbp, August 2012.

Apache Lucene 4

Andrzej Białecki, Robert Muir, Grant Ingersoll
Lucid Imagination
{andrzej.bialecki, robert.muir, grant.ingersoll}@lucidimagination.com

ABSTRACT
Apache Lucene is a modern, open source search library designed to provide both relevant results as well as high performance. Furthermore, Lucene has undergone significant change over the years, starting as a one-person project to one of the leading search solutions available. Lucene is used in a vast range of applications from mobile devices and desktops through Internet scale solutions. The evolution of Lucene has been quite dramatic at times, none more so than in the current release of Lucene 4.0. This paper presents both an overview of Lucene's features as well as details on its community development model, architecture and implementation, including coverage of its indexing and scoring capabilities.

Categories and Subject Descriptors
H.3.3 [**Information Search and Retrieval**]: Information Search and Retrieval

General Terms
Algorithms, Performance, Design, Experimentation

Keywords
Information Retrieval, Open Source, Apache Lucene.

1. INTRODUCTION
Apache Lucene is an open source Java-based search library providing Application Programming Interfaces for performing common search and search related tasks like indexing, querying, highlighting, language analysis and many others. Lucene is written and maintained by a group of contributors and committers of the Apache Software Foundation (ASF) [1] and is licensed under the Apache Software License v2 [2]. It is built by a loosely knit community of "volunteers" (as the ASF views them, most contributors are paid to work on Lucene by their respective employers) following a set of principles collectively known as the "Apache Way" [3].

Today, Lucene enjoys widespread adoption, powering search on many of today's most popular websites, applications and devices, such as Twitter, Netflix and Instagram [20, 4, 5] as well as many other search-based applications [6]. Lucene has also spawned several search-based services such as Apache Solr [7] that provide extensions, configuration and infrastructure around Lucene as well as native bindings for programming languages other than Java. As of this writing, Lucene 4.0 is on the verge of being officially released (it likely will be released by the time of publication) and represents a significant milestone in the development of Lucene due to a number of new features and efficiency improvements as compared to previous versions of Lucene. This paper's focus will primarily be on Lucene 4.0.

The main capabilities of Lucene are centered on the creation, maintenance and accessibility of the Lucene inverted index [31]. After reviewing Lucene's background in section 2 and related work in section 3, the remainder of this paper will focus on the features, architecture and open source development methodology used in building Lucene 4.0. In Section 4 we'll provide a broad overview of Lucene's features. In section 5, we'll examine Lucene's architecture and functionality in greater detail by looking at how Lucene implements its indexing and querying capabilities. Section 6 will detail Lucene's open source development model and how it directly contributes to the success of the project. Section 7 will provide a meta-analysis of Lucene's performance in various search evaluations such as TREC, while section 8 and 9 will round out the paper with a look at the future of Lucene and the conclusions that can be drawn from this paper, the project and the broader Lucene community.

2. BACKGROUND
Originally started in 1997 by Doug Cutting as a means to learning Java [8] and subsequently donated to The Apache Software Foundation (ASF) in 2001 [9], Lucene has had 32 official releases encompassing major, minor and patch releases [10, 11]. The most current of those releases, at the time of writing is Lucene 3.6.0.

From its earliest days, Lucene has implemented a modified vector space model that supports incremental modifications to the index [12, 19, 37]. For querying, Lucene has developed extensively from the first official ASF release of 1.2. However even from the 1.2 release, Lucene supported a variety of query types, including: fielded term with boosts, wildcards, fuzzy (using Levenshtein Distance [13]), proximity searches and boolean operators (AND, OR, NOT) [14]. Lucene 3.6.0 continues to support all of these queries and the many more that have been added throughout the lifespan of the project, including support for regular expressions, complex phrases, spatial distances and arbitrary scoring functions based on the values in a field (e.g. using a timestamp or a price as a scoring factor) [10]. For more information on these features and Lucene 3 in general, see [15].

Three years in the making, Lucene 4.0 builds on the work of a number of previous systems and ideas, not just Lucene itself.

Lucene incorporates a number of new models for calculating similarity, which will be described later. Others have also modified Lucene over the years as well: [16] modified Lucene to add BM25 and BM25F; [17] added "sweet spot similarity" and ILPS at the U. of Amsterdam has incorporated language modeling into Lucene [18]. Lucene also includes a number of new abstractions for logically separating out the index format and related data structures (Lucene calls them Codecs and they are similar in theory to Xapian's Backends [32]) from the storage layer - see the section Codec API for more details.

3. RELATED WORK

There are numerous open source search engines available today [30], with different feature sets, performance characteristics, and software licensing models. Xapian [32] is a portable IR library written in the C++ programming language that supports probabilistic retrieval models. The Lemur Project [33] is a toolkit for language modeling and information retrieval. The Terrier IR platform [34] is an open-source toolkit for research and experimentation that supports a large variety of IR models. Managing Gigabytes For Java (MG4J) [35] is a free full-text search engine designed for large document collections.

4. LUCENE 4 FEATURES

Lucene 4.0 consists of a number of features that can be broken down into four main categories: analysis of incoming content and queries, indexing and storage, searching, and ancillary modules (everything else). The first three items contribute to what is commonly referred to as the core of Lucene, while the last consists of code libraries that have proven to be useful in solving search-related problems (e.g. result highlighting.)

4.1 Language Analysis

The analysis capabilities in Lucene are responsible for taking in content in the form of documents to be indexed or queries to be searched and converting them into an appropriate internal representation that can then be used as needed. At indexing time, analysis creates tokens that are ultimately inserted into Lucene's inverted index, while at query time, tokens are created to help form appropriate query representations. The analysis process consists of three tasks which are chained together to operate on incoming content: 1) optional character filtering and normalization (e.g. removing diacritics), 2) tokenization, and 3) token filtering (e.g. stemming, lemmatization, stopword removal, n-gram creation). Analysis is described in greater detail in the section on Lucene's document model below.

4.2 Indexing and Storage

Lucene's indexing and storage layers consist of the following primary features, many of which will be discussed in greater detail in the Architecture and Implementation section:

- Indexing of user defined documents, where documents can consist of one or more fields containing the content to be processed and each field may or may not be analyzed using the analysis features described earlier.
- Storage of user defined documents.
- Lock-free indexing [20]
- Near Real Time indexing enabling documents to be searchable as soon as they are done indexing
- Segmented indexing with merging and pluggable merge policies [19]
- Abstractions to allow for different strategies for I/O, storage and postings list data structures [36]
- Transactional support for additions and rollbacks
- Support for a variety of term, document and corpus level statistics enabling a variety of scoring models [24].

4.3 Querying

On the search side, Lucene supports a variety of query options, along with the ability to filter, page and sort results as well as perform pseudo relevance feedback. For querying, Lucene provides over 50 different kinds of query representations, as well as several query parsers and a query parsing framework to assist developers in writing their own query parser [24]. More information on query capabilities will be provided later.

Additionally, Lucene 4.0 now supports a completely pluggable scoring model [24] system that can be overridden by developers. It also ships with several pre-defined models such as Lucene's traditional vector-space scoring model, Okapi BM25 [21], Language Modeling [25], Information Based [22] and Divergence from Randomness [23].

4.4 Ancillary Features

Lucene's ancillary modules contain a variety of capabilities commonly used in building search-based applications. These libraries consist of code that is not seen as critical to the indexing and searching process for all people, but nevertheless useful for many applications. They are packaged separately from the core Lucene library, but are released at the same time as the core and share the core's version number. There are currently 13 different modules and they include code for performing: result highlighting (snippet generation), faceting, spatial search, document grouping by key (e.g. group all documents with the same base URL together), document routing (via an optimized, in-memory, single document index), point-based spatial search and auto-suggest.

5. ARCHITECTURE AND IMPLEMENTATION

Lucene's architecture and implementation has evolved and improved significantly over its lifetime, with much of the work focused around usability and performance, with the work often falling into the areas of memory efficiencies and the removal of synchronizations. In this section, we'll detail some of the commonly used foundation classes of Lucene and then look at how indexing and searching are built on top of these. To get started, Figure 1 illustrates the high-level architecture of Lucene core.

5.1 Foundations

There are two main foundations of Lucene 4: text analysis and our use of finite state automata, both of which will be discussed in the subsections below.

5.1.1 Text Analysis

The text analysis chain produces a stream of tokens from the input data in a field (Figure 3). Tokens in the analysis chain are represented as a collection of "attributes". In addition to the expected main "term" attribute that contains the token value there

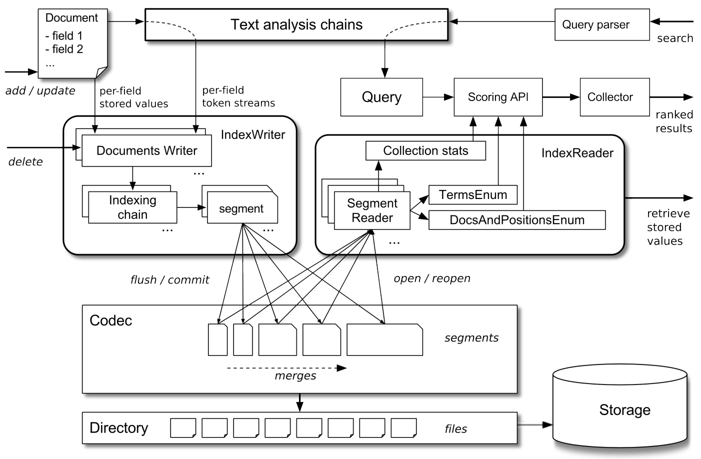


Figure 1 Lucene's Architecture

can be many other attributes associated with a token, such as token position, starting and ending offsets, token type, arbitrary payload data (a byte array to be stored in the index at the current position), integer flags, and other custom application-defined attributes (e.g. part-of-speech tags).

Analysis chains consist of character filters (useful for stripping diacritics, for instance), tokenizers (which are the sources of token streams) and series of token filters that modify the original token stream. Custom token attributes can be used for passing bits of per-token information between the elements of the chain.

Lucene includes a total of five character filtering implementations, 18 tokenization strategies and 97 token filtering implementations and covers 32 different languages [24]. These token streams performing specific functions such as tokenization by patterns, rules and dictionaries (e.g. whitespace, regex, Chinese / Japanese / Korean, ICU), specialized token filters for efficient indexing of numeric values and dates (to support trie-based numerical range searching), language-specific stemming and stop word removal, creation of character or word-level n-grams, tagging (UIMA), etc. Using these existing building blocks, or custom ones, it's possible to express very complex text analysis pipelines.

5.1.2 Finite State Automata

Lucene 4.0 requires significantly less main memory than previous releases. The in-memory portion of the inverted index is implemented with a new finite state transducer (FST) package. Lucene's FST package supports linear time construction of the minimal automaton [38], FST compression [39], reverse lookups, and weighted automata. Additionally, the API supports pluggable output algebras. Synonym processing, Japanese text analysis, spell correction, auto-suggest are now all based on Lucene's automata package, with additional improvements planned for future releases.

5.2 Indexing

Lucene uses the well-known inverted index representation, with additional functionality for keeping adjacent non-inverted data on a per-document basis. Both in-flight and persistent data uses variety of encoding schemas that affect the size of the index data and the cost of the data compression. Lucene uses pluggable mechanisms for data coding (see the section on Codec API below) and for the actual storage of index data (Directory API). Incremental updates are supported and stored in index extents

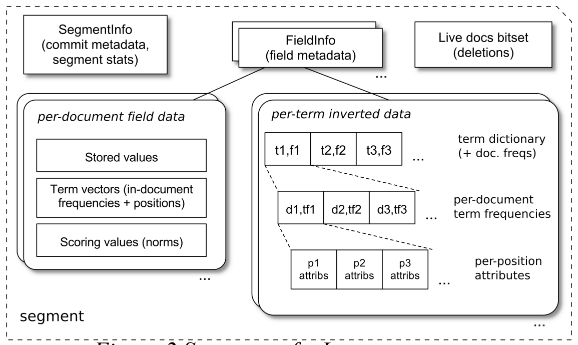

SegmentInfo (commit metadata, segment stats)
FieldInfo (field metadata)
Live docs bitset (deletions)
per-document field data
Stored values
Term vectors (in-document frequencies + positions)
Scoring values (norms)
per-term inverted data
t1,f1
t2,f2
t3,f3
term dictionary (+ doc. freqs)
d1,tf1
d2,tf2
d3,tf3
per-document term frequencies
p1 attribs
p2 attribs
p3 attribs
per-position attributes
segment

Figure 2 Structure of a Lucene segment.

(referred to as "segments") that are periodically merged into larger segments to minimize the total number of index parts [19].

5.2.1 Document Model

Documents are modeled in Lucene as a flat ordered list of fields with content. Fields have name, content data, float weight (used later for scoring), and other attributes, depending on their type, which together determine how the content is processed and represented in the index. There can be multiple fields with the same name in a document, in which case they will be processed sequentially. Documents are not required to have a unique identifier (though they often carry a field with this role for application-level unique key lookup) - in the process of indexing documents are assigned internal integer identifiers.

5.2.2 Field Types

There are two broad categories of fields in Lucene documents - those that carry content to be inverted (indexed fields) and those with content to be stored as-is (stored fields). Fields may belong to either or both categories (e.g. with content both to be stored and inverted). Both indexed and stored fields can be submitted for storing / indexing, but only stored fields can be retrieved - the inverted data can be accessed and traversed using a specialized API.

Indexed fields can be provided in plain text, in which case it will be first passed through text analysis pipeline, or in its final form of a sequence of tokens with attributes (so called "token stream"). Token streams are then inverted and added to in-memory segments, which are periodically flushed and merged. Depending on the field options, various token attributes (such as positions, starting / ending offsets and per-position payloads) are also stored with the inverted data. It's possible e.g. to omit positional information while still storing the in-document term frequencies, on a per-field basis [36].

A variant of an indexed field is a field where the creation and storage of term frequency vectors was requested. In this case the token stream is used also for building a small inverted index consisting of data from the current field only, and this inverted data is then stored on a per-document and per-field basis. Term frequency vectors are particularly useful when performing document highlighting, relevance feedback or when generating search result snippets (region of text that best matches the query terms).

Stored fields are typically used for storing auxiliary per-document data that is not searchable but would be cumbersome to obtain otherwise (e.g. it would require retrieval from a separate system). This data is stored as byte arrays, but can be manipulated through a more convenient API that presents it as UTF-8 strings, numbers, arrays etc., or optionally it can be stored using strongly typed API (so called "doc values") that can use a more optimized storage format. This kind of strongly typed storage is used for example to store per-document and per-field weights (so called "norms", as they typically correspond to field length normalization factor that affects scoring).

5.2.3 Indexing Chain

The resulting token stream is finally processed by the indexing chain and the supported attributes (term value, position, offsets and payload data) are added to the respective posting lists for each term (Figure 3). Term values don't have to be UTF-8 strings as in previous versions of Lucene - version 4.0 fully supports arbitrary byte array values as terms, and can use custom comparators to define the sorting order of such terms.

Also at this stage documents are assigned their internal document identifiers, which are small sequential integers (for efficient delta compression). These identifiers are ephemeral - they are used for identifying document data within a particular segment, so they naturally change after two or more segments are merged (during index compaction).

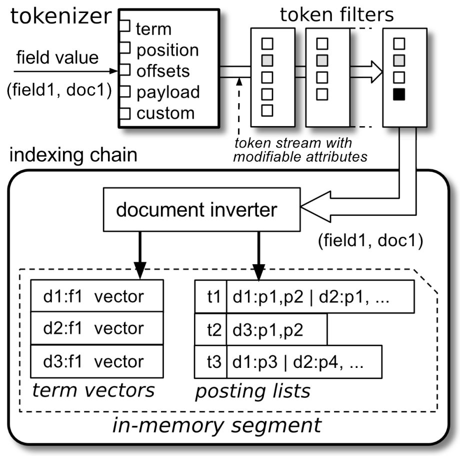


Figure 3 Indexing Process

5.3 Incremental Index Updates

Indexes can be updated incrementally on-line, simultaneously with searching, by adding new documents and/or deleting existing ones (sub-document updates are a work in progress). Index extents are a common way to implement incremental index updates that don't require modifying the existing parts of the index [19].

When new documents are submitted for indexing, their fields undergo the process described in the previous section, and the resulting inverted and non-inverted data is accumulated in new in-memory index extents called "segments" (Figure 2), using a compact in-memory representation (a variant of Codec - see below). Periodically these in-memory segments are flushed to a persistent storage (using the Codec and Directory abstractions), whenever they reach a configurable threshold - for example, the total number of documents, or the size in bytes of the segment.

5.3.1 The IndexWriter Class

The IndexWriter is a high-level class responsible for processing index updates (additions and deletions), recording them in new segments and creating new commit points, and occasionally triggering the index compaction (segment merging). It uses a pool of DocumentWriter-s that create new in-memory segments.

As new documents are being added and in-memory segments are being flushed to storage, periodically an index compaction (merging) is executed in the background that reduces the total number of segments that comprise the whole index.

Document deletions are expressed as queries that select (using boolean match) the documents to be deleted. Deletions are also accumulated, applied to the in-memory segments before flushing (while they are still mutable) and also recorded in a commit point so that they can be resolved when reading the already flushed immutable segments.

Each flush operation or index compaction creates a new commit point, recorded in a global index structure using a two-phase commit. The commit point is a list of segments and deletions comprising the whole index at the point in time when the commit operation was successfully completed. Segment data that is being flushed from in-memory segments is encoded using the configured Codec implementation (see the section below).

In Lucene 3.x and earlier some segment data was mutable (for example, the parts containing deletions or field normalization weights), which negatively affected the concurrency of writes and reads - to apply any modifications the index had to be locked and it was not possible to open the index for reading until the update operation completed and the lock was released.

In Lucene 4.0 the segments are fully immutable (write-once), and any changes are expressed either as new segments or new lists of deletions, both of which create new commit points, and the updated view of the latest version of the index becomes visible when a commit point is recorded using a two-phase commit. This enables lock-free reading operations concurrently with updates, and point-in-time travel by opening the index for reading using some existing past commit point.

5.3.2 The IndexReader Class

The IndexReader provides high-level methods to retrieve stored fields, term vectors and to traverse the inverted index data. Behind the scenes it uses the Codec API to retrieve and decode the index data (Figure 1).

The IndexReader represents the view of an index at a specific point in time. Typically a user obtains an IndexReader from either a commit point (where all data has been written to disk), or directly from IndexWriter (a "near-realtime" snapshot that includes both the flushed and the in-memory segments).

As mentioned in the previous section, segments are immutable so the deletions don't actually remove data from existing segments. Instead the delete operations are resolved when existing segments are open, so that the deletions are represented as a bitset of live (not deleted) documents. This bitset is then used when enumerating postings and stored fields and during search to hide deleted documents. Global index statistics are not recalculated, so they are slightly wrong (they include the term statistics of postings that belong to deleted documents). For performance reasons the data of deleted documents is actually removed only during segment merging, and then also the global statistics are recalculated.

The IndexReader API follows the composite pattern: an IndexReader representing a specific commit point is actually a list of sub-Readers for each segment. Composed IndexReaders at different points in time share underlying subreaders with each other when possible: this allows for efficient representation of multiple point-in-time views. An extreme example of this is the

Twitter search engine, where each search operation obtains a new IndexReader [20].

5.4 Codec API

While Lucene 3.x used a few predefined data coding algorithms (a combination of delta and variable-length byte coding), in Lucene 4.0 all parts of the code that dealt with coding and compression of data have been separated and grouped into a Codec API.

This major re-design of Lucene architecture has opened up the library for many improvements, customizations and for experimentation with recent advances in inverted index compression algorithms. The Codec API allows for complete customization of how index data is encoded and written out to the underlying storage: the inverted and non-inverted parts, how it's decoded for reading and how segment data is merged. The following section explains in more detail how inverted data is represented using this API.

5.4.1 A 4-D View of the Inverted Index

The Codec API presents inverted index data as a logical four-dimensional table that can be traversed using enumerators. The dimensions are: field, term, document, and position - that is, an imaginary cursor can be advanced along rows and columns of this table in each dimension, and it supports both "next item" and "seek to item" operations, as well as retrieving row and cell data at the current position. For example, given a cursor at field *f1* and term *t1* the cursor can be advanced along this posting list to the data for document *d1*, where the in-document frequency for this term (TF) can be retrieved, and then positional data can be iterated to retrieve consecutive positions, offsets and payload data at each position within this document.

This level of abstraction is sufficient to not only support many types of query evaluation strategies, but to also clearly separate how the underlying data structures should be organized and encoded and to encapsulate this concern in Codec implementations.

5.4.2 Lucene 4.0 Codecs

The default codec implementation (aptly named "Lucene40") uses a combination of well-known compression algorithms and strategies selected to provide a good tradeoff between index size (and related costs of I/O seeks) and coding costs. Byte-aligned coding is preferred for its decompression speed - for example, posting lists data uses variable-byte coding of delta values, with multi-level skip lists, using the natural ordering of document identifiers, and interleaving of document ID-s and position data [36]. For frequently occurring very short lists (according to the Zipf's law) the codec switches to using the "pulsing" strategy that inlines postings with the term dictionary [19]. The term dictionary is encoded using a "block tree" schema that uses shared prefix deltas per block of terms (fixed-size or variable-size) and skip lists. The non-inverted data is coded using various strategies, for example per-document strongly typed values are encoded using fixed-length bit-aligned compression (similar to Frame-of-Reference coding), while the regular stored field data uses no compression at all (applications may of course compress individual values before storing).

The Lucene40 codec offers, in practice, a good balance between high performance indexing and fast execution of queries. Since the Codec API offers a clear separation between the functionality of the inverted index and the details of its data formats, it's very easy in Lucene 4.0 to customize these formats if the default codec is not sufficient. The Lucene community is already working on several modern codecs, including PForDelta, Simple9/16/64 (both likely to be included in Lucene 4.0) and VSEncoding [26], and experimenting with other representations for the term dictionary (e.g. using Finite State Transducers).

The Codec API opens up many possibilities for runtime manipulation of postings during writing or reading (e.g. online pruning and sharding, adding Bloom filters for fail-fast lookups etc.), or to accommodate specific limitations of the underlying storage (e.g. Appending codec that can work with append-only filesystems such as Hadoop DFS).

5.4.3 Directory API

Finally, the physical I/O access is abstracted using the Directory API that offers a very simple file system-like view of persistent storage. The Lucene Directory is basically a flat list of "files". Files are write-once, and abstractions are provided for sequential and random access for writing and reading of files.

This abstraction is general enough and limited enough that implementations exist both using java.io.File, NIO buffers, in memory, distributed file systems (e.g. Amazon S3 or Hadoop HDFS), NoSQL key-value stores and even traditional SQL databases.

5.5 SEARCHING

Lucene's primary searching concerns can be broken down into a few key areas, which will be discussed in the following subsections: Lucene's query model, query evaluation, scoring and common search extensions. We'll begin by looking at how Lucene models queries.

5.5.1 Query Model and Types

Lucene does not enforce a particular query language: instead it uses Query objects to perform searches. Several Queries are provided as building blocks to express complex queries, and developers can construct their own programmatically or via a Query Parser.

Query types provided in Lucene 4.0 include: term queries that evaluate a single term in a specific field; boolean queries (supporting AND, OR and NOT) where clauses can be any other Query; proximity queries (strict phrase, sloppy phrase that allows for up to N intervening terms) [40, 41]; position-based queries (called "spans" in Lucene parlance) that allow to express more complex rules for proximity and relative positions of terms; wildcard, fuzzy and regular expression queries that use automata for evaluating matching terms; disjunction-max query that assigns scores based on the best match for a document across several fields; payload query that processes per-position payload data, etc. Lucene also supports the incorporation of field values into scoring. Named "function queries", these queries can be used to add useful scoring factors like time and distance into the scoring model.

This large collection of predefined queries allows developers to express complex criteria for matching and scoring of documents, in a well-structured tree of query clauses.

Typically a search is parsed by a Query Parser into a Query tree, but this is not mandatory: queries can also be generated and combined programmatically. Lucene ships with a number of different query parsers out of the box. Some are based on JavaCC grammars while others are XML based. Details on these query parsers and the framework is beyond the scope of this paper.

5.5.2 Query Evaluation

When a Query is executed, each inverted index segment is processed sequentially for efficiency: it is not necessary to operate on a merged view of the postings lists. For each index segment, the Query generates a Scorer: essentially an enumerator over the matching documents with an additional score() method.

Scorers typically score documents with a document-at-a-time (DAAT) strategy, although the commonly used BooleanScorer sometimes uses a TAAT (term-at-a-time)-like strategy when the number of terms is low [27].

Scorers that are "leaf" nodes in the Query tree typically compute the score by passing raw index statistics (such as term frequency) to the Similarity, which is a configurable policy for term ranking. Scorers higher-up in the tree usually operate on sub-scorers, e.g. a Disjunction scorer might compute the sum of its children's scores.

Finally, a Collector is responsible for actually consuming these Scorers and doing something with the results: for example populating a priority queue of the top-N documents [42]. Developers can also implement custom Collectors for advanced use cases such as early termination of queries, faceting, and grouping of similar results.

5.5.3 Similarity

The Similarity class implements a policy for scoring terms and query clauses, taking into account term and global index statistics as well as specifics of a query (e.g. distance between terms of a phrase, number of matching terms in a multi-term query, Levenshtein edit distance of fuzzy terms, etc). Lucene 4 now maintains several per-segment statistics (e.g. total term frequency, unique term count, total document frequency of all terms, etc) to support additional scoring models.

As a part of the indexing chain this class is responsible for calculating the field normalization factors (weights) that usually depend on the field length and arbitrary user-specified field boosts. However, the main role of this class is to specify the details of query scoring during query evaluation.

As mentioned earlier, Lucene 4 provides several Similarity implementations that offer well-known scoring models: TF/IDF with several different normalizations, BM25, Information-based, Divergence from Randomness, and Language Modeling.

5.5.4 Common Search Extensions

Keyword search is only a part of query execution for many modern search systems. Lucene provides extended query processing capabilities to support easier navigation of search results. The faceting module allows for browsing/drilldown capabilities, which is common in many e-commerce applications. Result grouping supports folding related documents (such as those appearing on the same website) into a single combined result. Additional search modules provide support for nested documents, query expansion, and geospatial search.

6. Open Source Engineering

Lucene's development is a collaboration of a broad set of contributors along with a core set of committers who have permission to actually change the source code hosted at the ASF. At the heart of this approach is a meritocratic model whereby permissions to the code and documentation are granted based on contributions (both code-based and non-code based) to the community over a sustained period of time and after being voted in by Lucene's Project Management Committee (PMC) in recognition of these contributions [3].

Development is undertaken as a loose federation of programmers coordinating development through the use of mailing lists, issue tracking software, IRC channels and the occasional face-to-face meeting. While all committers may veto someone else's changes, these rarely happen in practice due to coordination via the communication mechanisms mentioned. Project planning is very lightweight and is almost always coordinated by patches to the code that demonstrate the desired feature to some level more than abstract discussions about potential implementations. Releases are the coordinated effort of a community-selected (someone usually volunteers) release manager and a grouping of other people who validate release candidates and vote to release the necessary libraries. Lucene developers also strive to make sure that backwards compatibility (breakages, when known, are explicitly documented) is maintained between minor versions and that all major version upgrades are able to consume the index of the last minor version of the previous release, thereby reducing the cost of upgrades.

Lucene developers are often faced with the need to make tradeoffs between speed, index size and memory consumption, since Lucene is used in many demanding environments (Twitter, for example, processes, as of Fall 2011, 250 million tweets and billions of queries per day, all with an average query latency of 50 milliseconds or less [20].) For instance, the default Lucene40 codec uses relatively simple compression algorithms that trade index size for speed; field normalization factors use encoding that fits a floating point weight in a single byte, with a significant loss of precision but with great savings in storage space; large data structures (such as term dictionary and posting lists) are often accompanied by skip lists that are cached in memory, while the main data is retrieved in chunks and not buffered in the process' memory, relying instead on disk buffers of the operating system for efficient LRU caching.

Lucene 2, 3 and Lucene 4 have seen a significant effort to employ engineering best practices across the code base. At the center of these best practices is a test-driven development approach designed to insure correctness and performance. For instance, Lucene has an extensive suite of tests (for example, as of 7/1/2012, Lucene has 79% test coverage on 1 sample run at https://builds.apache.org/job/Lucene-trunk/clover/) and bench-marking capabilities that are designed to push Lucene to its limits. These tests are all driven by a test framework that supports the de facto industry standard notion of unit tests, but also the emerging focus on randomization of tests. The former approach is primarily used to test "normal" operation, while the latter, when run regularly (this happens many times throughout the day on Lucene's continuous integration system), is designed to catch edge cases beyond the scope of developers.

Since many things in Lucene are pluggable, randomly assembling these parts and then running the test suite uncovers many edge cases that are simply too cumbersome for developers to code up manually. For instance, a given test run may randomize the Codec used, the query types, the Locale, the character encoding of documents, the amount of memory given to certain subsystems and much, much more. The same test run again later (with a different random seed) would likely utilize a different combination of implementations. Finally, Lucene also has a suite of tests for doing large scale indexing and searching tasks. The results of these tests are tracked over time to provide better context for making decisions about incorporating new features or modifying existing implementations [24].

7. RETRIEVAL EVALUATION

At the time of this writing, the authors are not aware of any TREC-style evaluations of Lucene 4 (which is not unexpected, as it isn't officially released as of this writing), but Lucene has been used in the past by participants of TREC. Moreover, due to copyright restrictions on the data used in many TREC-style retrieval evaluations, it is difficult for a widespread open source community like Lucene's to effectively and openly evaluate itself using these approaches due to the fact that the community cannot reliably and openly obtain the content to reproduce the results. This is a somewhat subtle point in that it isn't that we as a community don't technically know how to run TREC-style evaluations (many have privately), but that we have decided not to take it on as a community due to the fact that there is no reliable way to distribute the content to anyone in the community who wishes to participate (e.g. who would sign and fill out the organizational agreement such as http://lemurproject.org/clueweb09/organization_agreement.clueweb09.worder.Jun28-12.pdf for the community?) and therefore it is not an open process on par with the community's open development process. For instance, assume contributor A has access to a paid TREC collection and makes an improvement to Lucene that improves precision in a statistically significant way and posts a patch. How does contributor B, who doesn't have access to the same content, reproduce the results and validate/refute the contribution? See [28] for a deeper discussion of the issues involved. Some in the community have tried to overcome this by starting the Open Relevance Project (http:lucene.apache.org/openrelevance) but this has yet to gain traction. Thus, it is up to individuals within the community who work at institutions with access to the content to perform evaluations and share the results with the community. Since most in the community are developers focused on implementation of search in applications, this does not happen publicly very often. The authors recognize this is a fairly large gap for Lucene in terms of IR research and is a gap these authors hope can be remedied by working more closely with the research community in the future.

In the past, some individuals have taken on TREC-style evaluations. In [17], a modified Lucene 2.3.0 was used in the 1 Million Queries Track. In [29], an unmodified Lucene 3.0, in combination with query expansion techniques, was used in the TREC 2011 Medical Track. In [30], Lucene 1.9.1 was compared against a wide variety of open source implementations using out of the box defaults. The impact of Lucene's boost and coordinate level match on tf / idf ranking is studied in [43]. Many researchers use Lucene as a baseline (e.g. [44]), a platform for experimentation or an example of implementation of standard IR algorithms. For example, [45] used Lucene 2.4.0 in an "out of the box" configuration, although it is not clear to these authors what an out of the box Lucene configuration is, since the community doesn't specify such a thing.

8. FUTURE WORK

While the nature of open source is such that one never knows exactly what will be worked on in the future ("patches welcome" is not just a slogan, but a way of development -- the community often jumps on promising ideas that save time or improve quality and these ideas often seemingly appear from nowhere.) In general, however, the community focus at the time of this writing is on: 1) finalizing the 4.0 APIs and open issues for release, 2) additional inverted index compression algorithms (e.g. PFOR) 3) field-level updates (or at least updates for certain kinds of fields like doc-values and metadata fields) and 4) continued growth of higher order search functionality like more complex joins, grouping, faceting, auto-suggest and spatial search capabilities. Naturally, there is always work to be done in cleaning up and refactoring existing code as it becomes better understood.

As important as the future of the code is to Lucene, so is the community that surrounds it. Building and maintaining community is and always will be a vital component of Lucene, just as keeping up with the latest algorithms and data structures is to the codebase itself.

9. CONCLUSIONS

In this paper, we presented both a historical view of Lucene as well as details on the components that make Lucene one of the key pieces of modern, search-based applications in industry today. These components extend well beyond the code and include an "Always Be Testing" development approach along with a large, open community collectively working to better Lucene under the umbrella that is known as The Apache Software Foundation.

At a deeper level, Lucene 4 marks yet another inflection point in the life of Lucene. By overhauling the underpinnings of Lucene to be more flexible and pluggable as well as greatly improving the efficiency and performance, Lucene is well suited for continued commercial success as well as better positioned for experimental research work.

10. ACKNOWLEDGMENTS

The authors wish to thank all of the users and contributors over the years to the Apache Lucene project, with a special thanks to Doug Cutting, the original author of Lucene. We also wish to extend thanks to all of the committers on the project, without which there would be no Apache Lucene: Andrzej Białecki, Bill Au, Michael Busch, Doron Cohen, Doug Cutting, James Dyer, Shai Erera, Erick Erickson, Otis Gospodnetić, Adrien Grand, Martijn van Groningen, Erik Hatcher, Mark Harwood, Chris Hostetter, Jan Høydahl, Grant Ingersoll, Mike McCandless, Ryan McKinley, Chris Male, Bernhard Messer, Mark Miller, Christian Moen, Robert Muir, Stanisław Osiński, Noble Paul, Steven Rowe, Uwe Schindler, Shalin Shekhar Mangar, Yonik Seeley, Koji Sekiguchi, Sami Siren, David Smiley, Tommaso Teofili, Andi Vajda, Dawid Weiss, Simon Willnauer, Stefan Matheis, Josh Bloch, Peter Carlson, Tal Dayan, Bertrand Delacretaz, Scott Ganyo, Brian Goetz, Christoph Goller, Eugene Gluzberg, Wolfgang Hoschek, Cory Hubert, Ted Husted Tim Jones, Mike Klaas, Dave Kor, Daniel Naber, Patrick O'Leary, Andrew C. Oliver, Dmitry Serebrennikov, Jon Stevens, Matt Tucker, Karl Wettin.

11. REFERENCES

[1] The Apache Software Foundation. The Apache Software Foundation. 2012. Accessed 6/23/2012. http://www.apache.org.

[2] Apache License, Version 2.0. The Apache Software Foundation. January 2004. Accessed 6/23/2012. http://www.apache.org/licenses/LICENSE-2.0

[3] How it Works. The Apache Software Foundation. Circa 2012. Accessed 6/24/2012. http://www.apache.org/foundation/how-it-works.html

[4] Interview with Walter Underwood of Netflix. Lucid Imagination. May, 2009. Accessed 6/23/2012. http://www.lucidimagination.com/devzone/videos-podcasts/podcasts/interview-walter-underwood-netflix

[5] Instagram Engineering Blog. Instagram. January 2012. Accessed 6/23/2012. http://instagram-

engineering.tumblr.com/post/13649370142/what-powers-instagram-hundreds-of-instances-dozens-of

[6] Lucene Powered By Wiki. The Apache Software Foundation. Various. Accessed 6/23/2012. http://wiki.apache.org/lucene-java/PoweredBy/

[7] Apache Solr. The Apache Software Foundation. Accessed 6/23/2012. http://lucene.apache.org/solr.

[8] Interview with Doug Cutting. Lucid Imagination. circa 2008. Accessed 6/23/2012. http://www.lucidimagination.com/devzone/videos-podcasts/podcasts/interview-doug-cutting

[9] Apache Subversion Initial Lucene Revision. The Apache Software Foundation. 9/18/2001. Accessed 6/23/2012. http://svn.apache.org/viewvc?view=revision&revision=149570

[10] Apache Subversion Lucene Source Code Repository. The Apache Software Foundation. various. Accessed 6/23/2012. http://svn.apache.org/repos/asf/lucene/java/tags/

[11] Apache Subversion Lucene Source Code Repository. The Apache Software Foundation. various. Accessed 6/23/2012. http://svn.apache.org/repos/asf/lucene/dev/tags/

[12] D. Cutting, J. Pedersen, and P. K. Halvorsen, "An object-oriented architecture for text retrieval," In Conference Proceedings of RIAO'91, Intelligent Text and Image Handling, 1991.

[13] Levenshtein VI (1966)."Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady 10: 707–10.

[14] Lucene 1.2 Source Code. The Apache Software Foundation. Circa 2001. Accessed 6/23/2012. http://svn.apache.org/repos/asf/lucene/java/tags/lucene_1_2_final/

[15] E. Hatcher, O. Gospodnetic, and M. McCandless. Lucene in Action. Manning, 2nd revised edition. edition, 8 2010.

[16] J. Pérez-Iglesias, J. R. Pérez-Agüera, V. Fresno, and Y. Z. Feinstein, "Integrating the Probabilistic Models BM25/BM25F into Lucene," arXiv.org, vol. cs.IR. 26-Nov.-2009.

[17] D. Cohen, E. Amitay, and D. Carmel, "Lucene and Juru at Trec 2007: 1-million queries track," Proc. of the 16th Text REtrieval Conference, 2007.

[18] A Language Modeling Extension for Lucene. Information and Language Processing Systems. Accessed 6/30/2012. http://ilps.science.uva.nl/resources/lm-lucene

[19] D. Cutting and J. Pedersen. 1989. Optimization for dynamic inverted index maintenance. In Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '90), Jean-Luc Vidick (Ed.). ACM, New York, NY, USA, 405-411.

[20] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-Time Search at Twitter."

[21] S. Robertson, S. Walker, and S. Jones, "Okapi at TREC-3," NIST SPECIAL 1995.

[22] Stephane Clinchant and Eric Gaussier. 2010. Information-based models for ad hoc IR. In Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '10). ACM, New York, NY, USA, 234-241

[23] G. Amati and C. J. Van Rijsbergen, "Probabilistic models of information retrieval based on measuring the divergence from randomness," ACM Transactions on Information Systems (TOIS), vol. 20, no. 4, pp. 357–389, 2002.

[24] Lucene Trunk Source Code. Revision 1353303. The Apache Software Foundation. 2012. Accessed 6/24/2012. http://svn.apache.org/repos/asf/lucene/dev/trunk/lucene/

[25] C. Zhai and J. Lafferty, "A study of smoothing methods for language models applied to ad hoc information retrieval," Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 334–342, 2001.

[26] Fabrizio Silvestri and Rossano Venturini. 2010. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In Proceedings of the 19th ACM international conference on Information and knowledge management (CIKM '10).

[27] Douglass R. Cutting and Jan O. Pedersen, Space Optimizations for Total Ranking, Proceedings of RAIO'97, Computer-Assisted Information Searching on Internet, Quebec, Canada, June 1997, pp. 401-412.

[28] TREC Collection, NIST and Lucene. Apache Lucene Public Mail Archives. Aug. 2007. Accessed 6/30/2012.

[29] B. King, L. Wang, I. Provalov, and J. Zhou, "Cengage Learning at TREC 2011 Medical Track," Proceedings of TREC, 2011.

[30] C. Middleton and R. Baeza-Yates, "A comparison of open source search engines," 2007.

[31] C. J. van Rijsbergen. Information Retrieval, 2nd edition. 1979, Butterworths

[32] The Xapian Project. Accessed 7/2/2012. http://www.xapian.org

[33] The Lemur Project. CIIR. Accessed 7/2/2012. http://www.lemurproject.org

[34] Terrier IR Platform. Univ. of Glasgow. Accessed 7/2/2012. http://www.terrier.org

[35] P. Boldi, "MG4J at TREC 2005," … Text REtrieval Conference (TREC 2005). 2005.

[36] V. Anh, A. Moffat. Structured index organizations for high-throughput text querying. String Processing and Information Retrieval, 304–315, 2006.

[37] G. Salton, E. A. Fox, and H. Wu. Extended Boolean information retrieval. Commun. ACM 26, 11 (November 1983), 1022-1036

[38] S. Mihov , D. Maurel. Direct Construction of Minimal Acyclic Subsequential Transducers. 2001.

[39] J. Daciuk, D. Weiss. Smaller Representation of Finite State Automata. In: Lecture Notes in Computer Science, Implementation and Application of Automata, Proceedings of the 16th International Conference on Implementation and Application of Automata, CIAA'2011, vol. 6807, 2011, pp. 118—192.

[40] Yves Rasolofo and Jacques Savoy. Term proximity scoring for keyword-based retrieval systems. In Proceedings of the 25th European conference on IR research (ECIR'03), Fabrizio Sebastiani (Ed.). Springer-Verlag, Berlin, Heidelberg, 207-218, 2003.

[41] S. Büttcher, C. Clarke, B. Lushman, B. Term proximity scoring for ad-hoc retrieval on very large text collections. Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, 621–622, 2006.

[42] A. Moffat, J. Zobel. Fast Ranking in Limited Space. In Proceedings of the Tenth International Conference on Data Engineering. IEEE Computer Society, Washington, DC, USA, 428-437, 1994.

[43] L. Dolamic, J. Savoy. Variations autour de tf idf et du moteur Lucene. In: Publié dans les 1 Actes 9e journées Analyse statistique des Données Textuelles JADT 2008, 1047-1058, 2008

[44] X. Xu, S. Pan, J. Wan. Compression of Inverted Index for Comprehensive Performance Evaluation in Lucene. 2010 Third International Joint Conference on Computational Science and Optimization (CSO), vol. 1, 382–386, 2010

[45] T. G. Armstrong, A. Moffat, W. Webber, and J. Zobel, "Has adhoc retrieval improved since 1994?," presented at the SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, 2009.

Galago: A Modular Distributed Processing and Retrieval System

[A Retrospective]

Marc-Allen Cartright, Samuel Huston, and Henry Feild

Center for Intelligent Information Retrieval
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
irmarc,sjh,hafeild@cs.umass.edu

ABSTRACT

The open source IR community must address the new needs of the current search engine landscape. While it is still possible for an individual to perform effective research or run a small to moderate-sized search engine on a single machine, the scope of search engine applications has moved far beyond these parameters. The exciting new frontiers of information retrieval lie now at the extremes: either the system and available resources are far more constrained than a desktop (as in mobile phones and tablets), or resources are expected to be available in quantities orders of magnitude larger (as in web-scale systems).

To inform the decisions in designing the next-generation of open source search engines (OSSEs), we present a retrospective assessment of the Galago search engine, an open source retrieval system developed at the University of Massachusetts Amherst. We have successfully deployed Galago over large clusters for both indexing and retrieval. At the other end of the spectrum, we have also successfully installed Galago on Android-based smart-phones and tablets, providing search capabilities over the personal data — tweets, social media posts, blog-feeds, emails, texts, browsing history, etc.— stored on one's cell phone.

These experiences have provided us with information that we feel is essential to communicate to all potential designers of open source search engines. In this paper, we discuss the aspects of Galago that we believe are worthy of carrying forward into the next generation of open source retrieval systems. Conversely, we also discuss the roadblocks encountered, both in terms of adoption by the larger research community and the difficulties in learning to use the system effectively. We hope that this retrospective will inform the architects of the next generation of open source retrieval systems.

Keywords

Galago, TupleFlow, retrieval system, search engine

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Information Storage and Retrieval-Systems and Software[Distributed Systems]; H.3.0 [**Information Systems**]: Information Storage and RetrievalGeneral

General Terms

Information Retrieval, Distributed Indexing

1. INTRODUCTION

The Galago[1] search engine is currently being developed at University of Massachusetts Amherst as a generational successor to Indri.[2] Indri emphasized two important factors: 1) the union of language models and inference networks and 2) processing speed. This model worked extremely well for its contemporary generation of research, and many groups used the software to produce a large body of published research. Galago has been designed with different goals in mind, to react to the next generation of research needs: 1) interoperation with a distributed processing environment, and 2) a modular, flexible processing model that allows drop-in components in virtually every step of the score calculation during retrieval. While we believe Galago has met these goals, to date Galago has not received the widespread adoption that Indri has. In this paper we take a look back at our own experiences with Galago in an attempt to learn as much as we can about the good, the bad, and the hopeful aspects of Galago.

We present our assessment as follows. When discussing positive aspects of Galago that we believe should be carried forward to the next generation of OSSEs, we present it as an *affordance*[3]. We then discuss issues we encountered when using Galago, as two-part assessments. We begin with a *problem* statement, which describes the specific issue we encountered with the system. We conclude the issue with the *lesson* that is the general rule or observation we hypothesize from our specific instance. We hope that this information will aid future system implementors by helping them to evolve the nascent affordances we found, and avoid the pitfalls we encountered.

[1]`http://www.lemurproject.org/galago.php`
[2]`http://www.lemurproject.org/indri/`
[3]Wikipedia states an affordance as a quality of an object, or environment, which allows an individual to perform an action. We use that definition here.

2. AFFORDANCES OF GALAGO

Despite the lack of widespread adoption, we believe Galago is a powerful retrieval system that emphasizes several elements that all future systems would do well to have. We focus on these elements here, and provide evidence in support of each claim.

2.1 Scoring Model Representation

Galago continues the use of a tree-based model from Indri, however several important changes make Galago's implementation much more powerful than Indri's. The Inference Network model described by Turtle and Croft [15] and implemented in Indri, provides a clean graphical way of describing a retrieval model. Additionally, it does so in a purely declarative way—the nodes in a query tree describe what they represent, but not how to materialize that information at retrieval time. Indri implemented this framework in a more formalized way by combining the Inference Network with Language Models. This proved to be a successful combination, as Indri is still in use as an active research system today, over 8 years after its initial development.

However, several issues limit the capabilities of Indri. The query language is difficult to update dynamically, therefore end users are limited to the constructs already defined in the language. Additionally, using the Inference Network requires adherence to a probabilistic interpretation of scoring documents. Many retrieval models do not produce values that can be considered probabilistic (the vector space model is an obvious example of this situation). Implementing these functions is not feasible without significant change to the code base and a thorough understanding of the scoring pipeline in Indri.

Galago solves these issues by generalizing away from a specific philosophy to a more general notion of a query tree. The only restriction in this model is that upon evaluation for a particular document, the tree reduces to a final value that is produced at the root node of the tree. Figure 1 shows the simple query `hubble telescope achievements` in the query tree representation. The `#combine` node at the top, when evaluated, produces a scalar value based on its parameters and the current document. We now discuss two powerful ideas that form the core of the query tree model: operators and traversals.

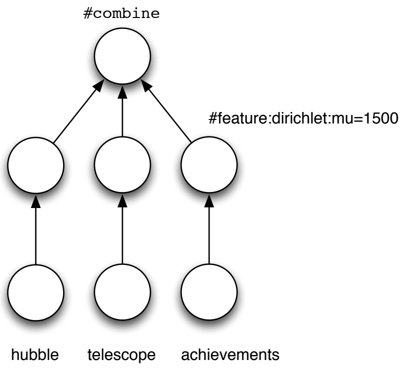


Figure 1: A simple query, represented as a tree. The middle layer of feature nodes in this query tree each convert frequency information about a term into a Dirichlet-smoothed probability.

Operators

An *operator* is a function over child nodes in the query tree that can produce a scalar upon evaluation of a document. In Figure 1, the only operator shown is `#combine`, which performs a linear combination of the child nodes of the operator. This may seem like a simple act, however, when generalized, the use of operators in this recursive manner means that given the proper operators, we can represent any arbitrarily complex function. In practice, we use operators to implement smoothing and scoring functions over raw terms, combine scores, and implement boolean match operations (and filtering and negated filtering operations). As we will see in the next section, operators can also work in conjunction with traversals to perform transformations across the entire tree to represent larger operations.

Traversals

A *traversal* is an operation over the query tree that transforms the tree in some way. Galago internally uses traversals extensively to annotate its query tree to prepare it for processing, check the correctness of submitted queries, optimize query execution [2], and rewrite the query tree, to name a few functions.

Operators and traversals are useful in isolation, but when you combine them together, you can implement highly expressive language constructs in a simple way. As a straightforward example, Figure 2 depicts the transformation of a small query tree under the `#sdm` operator. In this case the operator serves as a placeholder to indicate that the SDM-Traversal to expand the contained query using the Sequential Dependence Model described by Metzler and Croft [9]. The decomposed view of retrieval models afforded by query trees, in conjunction with operators and traversals, creates a powerful mechanism for implementing retrieval models very efficiently. We have additionally used combinations of operators and traversals to implement the Relevance Model [8], the field-based PRMS model [7], BM25 scoring [11] and it's field-based variant [10], to name a few of the implemented models. Each model was simple to implement and test in Galago, and is now part of the standard distribution of the system.

In this way, we can encapsulate a well-defined model in a shorthand form in the query language. A similar idea, known as *options*, has been a popular notion in the reinforcement learning community for over a decade [14]. An option is created to encapsulate a chain of actions that the system has deemed useful enough to treat as a primitive action. This allows increasing abstraction as the system progresses. In a similar fashion, as new operators and traversals are added to Galago, the query language can grow to include higher-level concepts as they are deemed useful enough to add.

2.2 Generalization of Distributed Processing

Galago comes packaged with its own distributed processing system, called TupleFlow. TupleFlow can be thought of as a MapReduce system, in that every process can consist of a map or reduce operation. The most well-known open-source implementation of MapReduce is Hadoop, maintained now by the Apache Software Foundation[4]. Hadoop has grown to be a field-tested implementation that has been scaled to clusters of several thousand machines to simultaneously support dozens of online users [5]. However, one place that we considered TupleFlow to far surpass Hadoop MapReduce was the option of multiple inputs and outputs for a processing stage. Hadoop has excellent support for single-stream input and outputs to processing stages, but adding even a single extra stream as input to the system can prove to be a test of patience. Consequently, implementing an ordered join of two or more streams, a popular operation in data processing, is an onerous task even for experienced Hadoop users.

[4]`http://hadoop.apache.org/mapreduce/`
[5]`http://research.yahoo.com/news/3374`

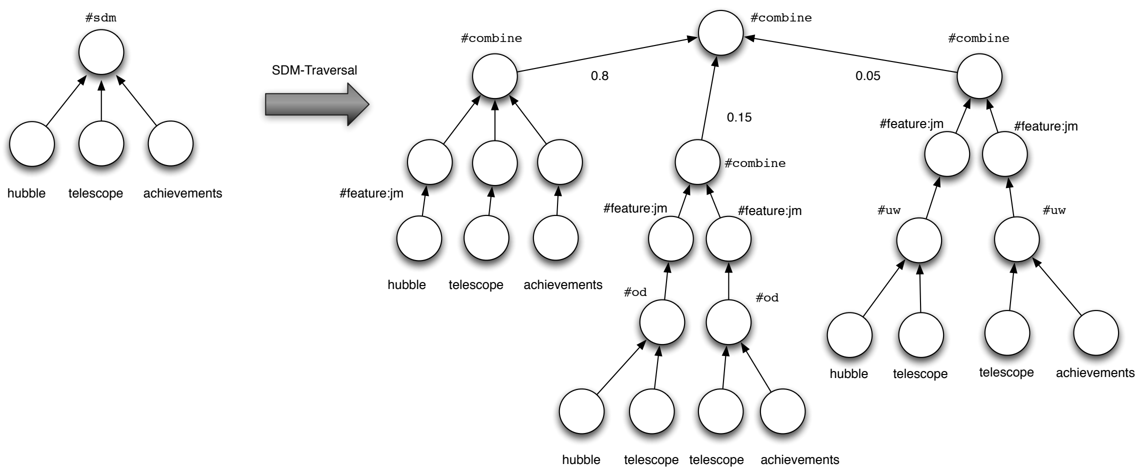


Figure 2: The expansion of the Sequential Dependence Model using a traversal.The layer of feature nodes in this query tree each convert frequency information about a term or a window into a Jelinek-Mercer-smoothed probability.

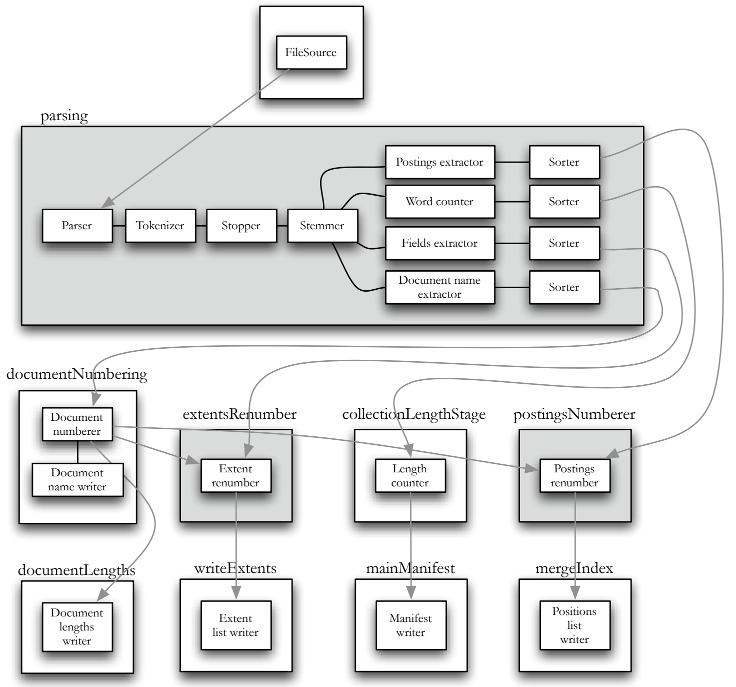


Figure 3: An example of indexing a collection using TupleFlow (generated by Trevor Strohman [12]).

Conversely, using multiple streams in TupleFlow requires indicating the extra connections in the configuration, and opening the stream in the processing stage, which requires only a single function call with the pipe name. Figure 3 shows the original indexing pipeline of Galago. The innermost boxes are *steps*, which are enclosed in *stages*. A single stage is run on a single machine. Shaded stages are *replicated*, meaning many instances of the same stage, with different input, are executed at the same time. A full explanation of the pipeline is beyond the scope of this paper. However, one can immediately see that several distinct stages can execute independently provided the prior input stage has completed. TupleFlow can analyze this dependence graph and execute these stages as soon as they are ready. A standard Hadoop implementation would require manual ordering of these stages, which would typically run serially without programmer intervention. After several years of experience with TupleFlow, we all agree that moving towards a general data processing model is beneficial for code reuse, higher-level reasoning, and processing. Trends in industry seem to agree — these ideas are being implemented as well in large-scale data processing systems, such as MR2[6], Spark[7] and Flume[8]. Although TupleFlow has not been widely adopted to date, we are fully aware of its capabilities, and it is clear to us that the idea of a *higher-order* distributed processing paradigm is essential to efficient research in the future of IR.

2.3 Pluggable Components

As mentioned previously, one of the original goals while building Galago was to allow users to easily extend the functionality. While not all components can be easily extended, many can, including parsing new corpus formats, query operators, query tree traversals, scoring regimes, and stream processing steps with TupleFlow. Using Java, developing pluggable components is easy since extensions can be packaged in completely separate archive (JAR) files. For example, to run Galago with a user defined query operator, the extension's JAR file is placed on the Java class path and the user tells Galago which class to associate with the operator at run time—and that's it. As all of the code is developed in Java, we have a guarantee that an external component developed elsewhere will run as intended on any system. In our own experience, external development has often provided an excellent development path for new components that we did not yet want to include in the main distribution. New components are developed and tested during research, when code is often not at its best. After the research is complete, we can assess the utility of the new components, and decide if we want to include them in the trunk of the source code. Often times integration into the main trunk provides a good opportunity to refactor the code into a more suitable form as well.

Finally, pluggable components allow users to contribute standalone extensions—not patches that need to be applied to the core code base—that can then be made publicly available and used with other extensions. Likewise, entire distributed processing programs can be made without the need to modify any of the core TupleFlow code, just as with Hadoop.

3. PROBLEMS ENCOUNTERED

In this section, we reflect on some of the issues we encountered while developing Galago and the lessons that we learned from the experience. Our hope is that these lessons apply not just to Galago, but OSSEs in general.

3.1 Steep Learning Curve

Problem: Learning to use Galago was often difficult and confusing. Several members of the CIIR have used both TupleFlow and Galago extensively in their research [1, 2, 3, 4, 5, 6, 13]. All users find the system useful and can effectively implement new components to add to the system in a short amount of time, often times within an afternoon. Unfortunately, the road to reach this point of expertise was long and complicated. The first two users spent almost a year learning the nuances of the system before they could effectively use it in research. Later adopters required less time as the early adopters were able to communicate the important aspects of the system effectively, saving new users several months of fumbling through a labyrinth of code. Had the system been better documented, we believe that would have led to the early adopters saving weeks, if not months, of time learning the details of Galago and TupleFlow.

[6]http://www.cloudera.com/blog/2012/02/mapreduce-2-0-in-hadoop-0-23/

[7]`http://www.spark-project.org/`

[8]`https://cwiki.apache.org/FLUME/`

Lesson: Thorough documentation of the system is crucial to success of future systems. Successful systems are often accompanied with copious amounts of documentation. A prime example of this model is the Hadoop MapReduce open-source implementation. Hadoop MapReduce is a complex system, however numerous individuals and organizations spent significant effort in documenting the system, both in providing code examples and reference texts explaining the important parts of the system. Without this documentation, it is unclear how many people would have had the spare time to learn to use such a sophisticated framework.

3.2 System Performance Analysis Problems

Problem: A VM complicates system performance analysis.

Indri is written in C++. The system fully compiles from source to machine code, making runtime execution very fast, and allowing for direct management of allocated memory. These are clear advantages when a researcher is concerned with system performance. However Galago was designed for modularity and extension. C++ is powerful, but it is also a difficult language to master, and may even have different behaviors on different machines depending on the architecture and compiler used.

To avoid these problems, Java is the language of choice for Galago. In many ways, it proved to be the right choice. At the time C++03 was in sore need of an update, and any modification of Indri proved to be torturous for any individual not intimate with most of the code base. Java removed the need for header files and moved the focus away from managing explicit pointers to implementing retrieval models and better design of processing algorithms.

However, several researchers at CIIR have shown interest in efficiency of retrieval systems, and Galago has proven to be a difficult system to deal with in this regard. Several procedures in Java, such as auto-boxing of primitives, and automatic garbage collection, have significant impact on wall-clock measurement and measurement of memory usage. In several instances, we have encountered large 'bumps' in timing data that we later realized was due to the virtual machine (VM)'s garbage collector performing a sweep. This kind of systemic incontinence is unacceptable from a systems measurement perspective.

In TupleFlow, the situation is not much better. Shuffling and sorting of large streams of data also suffer from overhead incurred in using the Java VM. For example the immutability of Strings, and then placement in the permGen memory pool required us to use our own string pooling mechanism to avoid exhausting memory too quickly. In a similar example, Hadoop MapReduce provides their own implementation of most of the boxed Java primitives in order to increase serialization and deserialization efficiency.

Lesson: Implementation language may inadvertently define a system's emphasis. The case of Galago shows two competing tensions in the research arena; efficiency and systems researchers prefer the low-level control afforded by a language such as C++, whereas researchers concerned with retrieval models (including learning-to-rank and users of external data sources) tend to prefer working in higher-level languages, where they can ignore issues such as memory-management or compression, and instead focus on the formulation of their respective scoring functions.

The choice of Java was relevant at the time due to limitations inherent in C++, and it seemed to provide a release from the purgatory of managing pointers and complicated inline functions in Indri. However this has also come at the price of control over several components of the system, and has made optimization of Galago more difficult. Ultimately, the choice of implementation language should be weighed against the main priority of the system. If you intend to support extensibility and portability, Java is still an obvious choice, as many projects have shown. However if your focus is on compression algorithms and indexing strategies, C++ provides a better platform for development.

Additionally, new languages, such as Scala[9] and Go[10], should be considered in future implementations. Although this may cause a "yet-another-language" issue, new languages are often developed to address the shortcomings of their predecessors. For example, Scala compiles to JVM bytecode, allowing it to use Java components. Additionally, the syntax of Scala is much less verbose than Java, and it even allows for rapid development of domain-specific languages. Future implementations of OSSEs may greatly benefit from the added capabilities of newer languages, however the choice of language, in many ways, defines the emphasis of the system being built.

3.3 Software Fragility

Problem: Backwards incompatibility. Right now systems such as Indri and Galago have several backward compatibility issues at the index and internal API levels. The standard update cycle for Indri and Galago currently suggests you rebuild any index you want to use with the new version, as the old indexes are simply considered defunct. When TREC collections or corporate collections numbered in the hundreds of thousands, or even into the low millions of documents, this was merely a tedious inconvenience. However, asking an end-user to rebuild a CLUE-sized index as a matter of process may well be unreasonable to many, and may even be impossible for those without the necessary resources. Additionally, changes to the internal API, which mostly affect plug-and-play systems like Galago, often impact any extensions users have created and renders them useless until they update to the new API.

Lesson: Design assuming that change is imminent. The internal mechanisms that interact with indexes should be capable of handling some amount of backwards compatibility. Lucene,[11] for example, guarantees that all index file formats are backwards compatible, preventing users from being forced to re-index collections. In an even larger scale example, protocol buffers, the data interchange format used most heavily at Google[12], was specifically designed for changes to occur to the definitions of the generated classes. As long as the changes are only additive, protocol buffers are guaranteed to be backwards compatible as well. Concerning the internal API, the best solution is to establish a standard that is sufficiently general such that the details behind the API can change without the need to adjust the API itself, while specific enough to allow users the necessary level of control within their extensions. While changes to the API cannot always be avoided, this will at least minimize the impact of small changes.

Problem: Difficult to extend. A major drawback of a system like Indri is the difficulty one encounters when attempting to add new functionality, such as a state of the art retrieval model. While Indri's C++ implementation allows for tight memory control and fast single-processor retrieval, adding additional functionality requires rooting around the internals, getting your hands dirty, and likely hitting many dead ends. What should take an hour can take days or weeks to the user unfamiliar with Indri's implementation. This is especially unpleasant given the necessity to explore new models in the fast paced world of information retrieval research. As we have already mentioned, one of the goals of Galago was to offer extensibility. However, in many of the earlier forms of Galago, extensibility was not always as prevalent as we hoped. Many functionalities were difficult to add in a clean and modular way, such as certain types of operators and index traversals such as passage or extent retrieval.

Lesson: Make modular extensibility a stronger focus. Retrospect also shows that some capabilities involve several axes, each of which should be designed for extension. Our canonical example is a user wanting to perform phrase-based retrieval over document passages. Passage-scoring requires a change in the semantics of what a "document" is, while phrases require knowing the positions of terms in documents. The interaction of these two concepts provides an interesting implementation challenge; one that would have influenced the design of the original system.

When a user wants to add functionality to a retrieval system, it should be possible to do so easily and without modifying the core system. That way the core can be updated independently of the extension. Part of the issue we encountered with Galago was not having the foresight to make certain components easily extendable. The key is to listen to what users want to extend but cannot. Rather than implement the desired functionality into the core, refactor the targeted component to be more modular and easily extended by users.

Problem: Different environments cause different problems. This problem plagued us in two different scenarios: at the distributed processing, "web-scale" level, and at the highly constrained, "mobile-device" level. We discuss each instance in turn, both of which lead us to a larger verdict.

Distributed indexing and retrieval. There is a large area of research that is emerging around distributed indexing and information retrieval. Information retrieval has long been focused on the problem of sorting and storing huge amounts of textual data, therefore parallel scalability is becoming one of the most important concerns in an IR system. A key problem in developing a distributed OSSE is that distributed processing environments each make different assumptions about the resources available to a distributed process. This means that each assumption that a system makes will reduce the number of clusters that can run the software.

High-level systems, such as Spark,[13] Pig, [14] and Hadoop, to name a few, provide high level interfaces for processing data. They require that the data is read and streamed through a series of functions provided by the distributed system and user defined functions are called for each data element. These systems generally take over the job generation, submission and control aspects of distributed programming. However, the assumptions made in these general processing systems may not be optimal for an IR system. A secondary concern is the measurement of parallel performance within systems like these cannot be tightly controlled.

Low-level systems, such as Grid Engine[15] and Mesos,[16] provide low level interfaces for running a set of programs on nodes within a cluster. In these systems, users must write code for job generation and control. The effect of node failure is a vital consideration when programming for these low-level systems. The storage of the data is also a major consideration for any distributed process. A centralized network attached storage can easily become a bottleneck for large clusters. A distributed file system is more scalable, but can lead to up to a network bottleneck, with up to $O(n^2)$ simultaneous communication channels between n running jobs.

[9]http://www.scala-lang.org/
[10]http://golang.org/
[11]http://lucene.apache.org/
[12]http://code.google.com/p/protobuf/

[13]http://www.spark-project.org/
[14]http://pig.apache.org/
[15]http://gridscheduler.sourceforge.net/
[16]http://incubator.apache.org/mesos/

In Galago we use the Tupleflow framework to generate jobs and provide submission control. A key problem of this system is that it assumes a centralized network attached storage system, which avoids the $O(n^2)$ blowout of a distributed file system, but can cause a bottleneck when performing many parallel disk operations. It is also important to note that Tupleflow's assumption of job control makes implementing an interface or job translation layer to high level distributed systems, such as Spark, Pig, or Hadoop, almost impossible. However, this same assumption allows TupleFlow to be easily extended to run on any cluster management software that allows direct submission of a series of binary or scripted jobs to be run in parallel.

Mobile phone deployment. When deploying Galago on an Android mobile phone platform, we encountered difficulty in even getting the system to operate correctly. Due to limitations in resources, mobile phones may only offer a subset of the standard API. In practice this meant that Galago did not have access to the full Java API when installed and executed on the Android JVM. Memory management and monitoring interfaces were not implemented in many early versions of the Android JVM. A crucial problem was that the Android environment replaces these unsupported API calls with *no-op* commands – this meant that compilation was possible, but execution would often produce errors from seemingly random, but dependent, sections of code.

Lesson: Be mindful of environmental assumptions. An OSSE must be careful about the assumptions it makes about the environment it will execute in. Tupleflow's assumption of a networked attached storage system directly limits several key parameters of the distributed processing space, such as 1) the number of parallel jobs, as the creation of too many jobs can overload the file server, and 2) the maximum number of concurrent open files, to name a couple. We believe that the best solution needs to appropriately abstract job control, data storage and transfer, and failure protection, to allow for maximum efficient scalability.

Conversely, when considering environments with limited resources, many of the decisions that aid the large-scale case are useless, or even detrimental, when resources are limited. Libraries and routines must be heavily optimized to squeeze every cycle and byte possible out of the scarce resources. While we offer no grand-unifying solution to this scale problem, we know OSSE designers must always be aware of the possible substrates their system may be planted in.

4. LOOKING FORWARD

Now that we have discussed the perceived advantages and disadvantages of using Galago, we turn towards "wishlist" items for the next-generation of OSSEs.

4.1 Unified Query Language

Each research retrieval system uses its own custom query language. For example, Indri supports a subset of INQUERY [17] queries in addition to several of its own, while Galago borrows from Indri, but differs in syntax and allows a more extensible formulation. Lucene and Terrier [18] each have their own query syntax (although their syntax is quite similar to each other). Table 1 shows some examples of the syntax used across these OSSEs. The difference in syntax means that a query formatted for Galago will not work with Indri, Lucene, or Terrier, causing issues if a user wants to move from one retrieval system to another. One way around the incompatibility of query languages is to settle on a standard, unified query syntax for the common operators across retrieval systems, e.g., for the operation of searching for a set of ordered terms. However, since each system has its own unique capabilities, it is also necessary to allow any unified query language to be extensible.

System	Proximity	Boolean not
Galago	`#uw10(a b)`	`#reject(#any(a) b)`
Indri	`#uw10(a b)`	`#not(a) b`
Lucene	`"a b"~10`	`-a b`
Terrier	`"a b"~10`	`-a b`

Table 1: An example of the query syntax for finding terms within a given proximity and using boolean negation under different retrieval systems.

While we do not presume to have a solution to this issue now, we believe the issue warrants discussion among the participants of the OSSE community. Many other communities have greatly benefited from standardization of the expression of their common concepts, surely the information retrieval community would stand to also gain by making a similar move.

4.2 External Data Services

A common theme in recent research is the use of external data sources in retrieval models. Sites like DBPedia,[19] Freebase,[20] and the Open Directory Project[21] provide free access to semi-structured data that provides information beyond a solitary indexed collection. In the upcoming wave of next-generation OSSEs, these data sources should be viewed as a persistent service, accessible by any researcher or client organization. There are obvious advantages to establishing common APIs to make use of these sites as services, including:

Less experimental variation. If all researchers had equal access to a set of static data services, then we can exclude potential sources of variance such as differences in data preparation that can often significantly impact results.

Less repeated work. Currently multiple organizations have to perform their own data acquisition and preparation for different data services. These processes are often labor intensive, and preclude any research involving these data sources. A single point of access and curation for these services could keep everyone from repeatedly "reinventing the wheel".

Reduced maintenance burden. Maintaining the API to a single data source is not itself difficult, but having to keep each of the systems up and running presents a large maintenance overhead for any organization. In the case of a smaller research group or a start up trying to break into a specific vertical of research, this overhead may be prohibitive. Spreading the maintenance work over several sites reduces the load on any single site, and certainly reduces wasted load due to unnecessary replication of maintenance.

4.3 Persistent Web-Scale Index

The ClueWeb project[22] is a considerable step towards bringing modern-day web-scale collections to information retrieval researchers. Unfortunately, not all information retrieval researchers can make use of the dataset, as compressed storage alone requires over 7 TB

[17]`http://www.ushmm.org/helpdocs/inquerylang.htm`
[18]`http://terrier.org/`

[19]`http://dbpedia.org/About`
[20]`http://www.freebase.com/`
[21]`http://www.dmoz.org/`
[22]`http://lemurproject.org/clueweb09.php/`

of space. On top of storage costs, it is simply not feasible to index a collection of that magnitude using a single machine. Even with enough resources available to process the collection, indexing the ClueWeb collection is not a trivial task, and future collections will only require more time and resources to manage.

As an alternative solution, we hope that the OSSE community would be willing to consider a crowdsourced-style solution, where instead of the same enormous monolithic collection being managed by each organization individually, instead each organization can be responsible for making the some portion of the collection available to other organizations as a callable API or service. This would provide the same benefits listed above, and each organization can instead focus on providing high reliability to a manageable set of documents, versus trying to simply complete the indexing process for themselves.

5. CONCLUSIONS

The open source IR community needs to reach some level of agreement in several key areas in order to move into the next phase of relevant research. In the past it was sufficient to perform experiments in an isolated environment, using either a single machine or a small cluster of machines specially purposed for the indexing task. However, if the next generation of open source search systems are to be relevant to clients and researchers alike, we must consolidate effort towards agreed standards. Towards this effort, we hope our experiences with Galago will provide valuable insight in the design of the next generation of open source search engines.

Galago provides three components that we believe should be standard elements of any next-generation open-source retrieval system: 1) A query tree representation of the query language, with operators and traversals that can be applied to the tree and composed in order to produce more complex higher-level functions; 2) integration with a distributed processing environment, preferably one that allows for high-level operations; and 3) extensibility to the core system. We believe the core of the system should serve as a skeleton for plugging in components that can be used during indexing and retrieval. It should be simple for an external user, with minimal knowledge of the internals, to extend the functionality of the core system.

Over the course of using and developing Galago, we also noted several issues with the system that, if possible, should be avoided in future OSSE implementations. While the effort to make Galago "everything to everyone" is admirable, it resulted in many difficulties that required redesigns of several components of the system, with still more improvements that could be made. We hope implementors of future systems can learn from our experiences, and design a software system that addresses each of these issues well before they are forced to deal with them.

Finally, we provide a "wish list" of ideas for the OSSE community. While these ideas are lofty, they would work towards the benefit of all involved parties, steering the focus away from the ever increasing, but necessary engineering and procedural overhead, and back towards developing cutting-edge search products and seminal research.

6. ACKNOWLEDGMENTS

This work was supported in part by the Center for Intelligent Information Retrieval, in part by NSF grant #IIS-0910884, and in part by NSF grant #CNS-0934322. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

7. REFERENCES

[1] M.-A. Cartright, E. Aktolga, and J. Dalton. Characterizing the subjectivity of topics. In *Proceedings of the 32nd international ACM SIGIR*, SIGIR '09, pages 642–643, 2009.

[2] M.-A. Cartright and J. Allan. Efficiency optimizations for interpolating subqueries. In *Proceedings of the 20th ACM CIKM*, CIKM '11, pages 297–306, 2011.

[3] M.-A. Cartright, H. Feild, and J. Allan. Evidence finding using a collection of books. In *Proceedings of the 4th ACM workshop on Online books, complementary social media and crowdsourcing*, BooksOnline '11, pages 11–18, 2011.

[4] J. Dalton, J. Allan, and D. A. Smith. Passage retrieval for incorporating global evidence in sequence labeling. In *Proceedings of the 20th ACM CIKM*, CIKM '11, pages 355–364, 2011.

[5] H. Feild, M.-A. Cartright, and J. Allan. The university of massachusetts amherst's participation in the inex 2011 prove it track. In S. Geva, J. Kamps, and R. Schenkel, editors, *Focused Retrieval of Content and Structure: 10th (INEX 2011)*, volume 7424 of *LNCS*. Springer, 2012.

[6] S. Huston, A. Moffat, and W. B. Croft. Efficient indexing of repeated n-grams. In *Proceedings of the fourth ACM WSDM*, pages 127–136, 2011.

[7] J. Kim and W. B. Croft. Retrieval experiments using pseudo-desktop collections. In *Proceedings of the 18th CIKM*, pages 1297–1306, 2009.

[8] V. Lavrenko and W. B. Croft. Relevance based language models. In *Proceedings of the 24th SIGIR*, pages 120–127, 2001.

[9] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *Proceedings of the 28th annual international ACM SIGIR*, SIGIR '05, pages 472–479, 2005.

[10] S. Robertson, H. Zaragoza, and M. Taylor. Simple BM25 extension to multiple weighted fields. In *Proceedings of the 13th CIKM*, pages 42–49, 2004.

[11] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th SIGIR*, pages 232–241, 1994.

[12] T. Strohman. *Efficient Processing of Complex Features for Information Retrieval*. PhD thesis, University of Massachusetts Amherst, 2007.

[13] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th annual international ACM SIGIR*, SIGIR '07, pages 175–182, 2007.

[14] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: a framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, Aug. 1999.

[15] H. Turtle and W. Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems (TOIS)*, 9(3):187–222, 1991.

A Framework for Bridging the Gap Between Open Source Search Tools

Madian Khabsa[1], Stephen Carman[2], Sagnik Ray Choudhury[2] and C. Lee Giles[1,2]
[1]Computer Science and Engineering
[2]Information Sciences and Technology
The Pennsylvania State University
University Park, PA
madian@psu.edu, shc5011@ist.psu.edu, sagnik@psu.edu, giles@ist.psu.edu

ABSTRACT

Building a search engine that can scale to billions of documents while satisfying the needs of the users presents serious challenges. Few successful stories have been reported so far [37]. Here, we report our experience in building YouSeer, a complete open source search engine tool that includes both an open source crawler and an open source indexer. Our approach takes other open source components that have been proven to scale and combines them to create a comprehensive search engine. YouSeer employs Heritrix as a web crawler, and Apache Lucene/Solr for indexing. We describe the design and architecture, as well as additional components that need to be implemented to build such a search engine. The results of experimenting with our framework in building vertical search engines are competitive when compared against complete open source search engines. Our approach is not specific to the components we use, but instead it can be used as generic method for integrating search engine components together.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval - search process; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Design, Documentation, Performance

Keywords

Search Engines, Software Architecture, Open Source

1. INTRODUCTION

In the past fifteen years, many search engines have emerged out of both industry and academia. However, very few have been successful [37]. Those are a number of challenges [28]. Firstly, the documents need to be collected before they are searchable. These documents may be found on the local intranet, a local machine, or on the Web. In addition, these documents range in format and type from textual files to multimedia files that incorporate video and audio. Expediently ranking the millions of results found for a query in a way that satisfies the end-user need is still an unresolved problem. Patterson [37] provides a detailed discussion of these hurdles.

As such, researchers and developers have spent much time and effort designing separate pieces of the search engine system. This lead to the introduction of many popular search engine tools including crawlers, ingestion systems, and indexers. Examples of such popular tools include: Apache Lucene/Solr, Indri, Terrier, Sphinx, Nutch, Lemur, Heritrix, and many others. The creators of each tool have introduced software with multiple features and advantages, but each one of them has its own limitations. Since the problems faced by designing a crawler are quite different from what an indexer developer would face, researchers have dedicated projects to tackling certain parts of the search engine. Furthermore, the same unit within the search engine (such as indexer) may introduce different challenges based on the application need. Working on scaling an indexer to support billions of documents is different than creating a real time indexer, therefore each use case has lead to a respective solution.

Few projects aim at building a complete open source search engine that includes a web crawler, ingestion module, indexer, and search interface. While complete search engine tools provide all the different pieces to run a search engine, these pieces tend to be outperformed by task specific open source search tools when compared against each other based on the specific task only. For example, while Apache Nutch provides an entire search engine solution, Heritrix, which is just a web crawler, is more powerful and versatile when compared solely against the Nutch crawler. This observation has lead us to consider building a unified framework where search engine components can be plugged in and out to form a complete search engine.

New projects are started everyday to solve a specific problem for search engines, or to to introduce new features. Likewise, many projects have been out of support and development after being abandoned by the community. The level of support and the richness of the features are what usually determines how prevalent an open source project is. We propose building a search engine framework that is modular and component agnostic where different crawlers or indices can be interchanged as long as they conform to a set of standards. This modularity facilitates plugging components that satisfies the users need with minimal to no changes of the framework's code base. Under such framework, powerful components can be included as they mature, and the community can focus on advancing specific parts of the search engine without worrying about building a complete search engine.

In this paper, we demonstrate how to exploit these freely available tools to build a comprehensive search framework for building vertical and enterprise search engines. We outline the architecture of the framework, and describe how each component contributes to building a search engine, and the standards and communication mechanisms between the different components. We rely on agreed upon standards to control the communication mechanisms between the different parts of the search engine in order to allow maximum flexibility and reduce coupling between the components. For the crawler, we assume that crawlers will save the crawled documents in WARC format, which became the standard for web archiving in 2009 [1]. Warc files are compressed files that contain records about the metadata of the crawled document along with the documents itself. We implement the middleware which is responsible for ingesting the crawled files, and pass them to the indexer over a REST API [26]. REST has become the defacto standard for accessing web services, and in this case we assume that indices are providing a REST API to communicate with the ingestion module and with the query interface. This assumption is rational as most indices end up providing some access mechanism over HTTP to support distribution.

The framework we are introducing here, *YouSeer*, not only can be used for building niche search engines but also for educational purposes. For the last three years YouSeer has been piloted in an advanced undergraduate/graduate class at Penn State designed to give students the ability to build high end search engines as well as properly reason around the various parts and metrics used to evaluate search engines. Students in this class were tasked with finding a customer within the Penn State community, usually a professor or graduate student, who is in need of a search engine. Students then were required to build a complete search engine from concept to delivery. This included crawling, indexing and query tuning if necessary. This class has yielded many niche based search engines of varying levels of complexity all using YouSeer as a software package. The level of technical know how in the class ranges from beginners with UNIX/Java to PhD level students in Computer Science/Engineering and Information Sciences and Technology.

The rest of this paper is organized as follows. Section 2 discusses related work and describes other open source search engines. Section 3 provides an overview of the architecture of YouSeer, while Section 4 discusses the workflow inside the framework. In Section 5 we describe the experiments. Finally, we conclude and identify areas of future work in Section 6.

2. RELATED WORK

The use of search engines to find content goes back to the days of library search, where librarians have used and developed the techniques of information retrieval to find content within books. These techniques have been carried out to the domain of web search, and enterprise search. Though web search added the concept of web crawler, or spider, to download the documents from the web, which can be used to discriminate the web search era from the previous era of librarian search.

The open source community felt the urge for an alternative to the commercial search engines that are dominating the market. Part of the need was to provide transparent solutions where users control the ranking of results and made sure they have not been manipulated. In addition, licensing the search services from these commercial search engines can be expensive.

ht://Dig [7] is one of the early open source search tools which was created back in 1995 at San Diego State University. The project is not designed to scale for the needs of entire web indexing. Instead it's designed to index content of few websites, or intranet. *ht://Dig* supported boolean and fuzzy queries, and had the ability to strip text out of HTML tags. Currently the project is out of support, as the latest release was back in 2004.

Apache Luecene [3] is considered one of the very popular search libraries which has been ported to multiple languages. It was originally written in Java by Doug Cutting back in 2000, and later it became part of the Apache Software Foundation. Though Lucene is not a search engine itself, but it's an indexing library which can be used to perform all the indexing operations on text files. It can be plugged into any search application to provide the indexing functionality.

Numerous search engines were developed on top of the Lucene library, including commercial and open source solutions. Nutch [29, 24] was among the early search engines developed on top of lucene. It added a web crawler and a search interface and used the lucene indexing library to build a comprehensive search engine. Nutch was later added to the Apache Software Foundation as a sub-project of Lucene. In developing Nutch, the developers have aimed at creating a scalable tool that can index the entire web. However, the largest crawl ever reported on Nutch was 100 million documents [29, 34] despite supporting parallel crawling on different machines. In addition, Nutch added link analysis algorithms to its ranking function to take into account the importance of the pages along with the relevancy.

Although Nutch provides rich set of crawling features, many other open source crawlers, i.e. Heritrix [5], provide far more complex and advanced features. For examples, the deciding rules for accepting a file or rejecting it in the crawling process are much powerful in Heritrix than Nutch. The ability to take check points, pause and resume crawling, and restore the crawling process in case of failure are also advantages of Heritrix that Nutch lacks. In addition, Nutch obeys the robots exclusion protocol, *Robots.txt*, and force the users to obey it without giving them the option ignore it totally or partially. Along with forcing minimum waiting time between fetching attempts for files on the same domain, the crawling process using Nutch may end up being slow.

Another popular distribution of Lucene is Apache Solr [4] which provides enterprise search solution on top of Lucene. Solr provides a RESTful like API on top of Lucene, so that all communication with the index is done over HTTP requests which makes Solr embed-able into any application without worrying about the implementation. On top of that, Solr provides distributed searching, query caching, spell corrections, and faceted search capabilities. But as mentioned, Solr is another search framework and not a complete search engine solution. The main missing component is a web crawler. Though it can be plugged into Nutch as the background indexer instead of Lucene. A query interface and results page are also missing, as the shipped interface is for testing purposes only. This is attributed to the fact that Solr is not a standalone application, rather it's a library or framework which get plugged into another application.

NutchWAX [13] attempts to merge Nutch with the web archive extensions such that the solution will search the web archive. Currently it only supports ARC files, though WARC [1] files (standard format for archiving web content) are easily converted to ARC. NutchWAX requires Hadoop platform [2] to run the jobs on it, as the tasks are implemented using the Map/Reduce paradigm [25].

Since Lucene proved to be a scalable indexing framework, many open source search engine adopted it and built solutions on top of it. For example, Hounder [6] not only capitalize on Lucene, but also on some modules of Nutch to provide large scale distributed indexing and crawling services. The crawling and the indexing processes are easily configured, launched and monitored via GUI or shell script. However, the options that can passed to the crawler are limited compared to large scale crawlers like Heritrix, as most of the configurations are regular expressions only. These regular expressions are either entered into a simple java GUI, or appended to the numerous configuration files. The extendibility of the system is not an easy task as well.

Another popular search framework is Xapian [21], which is built with C++ and distributed under GPL license. The framework is just an indexing library, but it ships with web site search application called Omega [14] that can index files of multiple formats. Though the Xapian framework doesn't contain a crawler, Omega can crawl files on the local files system only.

Researchers at Carnegie Mellon University and University of Massachusetts Amherst have developed Indri [40, 8] as a part of the Lemur project [33, 10]. Indri is a language model based search engine that can scale for 50 million documents on a single machine and 500 million documents on a cluster of machines. It supports indexing documents from different languages, and multiple formats including PDF, Word, PPT. In addition to traditional IR models, Indri combines inference networks along with language models which makes it a unique solution when compared to other frameworks. However, Indri doesn't contain a web crawler, and has to be used along with a 3rd party crawler. Nevertheless, Indri can ingest documents from the file system, TREC collections, or archived files in a WARC format. So, the choices for a crawler to use with Indri are large.

Besides indexing web content and intranet documents, some search engines were developed to index SQL databases. Providing full text search for DBMS content is important for enterprises with large databases, especially when the built in full text search is not fast enough. Sphinx [15] provides a full text search solution for SQL databases, as it comes with connectors to many commercial databases. Besides connecting to databases, Sphinx may be configured to index content from XML files which were written in specific format. But, since Sphinx is aimed at indexing SQL databases, it can't be considered as a complete search engine, and rather a SQL indexing framework.

At RMIT university, researchers have developed Zettair [22], an open source search engine which is written in C. Zettair's main feature is the ability to index large amounts of text files [31]. It's been used to index 426GB of TREC terabyte track collection, according to the official documentation page. On the flip side, Zettair can only deal with HTML and plain text files, thus lacking the feature of indexing rich media files. Besides, it assumes the user has already crawled the files, and doesn't provide any crawler out of the box.

Swish-e [16] is another open source search engine which is suitable for indexing small content, less than 1 million documents. It can be used to crawl the web via a provided perl script, and index files from various data-types: PDF, PPT ...etc. But since it can only support up to one million documents, scalability is not a feature of this search engine.

As academia continued to contribute to open source search engines, researchers at the University of Glasgow have introduced Terrier [17, 36, 35] as the "first serious answer in Europe to the dominance of the United States on research and technological solutions in IR" [36]. Terrier provides a flexible and scalable indexing system with implementations of many state-of-the-art information retrieval algorithms and models. Terrier has proven to compete with many other academic open source search engines, like Indri and Zettair, in TREC workshops [36]. To support large scale indexing, Terrier uses MapReduce on Hadoop clusters to parallelize the process. Interacting with Terrier is made easy for almost all users by providing both desktop and web interface. Nevertheless, as the case with many other open source search solutions, Terrier doesn't ship with a web crawler, but it can be integrated with a crawler that was also developed at the University of Glasgow: labrador [9].

MG4J *(Managing Gigabytes for Java)* [23, 11] is a search engine released under GNU lesser general public license. It's being developed at the University of Milan, where researchers are plugging state-of-the-art algorithms and ranking functions into it. The package lacks a fully-fledges web crawler, and relies on the user to provide the files, but it can crawl files on the file system. Despite the numerous advantages of the system, ease of use seems to elude this search engine.

WebGlimpse [20] is yet another search engine, though it has a different licensing model as it is free for students and open source projects, but needs to be licensed for any other use. It's built on top of Glimpse indexer which generate indices of very small size compared to the original text (2-4% of the original text) [30]. WebGlimpse can index files on the local filesystem, remote websites, or even crawl webpages from the web and index them. But the crawler has limited options which makes it incompetent to do a large scale web crawl.

mnoGoSearch [12] is another open source search engine which have versions for different platforms including Linux/Unix and Windows. It's a databases back ended search engine, which implements its own crawler and indexer. Since it's dependent on the databases in the back end, the database connectivity may become the bottleneck of the process in case the number of running threads passed the limit of concurrent open connections to the database. The configuration options of the crawler are also limited compared to Heritrix and Nutch. Besides, indexing rich media files like PDF and Doc is not supported internally, though external plugins can be used to convert these files.

When comparing open source search engines, many aspects are taken into consideration. These aspects include: completeness of the solution in terms of components (example: some libraries don't have crawlers), the scalability of the solution, the extendibility of the search engine, the supported file types, license restrictions, support of stemming, stop words removal, fuzzy search, index language, character encoding, providing snippets for the results, ranking functions, index size compared to the corpus size, and query response time.

Many researchers had done work on comparing the performance of multiple open source search engines. Middleton and Baeza-Yates compared 29 popular open source search engines in [31]. Their comparison considered many of the aspects mentioned before, along with precision and recall performance results on TREC [18] dataset. However, their analysis is more focused on the indexing section of the search engine without considering the crawling process at all. In fact, many of the libraries that they compare are only indexing libraries, and not complete search engines (i.e. Lucene). Another experiment on indexing with open source search libraries was performed in 2009 on data from Twitter [39]. This experiment was conducted on small data which doesn't test the scalability of the indexer. Similar to the study by Middleton and Baeza-Yates [31], [39] doesn ot take into consideration the crawling task of the search engine.

3. ARCHITECTURE AND IMPLEMENTATION

Our design must include the most important parts of a search engine, a crawler and the index engine. YouSeer's architecture is presented in Figure 1. While most of YouSeer's components can be substituted with equivalent open source components, we describe the architecture and the implementation using the components we deploy, without loss of generality of the approach.

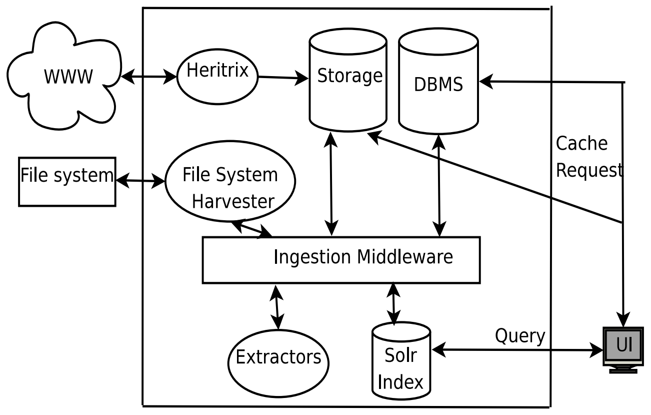


Figure 1: YouSeer Architecture.

The framework is implemented in Java and the interfaces are in JSP.

3.1 Crawler

A web crawler is a software that downloads documents from the web and stores them locally. The process of downloading is sequential where the crawler will extract the outgoing links from every downloaded document and schedule these links to be fetched later according to the crawling policy.

The Internet Archive's crawler, named Heritrix [32], was chosen as a web crawler for YouSeer. Heritrix serves as good example of embedding any web crawler into a search engine since it dumps the downloaded documents to the hard disk in the Warc format, which in 2009 became the standard for archiving web content[1]. By default, Heritrix writes the downloaded documents into compressed ARC files, where each file aggregates thousands of files. Compressing and aggregating the files is essential to keeping the number of files in the file system manageable, and sustaining lower access time.

Heritrix expects the seed list of the crawl job to be entered as a text file along with another file that defines the crawling policy. Then the crawler will proceed by fetching the URLs in the seed list and write them to ARC/Warc files. This process can be assumed to be the standard workflow of any web crawler, thus the integration of Heritrix can be used as example on how to integrate almost any web crawler into a search engine.

Heritrix provides flexible and powerful crawling options that make it ideal for multiple focused crawling jobs. These features include the ability to filter documents based on many deciding rules such as regular expressions, file types, file size, and override the policies per domain. The ability to tune the parameters of connection delay, and control the max wait time along with number of concurrent connections are advantageous when crawling for a vertical search engine. Despite the lack of support for parallel crawling on multiple instances, Heritrix is continuously being used at the Internet Archive to crawl and archive the web which can be argued to be the largest crawl ever to be conducted using an open source crawler. While teaching a search engine class, students have preferred using heritrix over other open source crawlers such as Nutch because Heritrix provides an easy to use web interface to run crawling jobs, and for the richness of the features and the detailed control of the parameters which the students can specify. This helped the students grasp the challenges of crawling the web, while at the same time gave them the chance to monitor how the crawl job is progressing and what parameters they need to change.

Besides the web crawler, YouSeer implements its own local hard drive harvester. This allows it to function as a desktop search engine as well. The crawler runs in a breadth-first manner, starting at a certain directory or network location iterating over the the files and folders inside that folder. This would complement our assumption in the framework that crawlers should produce Warc files, as the local file harvester would enable YouSeer to index documents mirrored onto the file system by different crawlers that do not produce Warc files.

3.2 Indexer

YouSeer adopts Apache Solr for indexing, which provides a REST-like API to access the underlying Lucene index. Dealing with an indexing interface with a RESTful API [26] over HTTP gives a layer of abstraction to the underlying indexing engine, and provides YouSeer with the ability to employ any indexing engine as long as it provides a REST API [26]. Such an API may be built as a wrapper on top of the existing non-web API. Thus, the indexing engine in YouSeer is just a URL with the operations that the index provides. These operations are typically: index, search, delete, and optimize.

Besides the native features of Lucene, Solr provides additional features like faceted search, distributed search, and index replication. All these features combined with the flexibility to modify the ranking function makes a good case for adopting Solr as indexer. In addition, Solr is reported to be able to index 3 billion documents [38].

YouSeer distribution deploys two instances of Solr, one for web documents and another one for files crawled from the desktop. The separation between the instances can be achieved either by having two standalone Solr instances, or two cores deployed on the same instance. Cores are methods for running multiple indices with different configuration in the same Solr instance. By default one core is configured to index content from the web, and the other core is used to index documents on the file system. In the case of multi-core solr, users maintain a single Solr instance, while having the ability to tune each index independently. This becomes a need as field numbers for web content differs from file-system content. More importantly, the ranking of web documents may be far more complicated than ranking file-system content. Since YouSeer is only aware of the URL of the core (a core is treated just like a dedicated index), it can be easily modified to use a dedicated index instead of a core in case the number of documents scales beyond what a single core can handle. Furthermore, Solr distributed search techniques can be used to replace a core when the number of documents grows beyond the capabilities of a single machine. This seamless transition is made possible because all the different distributions of the index (cores, standalone, distributed) provide the same RESTful API, and the ingestion module along with the query interface only care about the server URL without knowing the specific implementation of the server.

3.3 Database

A search engine occasionally needs to store some information in a database. Such information can be for reporting purposes, or needed for performing certain operations. YouSeer uses a database for three reasons: (1) keep track of successfully processed Warc/Arc files in order to avoid processing them again, (2) for storing metadata about cached documents, and (3) to log errors during ingestion. YouSeer uses MySQL as DBMS server, however SQLite was proved to be suitable for small to medium level datasets. YouSeer interacts with the database through JDBC, hence it can adopt any DBMS that has a JDBC driver.

3.4 Extractor

Search engines need to handle files in multiple formats ranging from simple html pages to files with rich content like Word and PDF along with audio and video. Apache TIKA [19] empowers YouSeer with the ability to extract metadata and textual content from various file formats such as PDF, WORD, Power Point, Excel Sheets, MP3, ZIP, and multiple image formats. Tika is currently a standalone Apache project that supports standard interface for converting and extracting metadata from popular document formats. While YouSeer ships coupled with Tika, it's still fairly straightforward to replace it with other converters as needed.

3.5 Ingestion Module

The ingestion module is where the crawled documents get processed and are held to be indexed. The Warc/Arc files are processed to extract the records containing individual documents and the corresponding metadata. Documents of predefined media types are passed to multiple extraction modules such as PDFBox to extract their textual content and metadata. The user specifies the mime types she is interested in indexing by editing a configuration file that lists the accepted mime types. The extracted information is later processed and submitted to the index. This module also stores the document's metadata into the database and keeps track of where the cached copy is stored.

The ingestion module is designed in a such a way that different extractors operate on the document, after that each extractor emits extracted fields, if any, to be sent to the index. By default the *Extractor* class provides all the out of the box extraction and population for the standard fields of the index such as title, url, crawl date and others, while *CustomExtractor* is left for the end user to implement. *CustomExtractor* is called after *Extractor* giving the user the ability to override the extracted fields, and extract new fields. This approach makes it easy for the users to implement their own information extractors. For example, while building a search engine for a newspaper website, the customer asked for providing search capability based on the publication date. The publication date could be extracted from the URL as the newspaper formats its URL as follows:

www.example.com/YYYY/MM/DD/article.html

To achieve this, we implemented the *CustomExtractor* class of the ingestion module so that it would extract the information from the URL and append the extracted date to the xml file which is to be sent to the indexer.

3.6 Query Interface

YouSeer has two search interfaces basic and advanced that provide access to the underlying Lucene index. The basic interface is similar to most search engines where users enter a simple query term before the relevant links are returned. The advanced search provides search fields for each searchable field in the index and allows the users to set the ranking criterion. Query suggestions, aka auto-complete, are displayed for the user while inputting the query terms. These suggestions are generated offline by extracting the word-level unigram, bigram, and trigram of terms in the index. When enough query logs are accumulated, they can be used for query suggestions instead of the index terms.

Furthermore, queries are checked for misspellings using the terms in the index instead of a third party dictionary. This would be suitable in the case of vertical search engines that deal with special domains terminologies.

Since the index is accessed through a REST web service, the query interface receives the query terms from the user and send an HTTP request to the index. The REST API provides a level of abstraction for the interface to communicate with multiple types of indices as long as they provide a similar API.

Along with the query interface, YouSeer provides an admin interface from which users can launch new ingesting jobs and track the progress of previously started jobs.

3.7 Documents Caching

Accessing older versions of some documents, or being able to view them while their original host is down, is considered an advantage for a search engine. The caching module keeps track of the different versions that have been crawled and indexed of a document. As documents are stored into Ward/Arc files, the relative location of the containing Warc file is stored in the index along with the documents fields. In addition to the surrogate file name, the index would contain the offset of the file within the Warc file. The offset is needed because Warc and Arc files can only be read sequentially. The Warc/Arc files are mounted on virtual directory on the web server, therefore they can be accessed over the network allowing them to be located on a different location than the server or the crawler.

When the user requests a cached version for an indexed document, the caching module locates the containing Arc/Warc file and seeks to the beginning of the document's record reading it and returning content to the user. If the requested file is not an HTML document, the module can convert DOC, PDF, PPT, XLS format and other formats into HTML.

YouSeer caching module provides integration with Google Docs preview, so that cached documents can be viewed as a Google document on the fly. The feature works on supported formats only, like PDF, Doc, and PPT. This allows users to quickly view rich media documents without the need to download them. Figure 2 shows the workflow of the caching module.

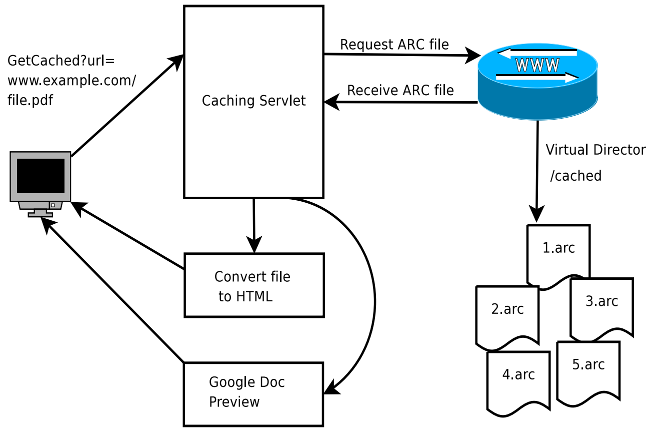


Figure 2: Cache Architecture

4. WORKFLOW

In this section we present an overview of how the whole system works. A typical job starts by scheduling a crawl task on Heritrix. First the seed URLs are provided and the rest of the parameters are defined. These parameters include the max-hop, max file size limit, max downloaded files limit, and other crawl politeness and request-delay values. The crawler proceeds by fetching the seed URLs, extracting their outgoing links, and scheduling these links for an in breadth-first crawl. As part of the parameter specification, the user chooses the format by which the crawled results are written into. The default is Arc, but other file formats such as Warc or simply mirroring the fetched documents on a hard drive are available. Converting the Arc files into Warc format can be accomplished through command line tool. Should the user keep the format as Arc, the downloaded documents are combined and then written to a single compressed ARC file [32], which is in this case limited to 100MB. Along with every document, Heritrix stores a metadata record in the compressed file.

The ingestion module, which is the middleware between the crawler and the index, waits for the ARC/WARC files to be written and then iterates on all the documents within the ARC file processing them sequentially. The ingestion process does not necessarily wait for the crawler to terminate, rather it keeps polling for new files to be written so it can process them. The middleware extracts the textual content from the HTML pages and the corresponding metadata created by Heitrix. For rich media formats such as Word, PDF, Power Point, YouSeer converts the document into text using Apache TIKA. The output of the middleware is an XML file containing the fields extracted from the documents. The URL of each document serves as the document ID within the index.

Each ingestion plug-in contributes to building this XML file by appending its result as an XML tag. The URL of the ARC file, and the offset of the document within the ARC file are appended to the XML file to expedite retrieval.

The resulting XML file from the processing is posted to the index. After processing all the documents within a single ARC file, the middleware commits the changes to the index and marks the ARC file as processed. While indexing, the word-level n-grams are extracted and added to the query suggestion module.

5. EXPERIMENTS

We perform a number of experiments to measure the performance of our proposed framework. The experiments entail crawling the web by focusing on a set of seed URLs then processing the crawled documents in the ingestion module before they are indexed.

In the first experiment, we aim at creating a search engine for the OpenCoursWare , OCW, [1] courses. We compile a seed list of 50 English speaking universities and crawl the seeds with Heritrix 1.14.4. We set a limit of 100,000 to the maximum number of files that can be downloaded. The job finished after reaching 100,000 documents in 4 hours and 17 minutes running 50 threads. We used an out of the box configuration for Heritrix, and only modified the max number of documents to be fetched. The size of the data crawled by Heritrix was 15 GB compressed into ARC files. For comparison, we use Nutch 1.5 to crawl the same seed list. Similar to Heritrix, we used 50 threads for crawling and keep the rest of the configurations to their default values. We limited the hops to 5 and specify the *topN* value at 20,000. *topN* controls the maximum number of pages that can be downloaded at each level of the crawl. This should limit the entire crawl to roughly 100,000 pages. The job terminated after 9 hours, and downloaded 42806 documents. The size of the entire crawl files was 838 MB, including *segments, linkdb, and crawldb*. We guess that Nutch prioritized crawling small HTML documents over PDF and PPT files.

For both YouSeer and Nutch, we used Apache Solr 3.6 as an indexing engine. We start by running Nutch solr indexer, and monitor the process using *Sysstat* [27], which is a popular tool for monitoring system resources and utilization on Linux. As Nutch does not allow specifying the number of threads for ingestion, unlike YouSeer, we started the ingestion command and monitored the threads long with memory and CPU usage through Sysstat. We recorded 16 active threads ran that under Nutch process during ingestion. The entire ingestions and indexing process took 3.35 minutes, that is 199 URLs/second and around 3.8 MBs/second. On the other hand, since YouSeer allows controlling the number of ingestion threads, we used the same number of threads as reported by Sysstat. YouSeer middleware took 15 minutes to process the 100,000 documents. That is 111 URL/second and around 16 MBs/second. Table 1 summarizes the results for OCW search engine experiment. The CPU and memory usage represent the max usage as captured by Sysstat. The machine on which the experiments was ran is Dell workstations with 2 dual core processors and 4 GB of memory. The CPU usage is normalized by the total number of CPUs.

[1] `http://www.ocwconsortium.org/`

Table 1: Comparison of different parameters for ingesting and indexing OpenCourseWare content

Parameter	YouSeer	Nutch
# docs	100,000	42806
Size	15 GB	838 MB
CPU	0.81%	0.25%
Memory	37.44%	14.37%
Time in minutes	15	3.35
URL / Second	119	199
MB / Sec	16	3.8

This experiments shows how YouSeer framework can ingest larger amount of data per second. And since Heritrix crawl jobs can run faster, plugging in different components of search engines seems to yield faster turn around time and larger processing power supporting our idea of utilizing different open source components rather than building all the pieces of a search engine.

In another experiment, we crawled for 20 million documents with 50 threads, this job took less than 40 wall clock hours. One million documents of multiple formats (pdf, html, ppt, doc, etc.) were indexed in less than 3 hours. These experiments were conducted on a Dell server with 8 processors, 4 cores each and 32 GB RAM, running linux.

6. CONCLUSION AND FUTURE WORK

We described the architecture of YouSeer, a complete open source search engine. The approach used for building YouSeer can be extended to support constructing powerful search tools by leveraging other open source components in such a way that maximizes usability and minimizes redundancy. YouSeer a natural fit for vertical search engines and to the enterprise search domain. It also serves as pedagogical tool for information retrieval and search engine classes. The ease of use and flexibility of modification makes adoptable for research experiments. Our experiments shows that YouSeer can be more effective that other complete open source search engines in certain scenarios.

We enumerated the list of open source libraries that the system uses and introduced a middleware to coordinate these modules. The current version of YouSeer is hosted on SourceForge and a virtual appliance box is available for download to eliminate the installation overhead.

In the future, we plan to introduce modules to parallelize the processing and take advantage of the MapReduce paradigm. We also look forward to investigating security models that would protect the data from being access by unauthorized users. Currently we rely solely on the web server and the operating system to provide security mechanisms.

7. ACKNOWLEDGMENTS

We acknowledge partial funding from NSF and the Information Technology Services at the Pennsylvania State University. We also thank Pradeep Teregowda, Mike Halm, and Kevin Kalupson for their contribution.

8. REFERENCES

[1] http://www.iso.org/iso/news.htm?refid=Ref1255.

[2] Apache hadoop. http://hadoop.apache.org/.

[3] Apache lucene. http://lucene.apache.org/.

[4] Apache solr. http://lucene.apache.org/solr/.

[5] Heritrix. http://crawler.archive.org/.

[6] Hounder on google code. http://code.google.com/p/hounder/.

[7] ht://dig. http://www.htdig.org/.

[8] Indri homepage. http://www.lemurproject.org/indri.php.

[9] Labrador homepage. http://www.dcs.gla.ac.uk/~craigm/labrador/.

[10] Lemure homepage. http://www.lemurproject.org.

[11] Mg4j homepage. http://mg4j.dsi.unimi.it/.

[12] mnogosearch homepage. http://www.mnogosearch.org.

[13] Nutchwax. http://archive-access.sourceforge.net/projects/nutch/.

[14] Omega homepage. http://xapian.org/docs/omega/overview.html.

[15] Sphinx homepage. http://www.sphinxsearch.com.

[16] Swish-e homepage. http://swish-e.org/.

[17] Terrier homepage. http://terrier.org/.

[18] Text retrieval conference (trec). http://trec.nist.gov/.

[19] Tika homepage. http://tika.apache.org/.

[20] Webglimpse homepage. http://webglimpse.net.

[21] Xapian homepage. http://xapian.org.

[22] Zettair homepage. http://www.seg.rmit.edu.au/zettair.

[23] P. Boldi and S. Vigna. MG4J at TREC 2005. In E. M. Voorhees and L. P. Buckland, editors, *The Fourteenth Text REtrieval Conference (TREC 2005) Proceedings*, number SP 500-266 in Special Publications. NIST, 2005. http://mg4j.dsi.unimi.it/.

[24] M. Cafarella and D. Cutting. Building nutch: Open source search. *Queue*, 2(2):61, 2004.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[27] S. Godard. Sysstat: utilities for linux. http://sebastien.godard.pagesperso-orange.fr/.

[28] M. Henzinger, R. Motwani, and C. Silverstein. Challenges in web search engines. In *ACM SIGIR Forum*, volume 36, pages 11–22. ACM, 2002.

[29] R. Khare, D. Cutting, K. Sitaker, and A. Rifkin. Nutch: A flexible and scalable open-source web search engine. *Oregon State University*, 2004.

[30] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23–32, 1994.

[31] C. Middleton and R. Baeza-Yates. A comparison of open source search engines. In *grid-computing*, volume 1, page 1.

[32] G. Mohr, M. Stack, I. Rnitovic, D. Avery, and M. Kimpton. Introduction to heritrix. In *4th International Web Archiving Workshop*, 2004.

[33] P. Ogilvie, , P. Ogilvie, and J. Callan. Experiments using the lemur toolkit. In *In Proceedings of the Tenth Text Retrieval Conference (TREC-10*, pages 103–108, 2002.

[34] C. Olston and M. Najork. Web Crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[35] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and D. Johnson. Terrier information retrieval platform. *Advances in Information Retrieval*, pages 517–519, 2005.

[36] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *Proceedings of ACM SIGIR'06 Workshop on Open Source Information Retrieval (OSIR 2006)*, 2006.

[37] A. Patterson. Why writing your own search engine is hard. *Queue*, 2(2):48, 2004.

[38] J. Rutherglen. Scaling solr to 3 billion documents. http://2010.lucene-eurocon.org/slides/Scaling-Solr-to-3Billion-Documents_Jason-Rutherglen.pdf. Apache Lucene EuroCon 2010.

[39] V. Singh. A comparison of open source search engines. http://zooie.wordpress.com/2009/07/06/a-comparison-of-open- source-search-engines-and-indexing-twitter/, 7 2009.

[40] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: a language-model based search engine for complex queries. Technical report, University of Massachusetts Amherst, 2005.

Towards an Efficient and Effective Search Engine

Andrew Trotman
Department of Computer Science
University of Otago
Dunedin, New Zealand
andrew@cs.otago.ac.nz

Xiang-Fei Jia
Department of Computer Science
University of Otago
Dunedin, New Zealand
fei@cs.otago.ac.nz

Matt Crane
Department of Computer Science
University of Otago
Dunedin, New Zealand
mcrane@cs.otago.ac.nz

ABSTRACT

Building an efficient and effective search engine requires both science and engineering. In this paper, we discuss the ATIRE search engine developed in our research lab, and both the engineering decisions and research questions that have motivated building ATIRE.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing – Indexing methods; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – Search process

General Terms

Algorithms, Performance

Keywords

Indexing, Storage, Efficiency, Pruning, Procrastination

1. INTRODUCTION

Information retrieval has been a hot research topic for decades due to the need to quickly and accurately answer users' queries across very large document collections, for example the web. Building such an efficient and effective search engine involves not only science but also engineering. Science provides a range of algorithms for fast searching and better ranking, and engineering is required so that systems can be tuned to their optimal performance.

There are a number of existing search engines, both proprietary and open-source, for example, Google, MG and Apache Lucene. However, we have built a new search engine called ATIRE from the ground up, to ensure we have a fast robust baseline in order to compare new information retrieval technologies; and to conduct state-of-the-art information retrieval research questions. The ATIRE search engine is a cross-platform search engine — running on Windows, Linux and Mac OSX — written in C/C++, with traditionally non-interoperable sections hand-coded to avoid the use of a third-party abstraction layer.

The questions we want to address are:

- **How to build a fast indexer:** It is very challenging to build a fast indexer due to the complexity of how much work is involved. Our indexer is multi-threaded with a unique pipeline methodology. We also implemented a memory management subsystem in the indexer for fast memory allocation. The indexer also supports the merging of multiple indexes into a single index.
- **What is the most efficient structure for the inverted index?** The full structure of the inverted index is rarely discussed in the literature, previous discussions have only discussed the techniques used for index representation [34]. In this paper, we discuss how we engineered our index structure.
- **How to search efficiently without sacrificing effectiveness?** We have been working on the optimisation of the term-at-a-time approach for query evaluation and for future work this will be used as a baseline for comparing various pruning algorithms; and comparing between term-at-a-time and document-at-at-a-time processing.
- **Does term proximity work?** We question whether term proximity and phrase searching are effective under current evaluation methodologies.
- **Other research questions?** There are a number of other research questions we intend to address in future work: generalisation of our fusion of ranking functions such as BM25 and PageRank; an exploration into the juxtaposition between diversity and relevance feedback; and fully distributed indexing and searching.

SIGIR 2012 Workshop on Open Source Information Retrieval.
August 16, 2012, Portland, Oregon, USA.

2. FAST INDEXING

The experiments and results shown were conducted on a collection of standard collections from both INEX and TREC forums, as described in Table 1. The experiments, with the exception of ClueWeb09 collections, were conducted on a dual quad-core Intel Xeon E5410 2.3GHz, DDR2 PC5300 9GB main memory, Seagate 7200RPM 500GB hard drive, and running Linux with kernel version 2.6.30. The ClueWeb-09 collection experiments were performed on an quad cpu AMD Opteron 6276 2.3GHz 16-core, 512GB PC12800 main memory, 6x 600GB 10000 RPM hard drives, and running Linux with kernel version 2.6.32.

In order to produce an index quickly, the indexer in ATIRE uses several optimisations and a unique pipeline procedure based on the producer/consumer model.

The main optimisation that ATIRE uses when indexing is the use of an internal memory management system that

	Size				Words	
Collection	Collection	Index	%	Documents	Unique	Total
Wall Street Journal [14]	517MB	64MB	12.4%	173,252	229,514	84,881,717
WT10g [4]	10GB	837MB	8.4%	1,692,096	5,512,114	1,348,119,626
2009 INEX Wikipedia [29]	50.7GB	1.6GB	3.2%	2,666,190	11,874,077	2,341,271,195
WT100g/VLC2 [16]	100GB	7.1GB	7.1%	20,616,457	25,250,355	12,690,145,498
.gov2 [9]	400GB	12GB	3%	25,205,179	40,641,599	32,573,784,848
ClueWeb09 Category B	1.5TB	32GB	2%	50,220,423	96,298,556	71,319,689,402
ClueWeb09 Category A	12.5TB			503,903,810		
(excl. 70% spam)	3.8TB	76GB	2%	150,954,279	127,651,335	189,731,940,667

Table 1: Summary of Collections Used

Memory Manager	Indexing Time (mm:ss)
System	14:18
ATIRE	10:37

Table 2: Indexing times for INEX 2009 Wikipedia collection across four .tar.gz files with different memory managers

requests large blocks of memory from the system and divides this up as necessary, this overhead can be measured by compiling without this intermediate manager and using the system memory management. This optimisation alone reduces the time taken to index the 2009 INEX Wikipedia collection by one-third, as shown in Table 2, these times are taken from a single run, with disk caches flushed between runs, but are indicative of typical performance.

The indexing pipeline that ATIRE uses internally is unique among open-source search engines. The pipeline consists of a group of parallel producer/consumer inspired objects that either deal with streams of data, or file-like objects that are created from these streams. Each step in the pipeline is focused on only performing one operation on the passed-through data, minimising the amount of inspection performed at each step.

These different stages in the pipeline can be combined quickly and efficiently to allow new types of content to be indexed. For instance, an existing object that un-tars, and an object that un-gzips can be combined to allow the indexing of .tar.gz files.

Objects in the pipeline are allowed to perform secondary functions, for instance, compressing the original document and including it within the index (for post-processing such as focused retrieval and snippet generation). The input pipeline allows the indexer to filter out documents, such as those identified as spam, and a best-effort attempt to clean incoming data to negate any pre-processing of document collections that might otherwise be necessary. This is motivated by our underlying philosophy that the indexer should be able to index any standard test collection out of the box without any pre-processing.

At the end of the pipeline each document is indexed separately and folded into the overall index. This, combined with the indexing pipeline, allow documents to be indexed completely in parallel on a single computer.

The effect that parallel indexing has on the indexing time is shown in Table 3. The times shown are from a single run, with disk caches flushed between experiments, but are typical times experienced. This table shows that as the number

Number Input Files (.tar.gz format)	Indexing Time (mm:ss)
1	19:35
2	11:47
4	10:37

Table 3: Indexing times for INEX 2009 Wikipedia collection with varying number of input files

Input Format	Indexing Time (mm:ss)
.tar	10:35
.tar.gz	10:37
.tar.lzo	10:09
.tar.bz2	19:10
File count	5:40
Extracted line count	6:54
Individual files	64:30

Table 4: Indexing times for INEX 2009 Wikipedia collection under various compression schemes across four files

of files increases, and our ability to index these files in parallel increases with it, that the total time to index is decreased.

This leads us to believe that our indexer is approaching the point where indexing is bound by decompression time. Indexing times for the INEX 2009 Wikipedia collection when split across four files are shown in Table 4, with the time to index the individual extracted files shown for comparison, as a clearly input bound operation (probably by the ability to open and close files). We are pleased to notice that we have already crossed the point at which we take less than twice the time as simply counting the number of files within the tar file. We also show the total time taken to count the number of lines in extracted files as a target to aim for. We have not yet tuned the number of threads our indexer uses against the number of cores in the machine.

The ATIRE search engine defines a word to be a sequence of characters or numbers, where a character or number is determined by the unicode specification. We currently assume that input is in UTF-8 format, and can process encoding errors that may be encountered such as missing continuation bytes. The input is decomposed, normalised and lower-cased following the unicode specifications. ATIRE supports CJK, and includes chinese segmentation, and has been used in experiments at NTCIR. Currently ATIRE does not support entities such as `&aacute;` and ignores any processing directives contained within the document, except for comments.

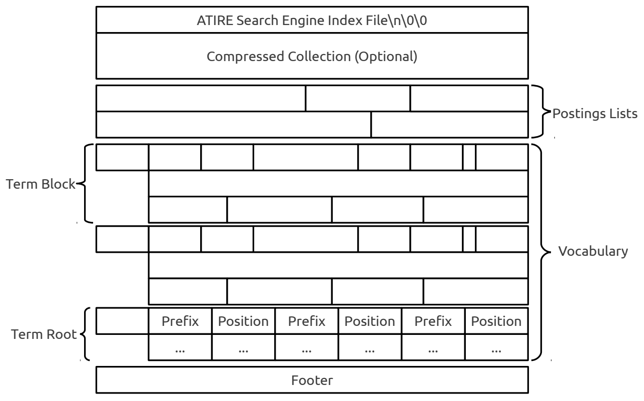

ATIRE Search Engine Index File\n\0\0
Compressed Collection (Optional)
Postings Lists
Term Block
Vocabulary
Term Root
Prefix
Position
Footer

Figure 1: The overall structure of the index file

ATIRE performs all indexing in memory on a single machine, although distributed indexing is being investigated. To work around this limitation, ATIRE also includes a tool to combine previously generated indexes with minimal memory overhead. By processing each indexed term separately the merge tool only requires enough memory to contain the merged postings list, and to maintain the first level of the dictionary structure described below. This allows the indexing of collections that otherwise could not be indexed on commodity hardware, for example the ClueWeb09 Category A collection.

3. INDEX STRUCTURE

The ATIRE search engine generates a single index file that consists of multiple distinct sections. By restricting the index to a single file we minimise the likelihood of a user not having all parts of the index at search time.

The first few bytes of the index file contain the string `ATIRE Search Engine Index File\n\0\0`, so that the file type can be identified by a person using the command `head -n 1`. A diagrammatic overview of the index structure is shown in Figure 1.

The first section of the index file is optional, and contains the compressed original documents in the collection. This feature allows for, among other things, focused retrieval and snippet generation. The location of each compressed document is stored in a special term inside the index.

The second section of the index file contains the postings lists for each of the terms. Traditionally postings lists are stored as a sequence of ⟨docid, term frequency⟩ pairs, ordered by docid. In the ATIRE search engine we instead sort on term frequency first [26, 27], then for each term frequency we store the docids in a difference encoded list, terminated with a 0. This ordering on term frequencies first is referred to as impact ordered.

The ATIRE search engine supports the use of precomputed quantised impact scores, where instead of storing the term frequency values we instead precompute the RSV for each term with respect to each document after indexing is complete [1].

In order to better compress these numbers, they are quantised into integers using the *Uniform* method [1], which preserves the original distribution of numbers, so no additional decoding is required at query time [23, 1]. In the ATIRE search engine, these values are scaled to 1–255, so that they may be stored in one byte.

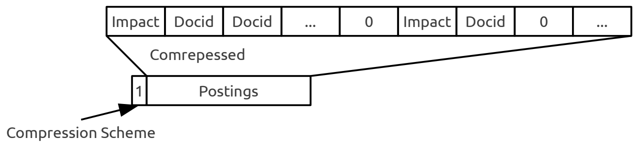


Figure 2: The structure of a postings list

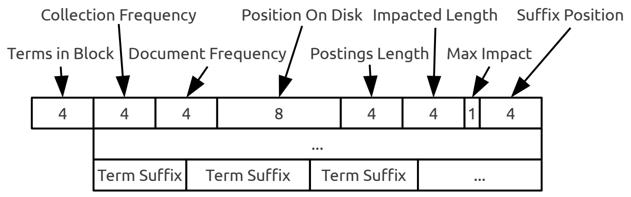


Figure 3: The structure of a vocabulary leaf

Storing the postings lists in this impact ordered format can be thought of as a form of compression, as fewer integers need be stored. In the worst case where every impact value is used, then in an impact ordered format, $D + 510$ integers need to be stored (due to scaling, quantisation and list termination), as opposed to $2D$, where D is the number of documents that contain the term.

We refer to each list of docids for each impact value as a quantum. Each quantum is stored as a difference encoded list, and the entire impact ordered list is then compressed. The ATIRE search engine is capable of compressing postings lists using different compression schemes (carryover 12, elias delta, elias gamma, golomb, none, relative 10, sigma, simple 9, and variable byte) to minimise the disk space taken by the index. By default, however, ATIRE uses variable byte compression. A diagrammatic representation of this structure is shown in Figure 2.

The third section of the index contains the vocabulary structure that holds all the terms that have been indexed. By default, the ATIRE search engine uses the *embedfixed* algorithm [18] to store vocabulary terms since this algorithm provides a good trade-off between storage space and lookup time. The *embedfixed* algorithm stores the vocabulary in a two level B-tree structure. The first level of which contains the unique first four character prefixes of terms in the vocabulary. Each leaf node in the second layer contains the suffixes of those terms that share the common prefix of the parent node. In essence, it is a form of front-encoding that can be searched efficiently.

As well as storing terms in the vocabulary, a number of variables are also required for each term; they are collection and document frequencies used for ranking purposes, location of the postings list stored on disk and the list length used for retrieval of the postings from disk. These variables are stored in the leaf nodes of the vocabulary B-tree.

Aside from these variables associated with each term, extra variables are introduced for each term: the *postings_length* holds the compressed length of the postings list; the *impacted_length* variable stores the number of integers in the decompressed postings list. These values allow our decompression routines to take the form "decompress n integers from this pointer", and by identifying the longest decompressed postings lists (which is stored in the file footer), allocate a single buffer for decompression purposes at search time.

The suffixes for each term inside the leaf node are stored as a null terminated set of strings at the end of the leaf node block. For this reason the *suffix_position* variable identifies where in this block of suffixes the suffix for this individual term begins. The *local_max_impact* holds the maximum impact value for the term, and is used for early termination and pruning of query evaluation [26, 27]. Figure 3 shows a diagrammatic layout of these variables and the number of bytes assigned to them.

As a further space saving we can store the postings lists directly in the vocabulary structure for terms that occur either once or twice in the collection. This can be done by re-purposing some of the variables in the vocabulary leaves, much like a `union` in the C programming language. How to process these lists can be determined at run time by examining the document frequency for the term. It not only saves the storage space for the postings, but also eliminates the extra storage needed for the impact header and postings list header.

The ATIRE search engine has the option of loading indexes completely into memory at search time. In the case that the index is not loaded completely into memory, the vocabulary root is loaded into memory, and during query evaluation is binary searched. The relevant term leaf is then loaded into memory and binary searched to find the term details, then the document frequency is checked. This technique is a form of pre-fetching [20], and saves an extra disk seek and read. Further details of this method are to be published at a later date.

Lastly, the index file contains a footer that contains variables that describe the index, and are used to minimise the number of memory allocations needed when performing query evaluations, such as the length of the longest postings list. These variables are designed to allow the ATIRE search engine to perform no dynamic memory allocation at search time. Certain variables that are associated with the index that are known at indexing time, such as whether the impact values are pre-calculated RSV scores, are stored within the index itself as special terms.

As shown in Table 1, the ATIRE search engine is capable of producing compact indexes that are a fraction of the size of the original collection, the rate of which depends largely on the ratio between indexable and non-indexable content. The ClueWeb09 Category A index was constructed with spam filtering set to discard the 70% most spammiest documents, as suggested by Cormack et. al. [10], with the number of documents included in the index shown in brackets in Table 1.

Table 5 shows some comparisons for indexing and searching times across the ClueWeb09 Category A and B collections. Each index was constructed without quantisation and searching was performed using a single thread, with none of the optimisations discussed below, across queries 101–150.

4. QUERY EVALUATION

There are two main query evaluation methods used in information retrieval systems, *document-at-a-time* and *term-at-a-time* processing. The document-at-a-time approach completely evaluates one document at a time before moving to the next, while the term-at-a-time approach process one query term at a time.

Collection	Index Time (hh:mm:ss)	Search Time Per Query	MAP
ClueWeb09 Cat. B	4:10:20	11.9s	0.1216
ClueWeb09 Cat. A (excl. 70% spam)	20:30:38	30.7s	0.1028

Table 5: Comparison of timings for indexing and searching across the ClueWeb09 collection

There are advantages and disadvantages to the two approaches; (1) term-at-a-time requires an array of intermediate accumulators (one for each document) to hold the accumulated results between the evaluation of each term, while document-at-a-time only needs to hold the top n documents (where n is the number of documents to return). Turtle & Flood [33] state that document-at-a-time is more cost efficient than term-at-a-time based on the assumption that the intermediate accumulators are stored on disk. They state that the performance of the two methods would be equivalent if the accumulators could be stored in memory. (2) Document-at-a-time requires a random scan of postings lists for all the query terms in order to fully evaluate a document. This scan takes time especially if all postings lists cannot be held in memory. Skipping [21] and blocking [22] were introduced to allow pseudo-random access into postings lists. However, there is an extra overhead to build skipping and blocking, and the index size increases. Broder et al. [6] addressed this random scan problem by introducing a new document-at-a-time query processing algorithm called WAND which can smartly skip some unnecessary postings for fast scanning. Ding & Suel [12] further extended the WAND algorithm and introduced Block-Max WAND which can further skip more unnecessary postings. The skipping criteria for both of the algorithms are based on the runtime calculation of current thresholds of the maximum impacts for all query terms. (3) Postings lists for document-at-a-time must be longer because the postings lists are sorted on doc id and are not impact ordered. (4) Intuitively, document-at-a-time is more suitable for conjunctive search while term-at-a-time for disjunctive search.

Most criticisms aimed at term-at-a-time approach are towards the requirement of the intermediate accumulators and the need to sort the accumulators to return the top documents. However, we believe that it is more difficult to manage the memory for all postings lists and efficiently random scan the postings lists for document-at-a-time. We are not concluding that term-at-a-time is better than document-at-a-time, or vice versa. Instead we have built a baseline using the term-at-a-time approach in the ATIRE search engine and will use this baseline to compare and investigate the document-at-a-time approach in future work.

The rest of this section discusses how the issues associate with the term-at-a-time approach are addressed in ATIRE for query evaluation.

4.1 Ranking Functions

By default, ATIRE uses a modified BM25 ranking function. This variant does not result in negative IDF values [1] and is

[1]We thank Shlomo Geva for this contribution.

defined as:

$$RSV_d = \sum_{t \in q} \log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1 + 1)\, tf_{td}}{k_1\left((1-b) + b \times \left(\frac{L_d}{L_{avg}}\right)\right) + tf_{td}}$$

Here, N is the total number of documents, and df_t and tf_{td} are the number of documents containing the term t and the frequency of the term in document d, and L_d and L_{avg} are the length of document d and the average length of all documents. The empirical parameters k_1 and b have been set to 0.9 and 0.4 respectively by training on the INEX 2008 Wikipedia collection.

There are a number of other ranking functions supported in ATIRE as well, for example: Bose-Einstein GL2, Divergence from randomness, Terrier DPH and DFRee, Language Models, and Pregen [30].

4.2 Pruning

The processing (decompression and similarity ranking) of postings and subsequent sorting of accumulators can be computationally expensive, especially when queries contain frequent terms. Processing of these frequent terms not only takes time, but also has little impact on the final ranking results. Postings pruning is a method to eliminate unnecessary processing of postings and provide partial scores for top-k documents. Postings pruning can be done at either index time or query time. Pruning at index time reduces the physical size of the index file [8, 25, 5]. However it is a lossy compression; pruned postings are not kept for access at query time.

Pruning at query time does not modify the index, but prunes postings at run-time during query evaluation. It allows different criteria at query time to be applied to keep track of top k documents. A number of pruning methods have been developed and proved to be efficient [7, 15, 24, 21, 32, 27, 1, 31, 17, 19]. ATIRE supports both pruning at index time and at query time, and pruning at query time is discussed in this section.

In ATIRE, the *heapk* pruning algorithm [17, 19] is used to keep track of the top-k documents. There are two stages in the algorithm. The first stage is the initialisation stage, as shown in Algorithm 1. N is the number of documents in the collection. *top_k* is the number of top documents (specified as a command-line parameter) to be returned. *result_list* keeps track of the number of current top candidate documents during evaluation. *acc* is the accumulator array which hold the intermediate similarity scores for each document. *heapk* is an array of pointers which will be used by the minimum heap to keep track of current top documents. *top_bitstring* is an array of bits (one bit for each document) to track if the document is marked as one of the top candidate documents.

Algorithm 1 Heapk Initialisation

```
Require: $N > 0$ and $lower\_k > 0$
1: $N \leftarrow total\_documents\_in\_collection$
2: $top\_k = lower\_k$
3: $result\_list = 0$
4: $acc \leftarrow$ new $array[N]$
5: $heapk \leftarrow$ new $array[N]$
6: $top\_bitstring \leftarrow$ new $array[N]$
```

The second stage is the update stage, shown in Algorithm 2. There are four steps. First (lines 1 to 3), the score is updated for the accumulator. Second (lines 4 to 12), if the number of the current top candidate documents is less than the required ($result\_list < top\_k$), it means the heap is not full. A new document (if $old\_value = 0$) can be simply added to the heap and the corresponding bit is set. When the heap is full ($result\_list = top\_k$), it is required to build the minimum heap on the *heapk* array. Third (lines 13 to 14), if *result_list* is no less than *top_k* and the current document is marked as set ($top\_bitstring[index]$), it means the document which is already in the top gets updated. Updating one of the top document could violate the properties of the minimum heap. It is necessary to call $min\_update()$ to partially fix the heap. Last (lines 15 to 18), if *result_list* is no less than *top_k* and the score is greater than the smallest score in the heap (which is $heapk[0]$), it means the document, which was not in the top, should now be inserted into the top to replace the smallest score. The bit of the smallest document in heap should be unset. The new document is inserted into the heap by calling *min_insert*.

Instead of repeatedly re-building the minimum heap for update and insertion operations, two special functions are implemented for efficiency optimisation. Every time one of the top candidate documents gets updated, the $min\_update()$ function is called. It first linearly scans the *heapk* array to locate the right pointer and then partially traverses down the subtree of the pointer for proper update of the minimum heap. The linear scan is required because the minimum heap is not a binary search tree. Every time a new document is going to be inserted into the minimum heap, the $min\_insert()$ function is called. It first replaces the document with the smallest score and then partially traverses down the tree for proper update of the minimum heap.

Algorithm 2 Heapk Update

```
Require: $index \geq 0$ and $score > 0$
1: $old\_value \leftarrow$ get the current value of $acc[index]$
2: $acc[index] \leftarrow acc[index] + score$
3: $new\_value \leftarrow$ get the current value of $acc[index]$
4: if $result\_list < top\_k$ then
5:    if $old\_value = 0$ then
6:       $heapk[result\_list] \leftarrow$ address of $acc[index]$
7:       $result\_list \leftarrow result\_list + 1$
8:       set the bit of $top\_bitstring[index]$
9:    end if
10:   if $result\_list = top\_k$ then
11:      build the minimum heap on $heapk$
12:   end if
13: else if $top\_bitstring[index]$ is set then
14:   $min\_update()$ to update the $heapk$
15: else if $new\_value >$ the value of $heapk[0]$ then
16:   unset the bit of $top\_bitstring[heapk[0]]$
17:   $min\_insert(acc[index])$
18:   set the bit of $top\_bitstring[index]$
19: end if
```

The value of *lower_k* can be specified from command-line, used to tell the *heapk* pruning algorithm how many top documents to keep track of.

The performance of the *heapk* pruning algorithm was investigated in INEX 2010 and the results showed that the algorithm is not only CPU cost efficient but also effective. For details of the experiments and results, see our previous work [17].

4.3 Accumulator Initialisation

The term-at-a-time approach uses a number of accumulators, usually as a static array, to hold the intermediate accumulated results for each document. For large collections, there can be a large number of accumulators and it takes time to initialise them. One way to avoid this problem is to use few accumulators allocated using dynamic search structures [24, 21]. However, dynamic structures require more memory space for each accumulator. For example, a balanced Red-Black tree structure [11] uses about 20 and 32 bytes for each accumulator on 32- and 64-bit architectures respectively. Compared with only 4 bytes required in a static array, only 20% for 32-bit (12.5% for 64-bit) or less of the total number of accumulators should be allocated, otherwise the Red-Black tree structure uses more memory than a static array.

For ATIRE, a new efficient accumulator method has been developed. It not only keeps tracks of the top candidates (using the *heapk* algorithm) but also updates the less important accumulators. This allows initially low scoring candidates be to among the top ones at the final stage. The method uses two static arrays. One array is used to hold all accumulators (one for each document) and the other to hold a number of flags. Every flag is associated with a particular subset of the accumulators, indicating the initialisation status for that set of accumulators (either initialised or not). Essentially, we turn the one dimensional array of accumulators into a logical two dimensional table as shown in Figure 4. The dimension of the table is defined by *height* and *width*, and the number of the flags is the same as the height of the table.

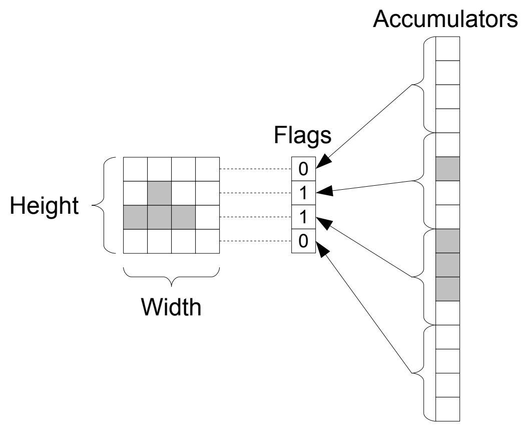


Figure 4: The representation of the accumulators in a logical two dimensional table

As shown in Algorithm 3, the width of the table has to be a whole number (at least 2), and the height can be calculated dynamically by referencing the width and the size of the document collection. Extra accumulators (shown as *padding* in the algorithm) are used to fill the gaps when the number of accumulators is not evenly divisible by the height. The allocation of the extra accumulators are so that we can perform block operation on whole rows. The number of extra accumulators required is usually small (the worst case is $width - 1$).

The update operation for the accumulators is shown in Algorithm 4. First, the index of an accumulator is divided to locate the logical row of the accumulator. Second, the status of the row flag is checked and two outcomes can happen; (1) If the flag has a value of 0, the associated accumulators in the row are initialised and the new value is then added to the accumulator. (2) If the flag has a value of 1, the new value can be simply added to the accumulator.

Algorithm 3 Accumulator Initialisation

```
Require: $width \geq 2$
1: $N \leftarrow total\_documents\_in\_collection$
2: $height \leftarrow (N/width) + 1$
3: $init\_flags \leftarrow$ new $array[height]$
4: initialise $init\_flags$
5: $padding \leftarrow (width * height) - N$
6: $acc \leftarrow$ new $array[N + padding]$
```

Algorithm 4 Accumulator Update

```
Require: $doc\_id \geq 0$ and $doc\_id < N$
1: $row \leftarrow doc\_id/width$
2: if $init\_flags[row] == 0$ then
3:     $init\_flags[row] \leftarrow 1$
4:     initialise the row of the accumulators in $acc$
5: end if
6: $acc[doc\_id] \leftarrow acc[doc\_id] + new\_rsv$
```

In order to find the optimal solution for the width of the table, a mathematical model for the algorithm was described and a simulation was performed. For a detailed discussion of the mathematical model, the experiments and results, please see Jia et al. [19].

4.4 Quantum At a Time

Instead of the traditional approaches of term-at-a-time and document-at-a-time, we propose a new query evaluation approach called *quantum-at-a-time.* Before the start of a query evaluation, all the quanta of the query terms are sorted on their impact values so that the highest impact quanta can be evaluated first and then the next highest, and so on until some of the remaining quanta cannot cause a change to the top-k documents.

The quantum-at-a-time approach is a mixture between term-at-a-time and document-at-a-time. This new approach is similar to score-at-time [2, 3] and Block-Max WAND [12]. The differences are that term ranks are used in score-at-a-time instead of the impact values to sort the quanta, and a block in Block-Max WAND can have postings with different impact values and Block-Max WAND is for document-at-a-time processing.

The quantum-at-a-time approach is targeted for efficient and effective pruning of postings, and better parallel processing of postings lists on multi-core architectures. We will discuss this work in future publications once we have completed it.

5. TERM PROXIMITY

ATIRE does not currently support positional indexes. We have built several search engines in the past and our experiences suggest that positional indexes are not effective under current academic IR evaluation methods that use a binary relevance model. If the precision improvement of a positional index cannot yet be demonstrated in a recognized forum such as TREC or INEX then it is difficult to justify having one.

Our informal reasoning for this is as follows. If the user enters a two word phrase then for a document to contain that phrase it must also contain both words. For a document to contain that phrase many times it must also contain both those words many times. That is, a document that would rank highly for the phrase would also rank highly for both words not as a phrase – and typically they do.

Further, examining the precision-at-1 (P@1) score for both approaches; if phrase searching is more effective than term searching then a specific set of conditions must be met: (1) the term search must not put a relevant document at position 1, and (2) the phrase search must do so. Simply replacing one relevant document with another has no effect on precision; and nor does replacing a non-relevant document with another non-relevant document.

The circumstances necessary for an improvement are hard to meet; but we accept that they can be so. If both words in the phrase are seen frequently in a document, but never as a phrase, then phrase searching should increase precision – in this case the phrase acts as a noise filter. An example of such a query is "The Who". A second example is when all the words are seen but not as the phrase. Again the phrase acts as a filter. An example of this can be seen when searching for the musician "Lisa Lisa" on the Apple iTunes Store.

We believe these examples are pathological and can be handled by storing n-grams in the vocabulary and welcome an evaluation forum running a phrase search track. Such an experiment was conducted at INEX 2009 but was inconclusive ("competitive, but not superior" [13]).

6. RELEVANCE FEEDBACK AND DIVERSIFICATION

The ATIRE search engine currently supports the use of pseudo-relevance feedback. Pseudo-relevance feedback makes the assumption that the top n returned documents are relevant to the query and inspects those documents to identify new and relevant keywords.

In ATIRE we use the KL-divergence for terms inside these top documents to identify the terms which are more likely to be used inside these top documents than would be expected by examining the entire collection. These identified terms are then added to the original query according to Rocchio's algorithm [28]. Terms that were added to the query with this method are given an equal weighting with, and may duplicate, the original terms. The ATIRE search engine allows for other methods for term selection to be incorporated, although currently only KL-divergence is supported.

Although we perform relevance feedback by identifying those terms that are used more frequently in relevant documents than one would expect, it can be thought of as a result of clustering the documents on topic. Relevance feedback promotes those documents that belong to clusters that contain documents that have been identified as relevant.

When a search engine is presented with an ambiguous query, such as "apple", then it can employ diversification in order to help maximise the usefulness of the returned results to the user. Diversification aims to select documents that are related to different possible interpretations of the original query (continuing the above example: the computer company, fruit, record company, etc.) so that each interpretation is given weight according to its likelihood.

One such method for diversification is to cluster the documents by topic, and then select documents from clusters that contain no previously selected documents. Currently the ATIRE search engine does not explicitly diversify results lists, but this is an active research area for us.

When presented as results of clustering documents, relevance feedback and diversification are juxtaposed against each other. Each method uses an opposing criteria to select documents, with diversification exploring cluster-space, and relevance feedback exploiting it. However, both of these ideas seem to improve the results.

7. BUT WAIT THERE'S MORE!

In addition to all the above discussed features, the ATIRE search engine also supports: stemming including Krovetz and Porter as well as soundex and metaphone; topsig; snippet generation and focused retrieval.

The ATIRE search engine can natively read assessment formats, evaluate queries against a large number of metrics, and produce runs for evaluation forums.

8. CONCLUSION AND FUTURE WORK

In future work we aim to change the storage format of postings lists in order to allow quantum-at-a-time processing discussed earlier in Section 4.4. In order to do this, we need to identify where each quantum is stored within a postings list, which will require the use of an impact header. This header structure is in development and we aim to also include support for incremental index updates.

With the ATIRE search engine, we have tackled optimisation of the term-at-a-time processing approach in several areas; (1) The index structure has been optimised. An impact header is created for each postings list for easy manipulation of those lists (sorted on impact values). (2) The *heapk* pruning algorithm is used to keep track of the top-k documents, thus eliminating the need to sort accumulators. (3) The cost of the accumulator initialisation has been minimised by using the logical two dimensional table.

In future work, we will use this baseline to compare with the document-at-a-time and quantum-at-a-time approaches. We will also continue our experiments in relevance feedback, diversification, focused and snippet retrieval in INEX. We hope one of the evaluation forums will run term proximity in the near future.

9. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. pages 35–42, 2001.

[2] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 226–233, New York, NY, USA, 2005. ACM.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*,

SIGIR '06, pages 372–379, New York, NY, USA, 2006. ACM.

[4] P. Bailey, N. Craswell, and D. Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Inf. Process. Manage.*, 39(6):853–871, Nov. 2003.

[5] R. Blanco and A. Barreiro. Probabilistic static pruning of inverted files. *ACM Trans. Inf. Syst.*, 28(1):1–33, 2010.

[6] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 426–434, New York, NY, USA, 2003. ACM.

[7] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. pages 97–110, 1985.

[8] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. pages 43–50, 2001.

[9] C. Clarke, N. Craswell, and I. Soboroff. Overview of the trec 2004 terabyte track. In *Proceedings of TREC*, volume 2004, 2004.

[10] G. Cormack, M. Smucker, and C. Clarke. Efficient and effective spam filtering and re-ranking for large web datasets. *Information retrieval*, 14(5):441–465, 2011.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithm.* The MIT Press, 1990.

[12] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM.

[13] S. Geva, J. Kamps, M. Lethonen, R. Schenkel, J. Thom, and A. Trotman. Overview of the inex 2009 ad hoc track. *Focused Retrieval and Evaluation*, pages 4–25, 2010.

[14] D. Harman. *Overview of the third text retrieval conference (TREC-3)*, volume 500. Diane Pub Co, 1995.

[15] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41:581–589, 1990.

[16] D. Hawking, N. Craswell, and P. Thistlewaite. Overview of trec-7 very large collection track. *NIST SPECIAL PUBLICATION SP*, pages 93–106, 1998.

[17] X.-F. Jia, D. Alexander, V. Wood, and A. Trotman. University of otago at inex 2010. In *INEX '10: Pre-Proceedings of the INEX.* ACM, 2010.

[18] X.-F. Jia, A. Trotman, and J. Holdsworth. Fast search engine vocabulary lookup. In *ADCS '11*, 2011.

[19] X.-F. Jia, A. Trotman, and R. O'keefe. Efficient accumulator initialisation. In *ADCS '10: Proceedings of the Fifteenth Australasian Document Computing Symposium*, 2010.

[20] X.-F. Jia, A. Trotman, R. O'Keefe, and Z. Huang. Application-specific disk I/O optimisation for a search engine. In *PDCAT '08*, pages 399–404, 2008.

[21] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[22] A. Moffat, J. Zobel, and S. T. Klein. Improved inverted file processing for large text databases. pages 162–171, 1995.

[23] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

[24] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

[25] E. S. d. Moura, C. F. d. Santos, B. D. s. d. Araujo, A. S. d. Silva, P. Calado, and M. A. Nascimento. Locality-based pruning methods for web search. *ACM Trans. Inf. Syst.*, 26(2):1–28, 2008.

[26] M. Persin. Document filtering for fast ranking. pages 339–348, 1994.

[27] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[28] J. Rocchio. Relevance feedback in information retrieval. 1971.

[29] R. Schenkel, F. Suchanek, and G. Kasneci. YAWN: A semantically annotated wikipedia xml corpus. March 2007.

[30] N. Sherlock and A. Trotman. Efficient sorting of search results by string attributes. In *ADCS '11*, 2011.

[31] A. Trotman, X.-F. Jia, and S. Geva. Fast and effective focused retrieval. volume 6203 of *Lecture Notes in Computer Science*, pages 229–241. 2010.

[32] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. pages 987–990, 2007.

[33] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6):831 – 850, 1995.

[34] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

SMART: An open source framework for searching the physical world

M-Dyaa Albakour, Craig Macdonald
Iadh Ounis
University of Glasgow, UK
{dyaa, craigm, ounis}@dcs.gla.ac.uk

Aristodemos Pnevmatikakis
John Soldatos
Athens Information Technology, Greece
{apne, jsol}@ait.gr

ABSTRACT

User queries are becoming increasingly local where people are interested in what their friends are up to or what is happening in their local area. Sensors can assist in *localised* information retrieval by giving the search engine direct access to events happening in the local world. In this paper, we describe an open source framework to search in real-time multimedia and social streams. The SMART framework offers a platform to retrieve information from both the physical world and from people interactions on social media. Examples where this framework can be useful include "smart cities" where people can have information needs such as 'what parts of the city has live music on and what do people think about those music events?'. We identify the challenges of building such a framework and the motivations behind releasing it as open source software. The open architecture of the framework brings about possibilities for extending it and deploying it in a wide variety of novel applications.

Keywords

Social search, real-time search, sensor search, smart cities

1. INTRODUCTION

The internet has grown in the last decade to connect a large number of sensing devices that can monitor the physical world such as cameras, microphone arrays, or light sensors. The number of sensors connected to the internet is magnitudes higher than the number of its users [8]. The availability of such connected sensors open opportunities to collect in real-time the status of the physical world and process this information to develop novel applications in the areas of 'smart' cities, social networking, surveillance and security. This has triggered the development of tools and techniques for searching sensor data [5, 6]. However, these methods are still largely based on the indexing and searching of previously defined (and usually textual) metadata. Indeed, while those methods exploit recent advances in sensor ontologies [10] in order to decouple the queries from the low-level details of the underlying sensors, they cannot provide effective search over arbitrary large and diverse sources of multimedia data derived from the physical world.

Moreover, with the emergence of social networks such as Twitter and Facebook, one can envisage situations where information stemming from both social and sensor networks can be combined. In fact, there is a mutual benefit from the convergence of both sensor networks and social networks. Social networks can benefit from the fact that human activity and intent can be directly derived from sensors, which obviates the needs for explicit user input. For example, Foursquare[1] uses smart phone-based GPS and mapping services to enable users to track their friends on the social network platforms. On the other hand, sensor networks could start their cooperation in a social way (i.e. based on information derived from social networks) [4]. For example, think of the query "I want a good restaurant in a place that is lively now". To answer this query, we need a system that can process information about: (i) How *lively* a location is *right now*. Audiovisual crowd analysis can answer that by providing metadata from processing the signals of the connected sensors. These signals should be processed in real-time to provide timely information about the status of the environment in various locations. (ii) How *good* the various restaurants in different areas are. This needs data from social networks (user-generated content by tagging or "liking" good places) and/or from the Linked Data cloud (e.g. restaurant critics) [7].

In this paper, we present our vision for an open source framework where information stemming from large-scale inter-connected sensors and social network streams can be indexed in real-time to facilitate searching the physical world.

2. THE SMART FRAMEWORK

The SMART[2] (Search engine for MultimediA enviRonment generated contenT) framework aims to provide an infrastructure where multimedia sensing devices in the physical world can be easily used to provide information about the status of their environments and make it available in real-time for search in combination with information from social networks. The name SMART acknowledges the vision of "smart cities".

The architecture of the SMART framework is illustrated in Figure 1, where four layers are identified. At the lowest level (physical) we have the sensing devices that provide the physical world data. The *edge node* represents the software layer that processes the raw sensor data to produce metadata about the environment, which is streamed in real-time to the search engine using an appropriate representation (e.g. RDF). Examples of processing algorithms can include

SIGIR 2012 Workshop on Open Source Information Retrieval.
August 16, 2012, Portland, Oregon, USA.

[1]`https://foursquare.com/`
[2]`http://www.smartfp7.eu`

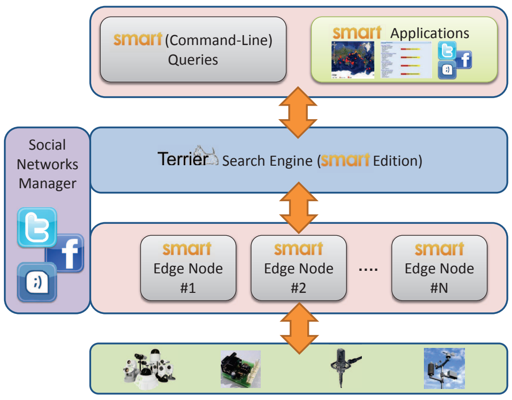


Figure 1: Architecture of the SMART framework

crowd data analysis for video streams and speech recognition in audio streams. The *search layer* collects the streams from the various edge nodes and indexes them in real-time using an efficient distributed index structure. It also employs an event detection and ranking retrieval model that uses features identified in the sensor and social streams to rank events that are relevant to the user queries. Queries can be directly specified or anticipated by the search layer using contextual information about the user, e.g. the user's location or their social profile. Finally, the uppermost *application/visualisation layer* offers reusable APIs to develop applications that can issue queries to the SMART search engine and process or visualise the results.

In the next sections, we will discuss in detail the top three software layers of Figure 1 and the challenges for building each layer. We then describe our vision of the open source SMART framework.

3. EDGE NODE LAYER

The edge node is introduced in this section. First, its aims and challenges are discussed, followed by an example using crowd analysis for metadata extraction.

3.1 Infrastructure Aims

The edge node is the interface of SMART with the physical world. Each edge node can cover sensors from a single geographic area, e.g. a city block or a public square in the city centre. At the edge node, the signal streams, either from physical sensors (e.g. audio/visual streams or environmental measurements), or from social networks, are processed to extract events of interest. To achieve this, the design of the edge node is influenced by: (a) state-of-the-art Internet-of-Things platforms, which typically filter, fuse, combine and reason over multiple data streams and (b) Linked Data technologies. The edge node's architecture is shown in Figure 2.

3.2 Challenges and Research Aims

The data streams collection and processing component provides a uniform way for interfacing to virtually any type of sensor and processing. These include perceptual components (i.e. the algorithms that attempt to interpret the streams' meaning and extract context) running on physical sensors' feeds (such as crowd analysis running on video feeds), as well as social networks filters implemented within the Social Networks Manager (e.g. sentiment analysis of

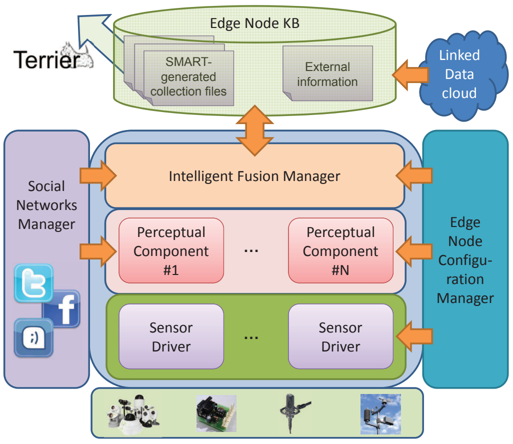


Figure 2: Edge node components

Tweets in the local area of the sensors [1]). This component is empowered by a common unified model for the metadata of the various feeds, which alleviates their heterogeneity while also facilitating their management within the edge node. The approach is similar to that adopted by the Pachube.com[3] platform [3], yet SMART edge nodes provide support for much richer metadata. The development and adoption of a common unified metadata model for all SMART feeds ensures the openness and extensibility of the platform in terms of new sensors, perceptual components and social networking feeds. The output of this component is in the form of continuous metadata streams, which will then need to be processed in real-time. By thresholding a single continuous metadata stream, low-level events are extracted. The moment the threshold is exceeded, the low-level event is signalled as active, while the moment the stream receives a smaller value, the low-level event is signalled as inactive. The Intelligent Fusion Manager subsequently undertakes the rule-driven combination of multiple low-level events (possibly stemming from different sensors or perceptual components). This component integrates sophisticated rule engines that facilitate high-level event recognition on the basis of complex rules over multiple low-level event streams. A rule engine that leverages event calculus techniques [2] is being integrated. The reasoning component leads to a semantic description of the events (i.e. based on the RDF format), which is in line with work undertaken in the scope of semantic sensor networks [15]. The Linked Data component [7] collects data relevant to the edge node that are available as part of the Linked Data cloud. Hence, the information available to the search engine is enriched. For example, it can provide the means for identifying geo-locations [13] associated with the target events.

The edge node stores all types of metadata (the continuous metadata streams, the low-level and the high-level events) in its Knowledge Base using an XML/JSON interface that complies with the aforementioned unified data model. The edge node also provides a web service API (RESTful API) to deliver events to the search engine

3.3 Example Based on Crowd Analysis

Crowds are a main source of events for SMART, especially in a "smart city" setting where citizens are empowered with

[3]`Pachube.com is now Cosm.com`

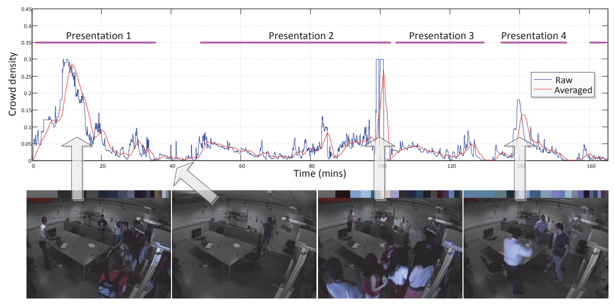


Figure 3: SMART continuous metadata and low-level crowd events as a function of time. Some typically processed frames are also shown.

knowledge about their environment. At a first level of analysis, we are interested in quantifying their density, which is based on an adaptive foreground segmentation. This segmentation is based on a variant of the Stauffer's algorithm [14]. Each pixel in the image is modelled as a Gaussian mixture (GMM), which is updated using a spatio-temporally adapted learning rate [12]. The GMM is then used to decide if the pixel belongs to the foreground by adaptively thresholding the accumulated sorted weight of the Gaussians. Therefore, the foreground mask is formed and is then cleaned-up by shadow removal and a morphological clean-up. The weighted density of foreground pixels gives the crowd density a continuous metadata stream. Using this stream, the low-level events related to crowd appearance or disappearance are easily obtained via thresholding. An example is illustrated in Figure 3, which shows the crowd density metric as a function of time. This has three large peaks, for which the corresponding processed frames indicating the foreground blobs are given. Another frame is an example of minor activity. The extracted low level metadata indicating crowded intervals are also shown as labelled horizontal lines. A sample video demonstrating how the crowd analysis works is available online.[4]

4. SEARCH ENGINE LAYER

The search engine layer is discussed in more detail in this section, where we identify its aims and the main challenges of developing this layer.

4.1 Infrastructure Aims

The SMART search layer indexes in real-time streams of updates from edge nodes and social networks. It is built using the Terrier[5] open source search engine [11] with enhanced real-time indexing and a scalable distributed architecture to handle the large amount of streams. The SMART search layer offers an interface to services and end users to retrieve 'interesting' events and associated relevant posts in the social networks for a given query. While an interesting event is a subjective notion that likely depends on the application, the search layer can make inferences on interestingness, based on how unusual an event is, and learning from training examples of interesting events. In other words, the search engine layer uses the short-term event detection that is performed in the edge node, e.g. the low-level events detected by the crowd analysis algorithms discussed in Section 3.2, to perform unusual event detection across multiple streams of metadata. The search layer should be capable of anticipating user queries depending on their context, e.g. their location or the time of day. The search layer also offers a web service API (RESTful API) to issue queries and view results as real-time mashups aggregated from the various types of processed streams.

4.2 Challenges and Research Aims

One of the main challenges for building the search layer is the *efficient and scalable indexing* of continuous metadata streams. The search layer is built using the open source Storm[6] framework which provides a distributed processing paradigm, similar to MapReduce, that can handle streams of data in real-time. We use this architecture to distribute the workload of indexing the streams using Terrier across multiple processing nodes in a cluster. The index is distributed across various shards and an accumulator keeps track of the global index statistics. Moreover, Terrier has been enhanced to use real-time, in-memory indices, such that as soon as an update from the edge node is received, it is indexed, and made available for search. A demonstrator that uses this infrastructure to search Twitter in real-time is available online.[7]

Another main challenge in the search layer is developing an event retrieval model that can rank 'interesting' events based on a long-term pattern identification of metadata streams. The event retrieval model can make inferences on interestingness, based on how unusual an event is by comparing metadata features, such as the crowd level, observed at a specific location in a specific time to global features observed in similar areas at similar times. For example, a crowded square in a city on a Friday evening is less interesting than a crowded narrow street on a Sunday morning as this will be reflected in the background statistics of the model. Moreover, learning-to-rank retrieval approaches [9] that use features from the sensor metadata (e.g. the crowd level locally and globally) are applied to facilitate the ranking of all events happening in different locations. In addition to features extracted from the sensor metadata streams, textual evidence from the social networks can be associated to events. The event retrieval model can associate keywords to those events. For example, people who tweet about live music in a city's main square may mention the band's name or the song that is being performed.

5. APPLICATION LAYER

The uppermost layer of the SMART platform (see Figure 1) contains the software applications that can deliver the real benefits of the framework to end users. The application layer mainly supports developers who want to create Web 2.0 services or smart phone applications that exploit the framework capabilities. For example, the application layer includes open source web applications that offer user interfaces to issue queries explicitly, or implicitly using the user context, to the search engine API and receive in real-time up-to-date results (events). In addition, it includes open source mashups that use the search layer visualisation APIs

[4]http://www.youtube.com/watch?v=akkyRu68rqE
[5]http://terrier.org
[6]https://github.com/nathanmarz/storm/
[7]http://www.smartfp7.eu/content/twitter-indexing-demo

to display for example newly-breaking events as real-time balloon pop-ups on a map.

6. OPEN SOURCE VISION

SMART is designed as an open source framework, extensible in terms of sensors, multimedia processing components and event retrieval models. As described in Section 4, the main components of the SMART search engine are built upon the existing Terrier open source information retrieval platform, allowing for the real-time indexing and retrieval of multiple and massive-scale sensor and social networks streams. The SMART open source framework is designed to benefit from the power of the open source development philosophy, by enabling application developers and organisations to build new tailored services and products on top of the SMART open source infrastructure.

In particular, SMART will form an open source community for sustaining and evolving its components. It adopts a crowd-sourcing approach to the deployment of physical sensors, social networking feeds and associated repositories, which will become searchable through SMART. Thanks to an open specification for describing data streams, the open source framework facilitates prospective information providers (including sensor infrastructure providers) to connect and contribute edge nodes and data feeds (as described in Section 3) to the SMART search engine. Hence, SMART is designed to integrate a variety of community-based sensor feeds contributed by third parties such as smart cities, sensor deployers and individuals. For example, algorithms that analyse the level of water pollution can be implemented for the corresponding sensors and made available for the fishing industry. Likewise, the SMART open source infrastructure supports virtual sensors streams such as data feeds stemming from social networks (including filters over social networks such as Twitter). In this case, SMART allows social sensors (e.g. gender analysis or sentiment analysis filters on Twitter) to be used while developing applications for smart cities. Finally, SMART adopts the business friendly MPL 2.0 (Mozilla Public License) in order to facilitate service integrators to build custom search applications in response to specific business requirements of their customers (e.g. surveillance applications). The open source application layer makes it easier for such services to be rapidly implemented. In this way, SMART intends to support both a public crowd-sourcing paradigm and a private enterprise-related one. The first release of SMART is planned for the end of 2012.

7. CONCLUSIONS

We introduced an open source unified framework that allows the real-time indexing and retrieval of sensor and social streams. The framework bridges the gap between social and sensor networks and brings them closer together. The framework is currently being developed as part of the EC co-funded project SMART. We presented the research challenges for implementing the various components of the framework. In particular, the main challenges reside in developing a uniform interface to the processing algorithms of the sensor streams, the effective real-time indexing of social and sensor metadata streams and the development of efficient and effective event retrieval models. Releasing the framework as open source software and sustaining an open source community to support it is a strategical decision for a wider spread of this new technology and a wider participation from the industrial and research communities.

Acknowledgments

Part of this work has been carried out in the scope of the EC co-funded project SMART (FP7-287583). The authors acknowledge contributions from all partners of the project.

8. REFERENCES

[1] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. Passonneau. Sentiment Analysis of Twitter Data. In *Proceedings of LSM'11*, 2011.

[2] A. Artikis, M. Sergot, and G. Paliouras. Run-Time Composite Event Recognition. In *Proceedings of DEBS'12*, 2012.

[3] E. Borden. Pachube Internet of Things "Bill of Rights". *http://blog.cosm.com/2011/03/pachube-internet-of-things-bill-of.html*.

[4] J. G. Breslin, S. Decker, M. Hauswirth, G. Hynes, D. Le Phuoc, A. Passant, A. Polleres, C. Rabsch, and V. Reynolds. Integrating Social Networks and Sensor Networks. In *Proceedings of W3C-FSN'09*, 2009.

[5] J. Camp, J. Robinson, C. Steger, and E. Knightly. Measurement Driven Deployment of a Two-tier Urban Mesh Access Network. In *Proceedings of MobiSys'06*, 2006.

[6] D. Guinard and V. Trifa. Towards the Web of Things: Web Mashups for Embedded Devices. In *Proceedings of WWW'09*, 2009.

[7] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[8] A. Jeffries. A Sensor In Every Chicken: Cisco Bets on the Internet of Things. *ReadWriteWeb*, 2009.

[9] Tie-Yan Liu. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[10] D. O'Byrne, R. Brennan, and D. O'Sullivan. Implementing the Draft W3C Semantic Sensor Network Ontology. In *Proceedings of PERCOM Workshops*, 2010.

[11] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *Proceedings of ACM SIGIR-OSIR'06*, 2006.

[12] A. Pnevmatikakis and L. Polymenakos. Robust Estimation of Background for Fixed Cameras. In *Proceedings of CIC'06*, 2006.

[13] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. LinkedGeoData: A Core for a Web of Spatial Open Data. *Semantic Web Journal*, 3(4), 2012.

[14] C. Stauffer and W. E. L. Grimson. Learning Patterns of Activity Using Real-Time Tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):747–757, 2000.

[15] K. Taylor. Semantic Sensor Networks: The W3C SSN-XG Ontology and How to Semantically Enable Real Time Sensor Feeds. In *Proceedings of SemTech'11*, 2011.

First Experiences with TIRA for Reproducible Evaluation in Information Retrieval

Tim Gollub, Steven Burrows, Benno Stein
Bauhaus-Universität Weimar
99421 Weimar, Germany
<first name>.<last name>@uni-weimar.de

ABSTRACT

The verifiability and comparability of computational experiments is a major shortcoming in scientific publications, even at top conferences. In recent years, various services emerged that try to address this problem by providing a global platform where researchers can upload programs along with experiment results. However, these platforms are not well accepted, partly due to their inherent top-down character: a single institution prescribes the formats and technologies to be used. We argue that a community-wide evaluation platform can evolve only from an ongoing bottom-up effort.

For the field of information retrieval we have been undertaking concrete steps to launch and foster this idea with TIRA [4]. Here, we present the concept and an implementation of a web-based experimentation environment that greatly simplifies maintenance and publishing of executable experiments for a research group. TIRA's system architecture retains researcher's full control over their research assets; moreover, no constraints with respect to data formats or programming technologies are prescribed. We see several reasons for researchers to publish their experiments as a web service with TIRA, namely, to simplify their experiment design and execution, to gain credibility, and to easily disseminate results.

This paper reports on experiences from developing TIRA towards our goal. Design goals are reviewed, existing evaluation platforms are analyzed, and the architecture of our current implementation is presented. In particular, we present insights from the first widespread use of TIRA at the PAN series of international plagiarism detection competitions in 2012. Altogether, our review is promising: the design decisions underlying TIRA are both powerful and flexible enough to cope with the widely varying programming preferences of the researchers.

Categories and Subject Descriptors: H.5.3 [Information Systems]: Information Interfaces and Presentation—Group and Organization Interfaces
Keywords: Open Evaluation, Experiment Management, Result Dissemination

1. MOTIVATION

John Ioannidis attracted considerable attention in 2005 with his essay "Why Most Published Research Findings Are False" [5]. Ioannidis argues that research findings published in papers are likely to be biased towards the approaches of the authors, commonly because of selective result reporting and unequal parameter tuning efforts. To improve upon this situation, he concludes that official evaluation initiatives are needed where researchers register their approaches for an objective assessment. In addition, the SWIRL 2012 meeting of 45 information retrieval researchers considered evaluation as a "perennial issue in information retrieval", and that a "community evaluation service" is of specific interest [1]. With initiatives such as TREC[1], CLEF[2], and PAN[3], the information retrieval research community has established evaluation campaigns with great success, with datasets of past campaigns being frequently used for current research.

SIGIR 2012 Workshop on Open Source Information Retrieval.
August 16, 2012, Portland, Oregon, USA.

We see two major limitations of these initiatives that we want to overcome with our open source evaluation platform TIRA (the "Testbed for Information Retrieval Algorithms"). First, it is obvious that the scale of these initiatives cannot address all interesting research questions that arise. To cover the bulk of remaining research questions, a community wide evaluation campaign is needed that is supported by convenient open software. Any researcher must be empowered to easily set up and conduct an evaluation initiative for a specific task of interest. Second, the annual schedule of renowned evaluation initiatives is problematic. In this respect, Armstrong et al. [2] analyzed the performance results achieved outside the official TREC initiative on various TREC collections as published in SIGIR and CIKM papers from 1998 to 2008. The findings showed that the vast majority of these papers are not streamlined with the official TREC results, which in turn leads to a series of false conclusions in the papers and "improvements that don't add up". To avoid the ignorance of existing results, ongoing evaluation initiatives are needed that continuously integrate new results submitted over the Web.

With TIRA, we are developing an open source evaluation platform where we aim to overcome the limitations stated above [3, 4]. The decisive feature of TIRA is that the software can be downloaded by any research group to organize and conduct an evaluation initiative on their local computing infrastructure. For every experiment, TIRA provides a web service through which participants can submit their algorithms or results at any time. TIRA evaluates new submissions automatically by executing the experiment evaluation software provided by the evaluation organizers from the command line of the underlying operating system. All experiment results are stored and indexed in a database, which is queried by the web service to display the current results.

In the remaining sections of the paper, the design goals for TIRA are presented and compared to existing experiment platforms in Section 2, whereas in Section 3 we explain the system architecture of TIRA in detail. In Section 4, we give an experience report of our first significant deployment of TIRA at the PAN plagiarism detection competition, and we provide lessons learned and future recommendations. We then summarize our work in Section 5.

[1]`http://trec.nist.gov`
[2]`http://www.clef-initiative.eu`
[3]`http://pan.webis.de`

2. DESIGN GOALS AND RELATED WORK

Our efforts to make the deployment of TIRA as simple and convenient as possible led to a set of five design goals that we consider as crucial for its widespread use. The design goals are based on the needs for local instantiation, web dissemination, platform independence, result retrieval, and peer to peer collaboration. Our assessment of existing experimentation frameworks with respect to these goals is depicted in Table 1, which shows that none of these systems fully comply.

Table 1: Assessment of existing experimentation frameworks with respect to our five proposed design goals.

Tool	URL	Domain	1	2	3	4	5
evaluatIR	[1]	IR	×	✓	✓	✓	×
expDB	[2]	ML	×	×	×	✓	×
MLComp	[3]	ML	×	✓	×	✓	×
myExperiment	[4]	any	×	✓	✓	✓	×
NEMA	[5]	IR	×	✓	×	✓	×
TunedIT	[6]	ML, DM	✓	✓	×	✓	×
Yahoo Pipes	[7]	Web	×	✓	×	×	×

[1] http://www.evaluatir.org/
[2] http://expdb.cs.kuleuven.be/expdb/
[3] http://www.mlcomp.org/
[4] http://www.myexperiment.org/
[5] http://www.music-ir.org/
[6] http://www.tunedit.org/
[7] http://pipes.yahoo.com/

1. *Local Instantiation.* In case data must be kept confidential, the platform must be able to reside with the data, hence the platform must be locally installable. Unlike centralized experiment platforms like MLComp and myExperiment, local instantiation allows experiments on sensitive data to be published as a service from a local host. External researchers can then use the service for comparison and evaluation of their own research hypotheses, whilst the experiment provider is in full control of the experiment resources.

2. *Web Dissemination.* URLs are definitive identifiers for digital resources. If all runs of an experiment are accessible over a unique URL, researchers can conveniently link the results in a paper with the experiment service used to produce them. Especially for standard pre-processing tasks or evaluations on private data, such a web service can become a frequently cited resource. In addition, attention can be attracted to one's work through integration of the service into home pages and blog articles. To address the issue of digital preservation, URLs should encode all information needed to recompute a resource, such as program and input parameter specifications, in case stored data is lost.

3. *Platform Independence.* The sophisticated and varying software and hardware requirements of information retrieval experiments as well as individual coding preferences of software developers render any development constraints imposed by the experimentation framework critical for its success. Ideally, software developers can deploy experiments as a service unconstrained by the utilized operating system, parallelization paradigm, programming language, or data formats. Local instantiation is one key to realize this goal. Furthermore, the experimentation framework must operate as a layer strictly on top of the experiment software and should use, instead of close intra-process communication such as in TunedIT, standard inter-process communication on the POSIX level and the file system to exchange information. This way, any running software can be deployed as a web service without internal modifications.

4. *Result Retrieval.* Especially for computationally expensive retrieval tasks, the maintenance of a public result repository can become a valuable asset of a research group. For example, experiment services that can index datasets with state-of-the-art natural language processing technology have the potential to raise the comparability of retrieval model research to a higher level. For clustering and result diversification research, comparability is enhanced by establishing static snapshots of the search results from major search engines regularly. The persistent storage of experiment results by the experimentation framework is key to achieve this goal. Even if the public release of an experiment service is not desired, the framework is still useful if it assumes responsibility for managing the raw experiment results and making them available across a research team.

5. *Peer to Peer Collaboration.* Consider a scenario where a consortium of service providers become renowned *gatekeepers* for various streams of research, and maintain the community-wide repository of state-of-the-art algorithms, datasets, and experiment results on their web site. The gatekeepers drive the standardization of data formats and can, by utilizing the retrieval facility, stage competitions in a semi-automated fashion. A mechanism for connecting the local framework instances to a network of experimentation nodes has to be provided to achieve this scenario. Note that currently none of the experimentation platforms implements peer to peer collaboration.

3. SYSTEM ARCHITECTURE

The basic functionality of TIRA is to take a locally executable program and turn it into a web service. To use TIRA for this purpose, the software is first downloaded and instantiated on the local computing infrastructure. System compatibility should not become an issue here, since we distribute TIRA as an executable Java JAR file.[4] For the deployment of new programs, TIRA requires a program specification file in JSON format: the *ProgramRecord*, as shown in Figure 1. In its minimal form, the *ProgramRecord* comprises (1) a unique name for the program, (2) the generic structure of the program execution command, and (3) the value range of each input parameter that affects the output of the program. An example of a generic program execution command and its respective input parameter specification is given in Figure 2. In general, more complex commands are possible that concatenate multiple programs via UNIX-pipes or define parameter substitutions that produce non-terminals (further parameters).

Provided with the information in the *ProgramRecord*, TIRA instantiates and updates all system components that are needed to establish a web service for the new program. All system components are shown in Figure 1. The operating principle of TIRA can be described as two major processes: the front-end process dealing with user interaction, and the back-end process dealing with program execution. As indicated in the component diagram, the *ProgramDatabase* takes on a special role in TIRA's system architecture, since it links the two processes together. The *ProgramDatabase* is instantiated for each *ProgramRecord* individually, it stores past and pending program runs, and it indexes the input parameters of the runs to provide basic retrieval functionality. Note that besides the default local database, TIRA can also connect to a database on a foreign TIRA instance to accomplish peer-to-peer collaboration. The front-end and back-end processes are unaffected by this distinction. In the remainder of this section, the components of these two processes are described beginning with the back-end process first, followed by the front-end process lastly.

The TIRA back-end process involves the *ProgramWrapper* and *ProgramScheduler* system components. For each *ProgramRecord*, an individual *ProgramWrapper* is instantiated to query its asso-

[4]See `http://tira.webis.de` for latest TIRA release information.

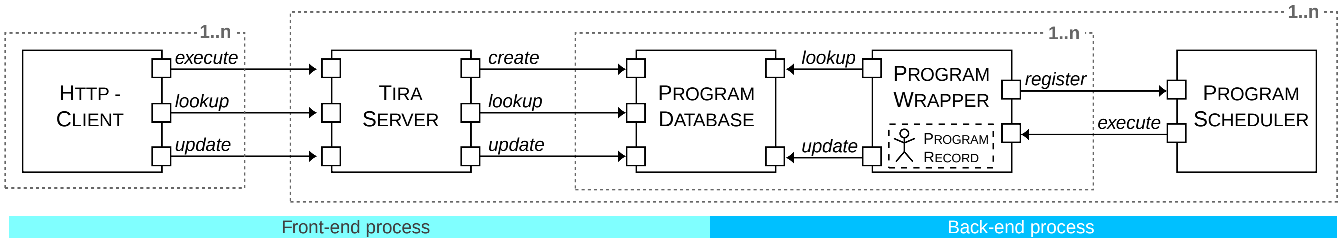


Figure 1: Component diagram of TIRA. Towards the left, the front-end process dealing with the user-interaction is illustrated. To the right, the back-end program execution process is shown. Requests are illustrated by arrows and imply a response from the requested component.

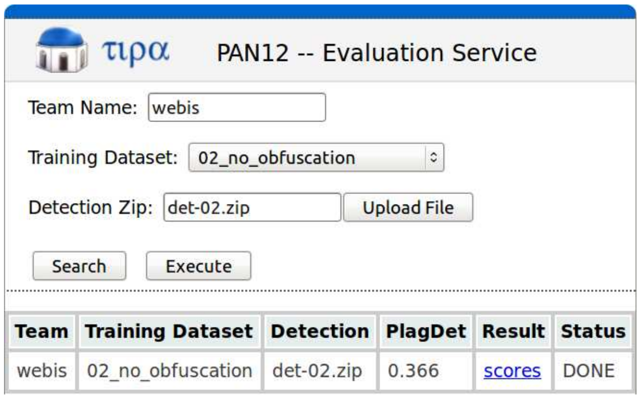
```
python myexp.py $param1 $param2 > result.txt
$param1 —> a | b | c
$param2 —> [0—9]+
```

Figure 2: BNF grammar for a Python program "myexp" with two input parameters for execution in TIRA.

ciated *ProgramDatabase* continuously for pending program runs. Given that TIRA instances might be equipped with different resources in a collaborative environment, the lookup request sent may contain constraints with respect to accepted input parameter values. When a matching program run is received, the *ProgramWrapper* registers this at the *ProgramScheduler* for addition to an execution queue. The *ProgramScheduler* keeps a pool of system threads, which continuously take the next run in the queue and request its execution. To start the program, the generic command in the *ProgramRecord* is substituted with the run-specific values and is called inside a run-specific working directory. During execution, the *ProgramWrapper* listens on the error output stream and updates the database with notifications and results, which then become available to the front-end process.

The TIRA front-end process involves the remaining *TiraServer* and *HttpClient* system components. The *HttpClient* is usually a web browser controlled by a TIRA user, but also a TIRA instance may fill this role to communicate with other TIRA instances. For each *ProgramRecord*, the *HttpClient* can access a web page on the *TiraServer* via a program-specific URL (e.g. `http://<domain>/program/myexp`). A screenshot of a TIRA web page is given in Figure 3. The TIRA web page features the program input parameters as HTML form elements, and offers functionality for retrieving program runs with specific parameter values (Search) and for executing new runs (Execute). The result table at the bottom contains the current execution status, and the results of all executed program runs are displayed. If the value range of an input parameter is specified in the *ProgramRecord* as an enumeration (cf. `$param1` in Figure 2), the input values are listed in a selection box. Otherwise in the case of an intrinsic definition (cf. `$param2` in Figure 2), a text input field is given instead. As a third option, TIRA allows submission files as input parameters, in which case a file upload element is shown to the user.

To retrieve specific program runs, the TIRA user can specify a subset of the input parameters and submit the HTML form by clicking the Search button. The *TiraServer* looks up the database and returns a web page with the matching results. For retrieval requests, the form is submitted using the HTTP GET method, which means that all form values are encoded into the URL. This URL can thus be used for the dissemination of results as discussed in Section 2. In case all input parameters are populated with valid values, the execution of the program can also be requested. Note that the *TiraServer* handles multiple values for parameters by generating an independent program run for each possible combination of values.



PAN12 -- Evaluation Service

Team Name: webis

Training Dataset: 02_no_obfuscation

Detection Zip: det-02.zip

Team	Training Dataset	Detection	PlagDet	Result	Status
webis	02_no_obfuscation	det-02.zip	0.366	scores	DONE

Figure 3: Screenshot of a TIRA web page for the PAN competition 2012. On the web page, PAN participants specify a dataset and upload their plagiarism detection results. On execute, TIRA runs an evaluation script and displays the performance assessment for the submission.

This gives TIRA users a convenient means to execute a series of runs with a single parameter specification. In case a combination of parameter values has not been seen before, a new program run is created with a pending status and stored in the database. Responsibility for the pending run is handed to and executed by the back-end process.

4. ANALYSIS OF TIRA AT PAN

In this section we report on the first deployment of TIRA in an official evaluation campaign. TIRA has been used as the training and evaluation platform for the "detailed comparison" task of the 2012 international PAN plagiarism detection competition.[5] The competition started with the release of training data in March 2012, and officially ended after the evaluation of the participant submissions in July 2012. For TIRA, its successful deployment in the challenge constitutes an important milestone and was an excellent opportunity to analyze the software under realistic conditions.

The participants of the PAN "detailed comparison" challenge were asked to develop software capable of solving the following task: Given a suspicious document and a potential source document pair, extract and record all plagiarized passages from the suspicious document and the corresponding source passages from the source document. Unlike the previous PAN competitions, the participants of 2012 did not submit their detection results on an unlabeled test set, but instead submitted their software. This strategy allowed a set of real plagiarism cases subject to non-disclosure to be incorporated into the test set to improve the authenticity of the evaluation. In addition, the organizers could evaluate the runtime characteristics of the submitted approaches for the first time.

[5]`http://pan.webis.de/`

Two TIRA services were deployed to support the running of the competition: (1) A service to compute performance scores on the training data, and (2) A service for the evaluation of the software submissions on the private test set. We now describe how TIRA has been used in each of these settings in the remainder of this section.

4.1 Training Phase Evaluation Service

For the training phase, the organizers released a dataset with ground truth to be used by the participants to train their approaches. A TIRA service was provided to evaluate the performance of an approach using the training set. On the TIRA service web page, participants were able to upload their compressed detection results and receive the "PlagDet" performance score in return (cf. Figure 3), which combines aspects of precision, recall, and granularity. To compute PlagDet, the compressed submissions were extracted and evaluated with a Python implementation of PlagDet. The generic execution command used by TIRA for the evaluation was hence: "`unzip -qq -o $det -d det && python perfmeasures.py -p $truth -d det > scores.txt`". The parameters starting with a $-symbol were substituted according to the data provided in the web page input fields similar to the example described in Section 3 and Figure 2.

For the organizers of PAN, the evaluation service provided some feedback about the progress of the participants. In the past competitions, the organizers observed that the majority of participants started working seriously only in the few days before the submission deadline. With the public evaluation service, we hoped to create an atmosphere where participants were motivated by the recorded PlagDet scores to date acting as a leader board. One week prior to the submission deadline, the evaluation service received 12 submissions from two of the eleven final participants. Three further participants started making submissions in the final week, resulting in 38 computed PlagDet scores altogether. The remaining six participants did not use the training phase evaluation service, and may have simply elected to evaluate their training results offline. Although the TIRA service was a useful tool for the participants, we learned that further incentives for its usage must be provided to effectively foster the early tinkering within the competition.

4.2 Test Phase Evaluation Service

In the test phase, TIRA was used to organize and conduct the evaluation of the submitted programs. In total, we received eleven plagiarism detection programs for evaluation on the hidden test set. Coincidentally, eleven "external detection" result sets were submitted in 2011 [6], suggesting that the submission of software was an acceptable demand of the participants. The software received varied greatly with respect to its size, runtime performance, and programming language used, and we received submissions for both Windows and Linux operating systems. In this respect, the system independence of TIRA has been successfully demonstrated. We managed to get all submitted software running, and with the exception of one submission, the output files produced were valid. For each of the submissions, we created a *ProgramRecord* based on the installation manual provided by the participants. Although the programs sometimes demanded inconvenient input specifications for processing the test data, the powerful parameter substitution mechanism of TIRA made the task achievable. To evaluate each submission against the test set, we implemented an additional TIRA service that sends an execution request for every document pair in the test set to the TIRA service of the submission. Here, the web dissemination capability of TIRA is highly convenient.

In the near future we plan to give the PAN 2012 participants the opportunity to opt-in for a public release of their plagiarism detection software as a TIRA service on our computing infrastructure. For future evaluation initiatives, we aim to develop an automated program deployment mechanism for TIRA: Participants download the evaluation resources for a competition and deploy them on a local TIRA instance. Once developing and testing on the local TIRA instance is done, TIRA sends the final *ProgramRecord* and software to the official TIRA evaluation instance, where it is automatically deployed and evaluated.

5. SUMMARY

Creating fully reproducible and comparable experiments in information retrieval is highly desirable, and various researchers have pointed out that advances in the state of the art in this field are difficult to account without such an achievement. A software service that meets this challenge and that is accepted within the research community must provide features such as local instantiation, web dissemination, platform independence, result retrieval, and peer to peer collaboration. The TIRA platform addresses these goals as a new web service to organize and operationalize specific programmable tasks runnable on the command line. Recently, TIRA has been deployed "in the wild" for the PAN series of international plagiarism detection competitions. Our preliminary findings are positive: even complex evaluations of software submissions can be easily managed, compared, and published. Based on this experience we aim to further develop TIRA towards a convenient tool for the information retrieval community to conduct evaluation initiatives.

Acknowledgements

The related work and architecture description (Sections 2 and 3) are reproduced from our paper "TIRA: Configuring, Executing, and Disseminating Information Retrieval Experiments" [4].

References

[1] J. Allan, W. B. Croft, A. Moffat, and M. Sanderson. Frontiers, Challenges, and Opportunities for Information Retrieval: Report from SWIRL 2012 The Second Strategic Workshop on Information Retrieval in Lorne. *SIGIR Forum*, 46(1):2–32, May 2012.

[2] T. G. Armstrong, A. Moffat, W. Webber, and J. Zobel. Improvements that don't add up: Ad-hoc Retrieval Results since 1998. In D. W.-L. Cheung, I.-Y. Song, W. W. Chu, X. Hu, and J. J. Lin, editors, *Proceedings of the Eighteenth ACM Conference on Information and Knowledge Management*, pages 601–610, Hong Kong, China, Nov. 2009.

[3] T. Gollub, B. Stein, and S. Burrows. Ousting Ivory Tower Research: Towards a Web Framework for Providing Experiments as a Service. In B. Hersh, J. Callan, Y. Maarek, and M. Sanderson, editors, *Proceedings of the Thirty-Fifth International ACM Conference on Research and Development in Information Retrieval (to appear)*, Aug. 2012.

[4] T. Gollub, B. Stein, S. Burrows, and D. Hoppe. TIRA: Configuring, Executing, and Disseminating Information Retrieval Experiments. In A. M. Tjoa, S. Liddle, K.-D. Schewe, and X. Zhou, editors, *Proceedings of the Ninth International Workshop on Text-based Information Retrieval at DEXA (to appear)*, Sept. 2012.

[5] J. P. A. Ioannidis. Why Most Published Research Findings Are False. *PLoS Medicine*, 2(8):696–701, Aug. 2005.

[6] M. Potthast. *Technologies for Reusing Text from the Web*. PhD Thesis, Bauhaus-Universität Weimar, Dec. 2011.

Design, implementation and experiment of a *YeSQL* Web Crawler

Pierre Jourlin
Laboratoire d'Informatique d'Avignon
Université d'Avignon et des pays de Vaucluse
BP 1228, 84911 AVIGNON CEDEX, France
Pierre.Jourlin@univ-avignon.fr

Romain Deveaud
Laboratoire d'Informatique d'Avignon
Université d'Avignon et des pays de Vaucluse
BP 1228, 84911 AVIGNON CEDEX, France
Romain.Deveaud@univ-avignon.fr

Eric Sanjuan-Ibekwe
Laboratoire d'Informatique d'Avignon
Université d'Avignon et des pays de Vaucluse
BP 1228, 84911 AVIGNON CEDEX, France
Eric.Sanjuan@univ-avignon.fr

Jean-Marc Francony
UMR PACTE
Université Pierre Mendès France - Grenoble 2
Grenoble, France
jeanmarc.francony@umrpacte.fr

Françoise Papa
UMR PACTE
Université Pierre Mendès France - Grenoble 2
Grenoble, France
francoise.papa@umrpacte.fr

ABSTRACT

We describe a novel, "focusable", scalable, distributed web crawler based on GNU/Linux and PostgreSQL that we designed to be easily extendible and which we have released under a GNU public licence. We also report a first use case related to an analysis of Twitter's streams about the french 2012 presidential elections and the URL's it contains.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms ; Design ; Experimentation

Keywords

Web Crawler; Web Robot; Web Spider; PostgreSQL ; Twitter ; Web ; Social Networks

1. INTRODUCTION

Where scalability is concerned, Apache Nutch®[1] and Heritrix[2] are probably the best-known and the most-accomplished open-source web crawlers. They both are sensible choice for Information Retrieval (IR) researchers who intend to build large web corpora. They can be configured to specific needs and can be extended and modified. However, the Java-language source code[3] of these two software toolkits are rather large and complex: 29349 lines of source code for Apache Nutch (v1.4) and 107377 for Heritrix (v3.1.0). Another possible drawback from the researcher's perspective is that they both access the data using unconventional systems : Nutch relies on Hadoop™and Heritrix relies on its own code for handling Internet Archive ARC files.

[1]http://nutch.apache.org/
[2]https://webarchive.jira.com/browse/HER

These systems belong to the "NoSQL" or "UnQL" approaches, supported by the assumption that the widely used SQL relational database standard is a inherent cause of scalability issues. However, this assumption is contested by several database experts. For instance, recent developments around the PostgreSQL project allow it to perform as well as- and sometimes outperform some - NoSQL databases[3]. This alternative approach has been named YesQL.

By taking profit of the capabilities of a PostgreSQL server, we implemented our web crawler in a total of only 911 lines of C-language code and 200 lines of SQL and PL/pgSQL. At the time this article was written and as far as we know, this is the only available web crawler that is based on PostgreSQL. The tests we performed have shown that instances of the crawler could process over 20 millions of URLs in a few days without beeing noticeably slowed by database operations. We thus believe this web crawler is well worth considering by IR researchers and programmers.

2. SOFTWARE DESCRIPTION

The source code repository is located at GitHub[4] under a GNU public license. Everyone can therefore easily download an up-to-date version of the toolkit, provide user's feedback, or join the developer's team. The crawling system can be briefly summarized as follows:

[3]There are alternatives written in Python, e.g. : Mechanize (36419 lines of code) and Scrapy (23096 lines of code)
[4]https://github.com/jourlin/WebCrawler

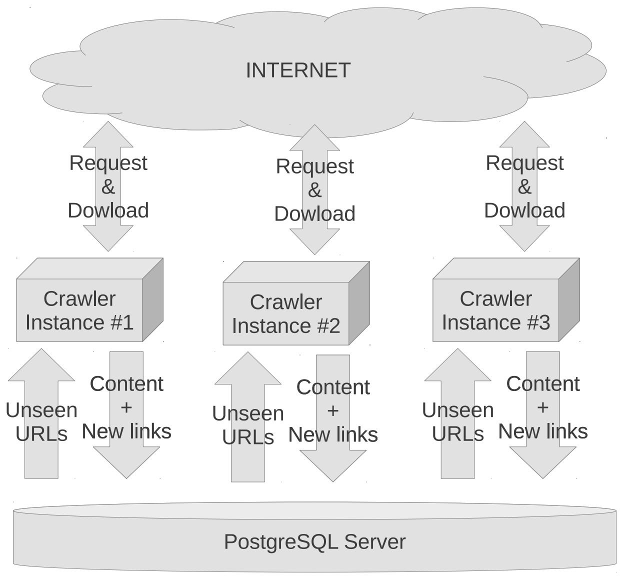


Figure 1: Web crawler organisation

- Links and URLs' data are stored in a PostgreSQL[5] database.
- The user can launch several crawler's instances on several, possibly distant machines.
- Each instance of the crawler iteratively:
  1. fetches a list of URLs to be explored by sending a simple SQL query to the database;
  2. downloads the web pages;
  3. extracts new hypertext links to possibly new URLs;
  4. sends the new data back to the server.

Figure 1 shows how the communication between internet, web crawler's instances and the PostgreSQL server.

The choice of URLs to be fetched is made by one SQL query and two PL/pgSQL additive scoring functions: one scores the URL according to its content, the other scores the URL according to the textual context in which they are linked. The programmer can thus easily implement any *focused* crawling strategy by modifying a single SQL fetch query and two scoring functions. The user can write them in PL/pgSQL in order to take advantage for instance, of PostgreSQL regular expressions. In order to achieve even better performance, he might also write them in C-language and take benefit of PostgreSQL's dynamic loadable objects capability. Figures 2 and 3 show a scoring function in PL/pgSQL that calculates a weighted count of keywords occuring in the URL itself (Figure 2) or in the anchor text that links to it (Figure 3).

Each crawler instance is only responsible for downloading and processing web pages. The downloading stage is per-

[5]http://www.postgresql.org/

```
CREATE OR REPLACE FUNCTION
ScoreURL(url url) RETURNS bigint AS
$$
DECLARE
score INT;
normurl TEXT;
BEGIN
normurl=normalize(CAST(url AS text));
IF CAST(url_top(url) AS TEXT) ='fr' THEN
score=1;
ELSE
score=0;
END IF;
IF substring(normurl, 'keyword1') IS NOT NULL THEN
score=score+2;
END IF;
IF substring(normurl, 'keyword2') IS NOT NULL THEN
score=score+1;
END IF;
RETURN score;
END;
$$ LANGUAGE plpgsql;
```

Figure 2: A Webcrawler strategy written in PL/PGSQL : scoring URLs

```
CREATE OR REPLACE FUNCTION
ScoreLink(context text) RETURNS int AS
$$
DECLARE
score INT;
normcontext TEXT;
BEGIN
normcontext=normalize(context);
score=0;
IF (substring(normcontext, 'keyword1') IS NOT NULL) THEN
score = score +1;
END IF;
IF  (substring(normcontext, 'keyword2') IS NOT NULL) THEN
score = score +1;
END IF;
RETURN score;
END;
$$ LANGUAGE plpgsql;
```

Figure 3: A Webcrawler strategy written in PL/PGSQL : scoring links

formed by the very mature *GNU/Wget* utility[6]. The database system is responsible for the coordination of multiple crawlers (thanks to SQL transactions), uniqueness of stored URLs and links (thanks to SQL constraints), crawling strategy (thanks to PL/pgSQL or C functions), etc. Insertions into a single SQL view triggers insertions into the more complex internal table structure.

3. USE CASE: COVERAGE OF "TWEETED" URLS

3.1 Context

Recent open free network visualisation tools have made easier the qualitative analysis of large social networks[1]. Based on these tools, scientists in humanities can visualize large relational data which lead to new hypothesis that will require further network crawling and data extraction. We show an example of such interaction between humanities and computer scientists made possible by our YeSQL crawler.

Political scientists have formulated the hypothesis that for the 2012 French presidential elections, candidates' communication departments accepted Twitter as a target media and integrated it to their communication system.

Their strategy was to better control their communication and to improve the dissemination of political messages they convey, in order to influence public opinion. What was at stake ? The saturation and the meshing of the media sphere, with coherent messages whatever the channel of dissemination they choose.

The empowerment of their communication during the campaign was linked to their capacity :

- to consolidate their network of opinion leaders thanks to Twitter,
- to be more reactive and to communicate “just in time” if unexpected events occur,
- to strengthen the efficiency of their activists network.

As a consequence, the relationships between their different communication devices has to be analysed.

3.2 Experiment

In order to evaluate this hypothesis, we conducted a capture of Twitter's messages and a parallel though independent web crawl of candidate web sites and newspaper's political pages. We then attempted to compare the two data sources. Twitter's markers (e.g. '#' and '@') facilitates the production of statistics on a given collection. Regarding the web, drawing statistics require a very well structured crawl, with good identification of identical URL and page contents. The YeSQL web crawler proved to be well suited to this task.

By filtering tweets from candidates, to candidates or mentioning a candidate (e.g. @fhollande, @bayrou, @melanchon2012, @SARKOZY_2012, etc.), we recorded 93592 tweets from february 6th at 00:00am to february 13th 2012 at 00:00am.

[6]http://www.gnu.org/software/wget/

Depth	# crawled URLs	% URLs covered (a)	% URLs covered (b)
0	2	0.00	0.00
1	34	0.08	1.00
2	1026	0.73	4.00
3	8543	1.84	8.00
4	56883	3.06	12.00
5	368247	7.33	27.00
6	2756671	15.28	40.00

Table 1: Tweeted URLs' coverage. (a): for all 4777 tweeted URLs ; (b): for the top 100 most frequently tweeted URLs. “Depth” is the minimum number of hyperlinks that one has to follow to reach an URL from the initial set.

26638 of those tweets contained a shortened URL (28.4%) from a set of 10447 unique shortened URL corresponding to 4777 unique effective URLs.

This filtering produced a homogeneous corpus based on a usage logic and identical annunciation rules. The reference to candidates' addresses produces a multi-voiced discourse folded up on the proper space of Twitter. Each “tweeted” URL is functioning as an interface with the outside of this space and brings back external information from the media space. Their identification is important as a marker of discourse evolution and also for its anchorage in the media and political topicality .

Independently from this collection, we started a web crawler instance that was allowed to download 20 pages in parallel, from february 20th at 00:00am to february 26th at 10:55pm. It was initiated on 32 initial URLs from newspapers' political pages and candidates' web sites and collected over 2.7 millions of URLs. In the following tables, we call ”depth” the minimum number of links needed to navigate from an initial URL (depth=0) to a crawled URL.

Table 1 shows the proportion of tweeted effective URLs that were crawled during this period. The fourth column shows that most frequently tweeted URLs are more likely to be covered by the crawl. These results show that most popular URLs have a significant probability to be directly retrieved by the crawler after millions of URLs have been crawled.

Figure 4 shows the proportion of tweeted URLs found in the crawling per tweeted frequency (number of times that the URL was tweeted). This gives an estimation of the crawling coverage with regard to URL's visibility.

Table 2 shows that the similar problem of tweeted domains instead of tweeted URLs is substantially easier. Indeed, the coverage is noticeably higher when only the URL's domains are considered. In particular, 100 most tweeted domains are almost totally (97.73%) covered by the web crawl.

More generally, we can observe that high “domain” coverage figures are obtained for relatively low “depth” levels. This suggests that the most popular URLs originates from sites that are the nearest neighbours of the 32 initial newspapers' political pages and candidates' web sites.

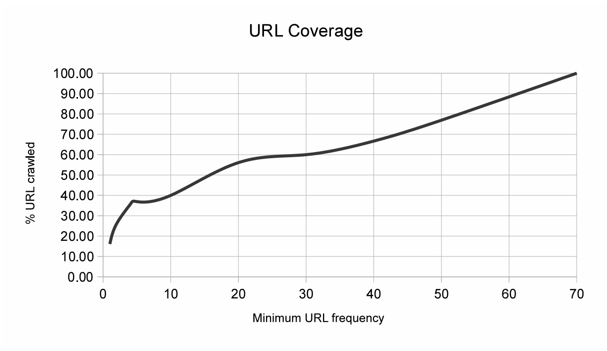


Figure 4: Crawler coverage per tweeted URL frequency.

Depth	# crawled domains	% domains covered (a)	% domains covered (b)
0	1	0.00	0.00
1	31	2.50	18.18
2	95	4.43	29.55
3	312	11.93	50.00
4	1596	27.73	81.82
5	8137	49.66	95.45
6	45992	72.50	97.73

Table 2: Tweeted URLs' domain coverage. (a): for all 4777 tweeted URLs ; (b): for the top 100 most frequently tweeted URLs. "Depth" is the minimum number of hyperlinks that one has to follow to reach an URL from the initial set.

This is not surprising considering that the web of political blogs is stable along month periods [2]. Moreover, all main French newspaper offer a blog service to their readers. The readers contributions to their websites allow them to capture most of the queries on the web dealing with politics.

Results in Table 2 also allow us to expect much better coverage of URLs by simply launching more crawler's instances, on a single or on multiple machines.

4. CONCLUSION

The web crawler we presented does not have all the functionalities that offer older and more ambitious projects such as Nutch and Heritrix. However, we have shown that recent functionalities introduced in PostGreSQL about data structures, triggers and language programming allow to develop powerful web mining tools that can deal with highly redundant data as well as less frequent signals. We illustrated this with a scalable crawler that can explore web networks at a fine grained level. In particular, this crawler can help in comparing the web to social networks like Twitter.

In this particular configuration and for this domain, current events about the french electoral campaign irrigates the two information spaces, the web and Twitter. The practice of "tweeting" URLs becomes usual in the context of modern approaches of information reporting and monitoring.

As we entered this field of investigation by studying the political "actors", we saw that a significant part of original informations are produced, published and tweeted by these actors.

We could also question the existence of significant reporting practices outside the control of political apparatus' dissemination strategies. If our results are confirmed in finer grain analysis, we will be able to reconsider the self-organising hypothesis that people tend to associate to social networks.

5. REFERENCES

[1] W. W. Cohen and S. Gosling, editors. *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010*. The AAAI Press, 2010.

[2] J.-P. Cointet and C. Roth. Local networks, local topics: Structural and semantic proximity in blogspace. In Cohen and Gosling [1].

[3] G. M. Roy. Perspectives on NoSQL. In *PGcon 2010: PostgreSQL Conference for Users and Developers*, Ottawa, Canada, 2012.

From Puppy to Maturity: Experiences in Developing Terrier

Craig Macdonald, Richard McCreadie,
Rodrygo L.T. Santos, and Iadh Ounis
terrier@dcs.gla.ac.uk
School of Computing Science
University of Glasgow
G12 8QQ, Glasgow, UK

ABSTRACT

The Terrier information retrieval (IR) platform, maintained by the University of Glasgow, has been open sourced since 2004. Open source IR platforms are vital to the research community, as they provide state-of-the-art baselines and structures, thereby alleviating the need to 'reinvent the wheel'. Moreover, the open source nature of Terrier is critical, since it enables researchers to build their own unique research on top of it rather than treating it as a black box. In this position paper, we describe our experiences in developing the Terrier platform for the community. Furthermore, we discuss the vision for Terrier over the next few years and provide a roadmap for the future.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

1. INTRODUCTION

The origins of the Terrier open source information retrieval (IR) platform[1] within the University of Glasgow can be traced back to 2001, when it was first created in Java to provide a common basis for research students to use for their PhD research. Since then, the platform has grown to be a scalable and mature open source platform released under the Mozilla Public License (MPL)[2], aimed at researchers and practitioners, permitting the rapid and effective research and development of information retrieval technologies.

Two of the key goals of Terrier are to be flexible and extensible, such that the platform can act as a corner-stone upon which both the academic community and practitioners can build. To this end, Terrier follows a modular design, whereby different components of the indexing and retrieval process can be customised. For instance, when working with Twitter, it can be advantageous to use a dedicated tokeniser that removes URLs and mentions from the text. By combining a modular design with an open source nature, Terrier can be adapted for a potentially unlimited number of use-cases.

However, it is also important to provide as much functionality out-of-the-box as possible. Indeed, Terrier strives to provide state-of-the-art efficient indexing and effective retrieval mechanisms. For example, due to its modularity, Terrier allows various ways of changing the ranking of documents, providing a huge variety of weighting models, including Okapi BM25 [17], language modelling [10, 23] and a vast number of models from the Divergence from Randomness framework [2]. It also includes field-based document weighting models for tackling more structured documents, such as BM25F [22] or PL2F [13]. As a result, Terrier is well known within IR evaluation forums such as TREC and FIRE as the basis of many effective systems.

Over the last decade, the Terrier platform has been developed and enhanced by a wide range of academics world-wide. Indeed, contributions made to Terrier have been driven by the desire to explore new research directions, and will remain so in the future. For instance, Terrier is currently being extended to tackle the new challenges of real-time search tasks, e.g. incremental indexing, live search and reproducibility of experimentation in such dynamic search scenarios. However, open source platforms require significant investments in time and manpower to develop and maintain. While such investments may not directly lead to research outcomes, they are of utmost importance for the research community. For this reason, in this paper, we detail our experiences in developing Terrier and provide a roadmap for future releases of the platform. In this way, the contributions of this position paper are two-fold: we describe the growth of the Terrier platform along the past decade and its latest developments, followed by a roadmap for the next generation of the open source search platform.

This paper is structured as follows. Section 2 discusses the philosophy that guides the development of Terrier. Section 3 describes recent developments of the platform, whereas Section 4 provides a roadmap for future developments. Finally, Section 5 provides our concluding remarks.

2. BUILDING FOR SUCCESS

Our overriding belief behind Terrier is that an information retrieval (IR) system 'should just work... out-of-the-box.' Many users will come to Terrier, and their first experience using the platform is key—we want to ensure that the crucial first experience facilitates the aim that they have for using the platform. In particular, we identify four dimensions that are key to the user experience, namely:

- Effectiveness: The platform should be effective by default. Moreover, it should provide easy access to the accepted state-of-the-art techniques, permitting experiments to be conducted with minimal development cost.

[1]`http://terrier.org`
[2]`http://www.mozilla.org/MPL/`

SIGIR 2012 Workshop on Open Source Information Retrieval.
ACM August 16, 2012, Portland, Oregon, USA.

- Efficiency: Scientific experiments in information retrieval have advantages over other scientific fields, in that experimental ground truths (i.e. relevance assessments) are generally reusable. While the efficiency of an IR research platform is not paramount, we believe that the system should be generally efficient so as to facilitate fast experiment iterations.
- Scalability: The size of IR test corpora has grown 500-fold since the first open source release of Terrier, as illustrated in Figure 1. Regardless of the scale of hardware resources available to a researcher, we believe that the IR system should be able to index and retrieve without challenging configuration.
- Adaptability: It must be possible to adapt the system to new requirements, whether this encompasses new retrieval strategies, new corpora, or different experimental paradigms.

These dimensions (which we collectively denote EESA) underlie the decisions behind the development of the Terrier platform, and the plans for its future. In particular, with Terrier, we aim to ensure that indexing and retrieval can be effectively performed by making efficient use of whatever resource is available, be it a single machine or a large cluster.

An important choice in Terrier has centred around the effectiveness/efficiency/adaptability tradeoff. In particular, IR researchers may not know a priori which particular weighting model they intend to use during retrieval time. Indeed, some retrieval approaches decide on the choice of weighting model on a per-query basis (e.g. [8]). For this reason, we do not believe that efficient retrieval approaches (e.g. score-at-a-time [3]) that tie the index to a particular retrieval approach are suitable for an experimental IR platform, even if they can produce marked efficiency improvements.

On the other hand, there are software engineering challenges in maintaining and improving a large platform such as Terrier. For instance, since Terrier 3.0 we have been following a testing regime that ensures that changes do not impact on correctness (unit tests) and effectiveness (end-to-end tests). In the following, we identify some specific improvements made to the Terrier over the last few years and how these are related to the EESA dimensions, before going on to identify a roadmap for further developments of Terrier.

3. RECENT IMPROVEMENTS

In this section, we highlight recent improvements in Terrier, covering both its indexing and retrieval architectures.

3.1 Indexing

Since the inception of Terrier, the ability to quickly produce compressed index structures representing collections of documents has been critical. However, over the last decade, the scale of the document collections of interest, the specifications of commodity hardware used for indexing and the search tasks that are being investigated have dramatically changed. For instance, Figure 1 illustrates how the size of IR test collections has grown over a 17 year period. These changes have and continue to introduce new indexing challenges that have driven the continual enhancement of Terrier's indexing processes.

The basis for much of the indexing functionality within Terrier stems from the development of single-pass indexing [9], which was released as part of Terrier 2.0. The idea behind single-pass indexing is that a central index structure can be built a document-at-a-time in a 'single-pass' over the collection. Moreover, this index can be created under tight memory constraints by compressing the inverted files and periodically writing partial posting-lists to disk, facilitating indexing on a single machine. Single-pass indexing is a fast and efficient means to index smaller (by today's standards) collections, spanning 100 million documents or less.

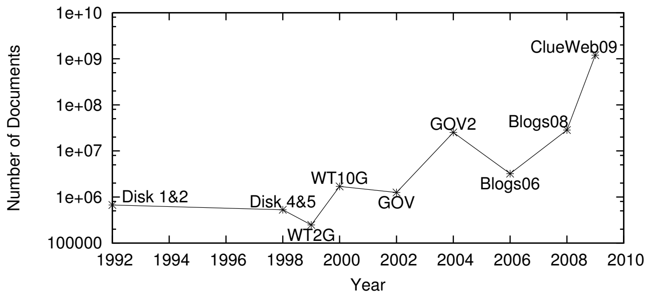


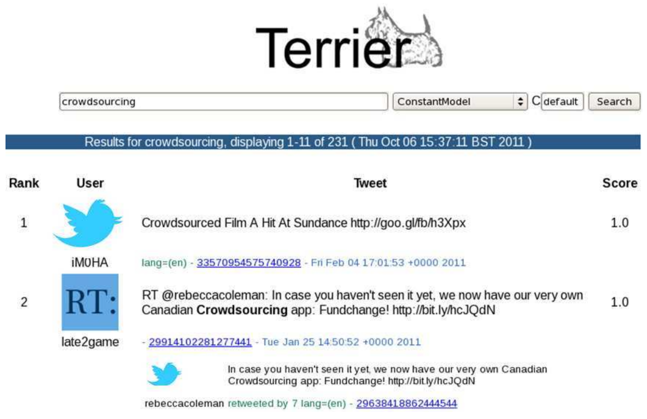
Figure 1: Evolution of the size of TREC corpora.

However, for larger collections such as the TREC ClueWeb-09 corpus that is comprised of 1.2 *billion* documents,[3] single-pass indexing is a slow process requiring weeks of processor time to complete on a single machine. MapReduce is a programming paradigm for the processing of large amounts of data by distributing work tasks over multiple processing machines [6]. The central concept underpinning MapReduce is that many data-intensive tasks are based around performing a *map* operation with a simple function over each 'record' in a large dataset. Terrier 2.2 became the first open source IR platform to implement large-scale parallelised indexing using MapReduce [14] and its Java implementation Hadoop.[4] This has enabled Terrier to scale its indexing process to efficiently tackle collections spanning billions of documents or more, given a suitable cluster of machines.

Furthermore, it is not just the scaling of existing IR processes that is of interest. To facilitate new search tasks, the indexing process needs to be flexible and extensible. Indeed, this is especially true as the focus of IR continues to move from traditional web documents to more specialised domains, such as the search of social media and other user-generated content sources [21]. To this end, the open source and modular nature of Terrier allows for the addition of indexing capability for new collections with specialised structured documents, in addition to custom tokenisation, stopword removal and stemming. For example, we released a custom package for Terrier 3.0 to support the processing of JSON tweets outside the normal release cycle.[5] Moreover, the entire indexing process can be extended to tackle entirely new search tasks. For instance, the ImageTerrier[6] project enhanced Terrier with image retrieval functionality.

Recent developments have focused on enhancing the deployment and demonstration capabilities of Terrier. From an indexing perspective, this involves efficiently storing document metadata for later display to the user. Terrier 3.5 supports automated metadata extraction and snippet gen-

[3]`http://boston.lti.cs.cmu.edu/Data/clueweb09`
[4]`http://hadoop.apache.org`
[5]`http://ir.dcs.gla.ac.uk/wiki/Terrier/Tweets11`
[6]`http://www.imageterrier.org`



crowdsourcing | ConstantModel

Results for crowdsourcing, displaying 1-11 of 231 (Thu Oct 06 15:37:11 BST 2011)

Rank	User	Tweet	Score
1	iM0HA	Crowdsourced Film A Hit At Sundance http://goo.gl/fb/h3Xpx lang=(en) - 33570954575740928 - Fri Feb 04 17:01:53 +0000 2011	1.0
2	late2game	RT @rebeccacoleman: In case you haven't seen it yet, we now have our very own Canadian **Crowdsourcing** app: Fundchange! http://bit.ly/hcJQdN - 29914102281277441 - Tue Jan 25 14:50:52 +0000 2011 In case you haven't seen it yet, we now have our very own Canadian Crowdsourcing app: Fundchange! http://bit.ly/hcJQdN rebeccacoleman retweeted by 7 lang=(en) - 29638418862444544	1.0

Figure 2: Terrier user interface for Twitter search.

eration that can be saved within a specialist meta index structure. This functionality can be paired with the Terrier front-end search interface to facilitate custom search applications, such as Twitter search, as illustrated in Figure 2.

3.2 Retrieval

When retrieving from large corpora in constrained memory situations, it is possible that the system does not have sufficient memory available to decompress the entire posting list for common query terms. From Terrier 3.0, we redesigned the retrieval mechanism such that the posting list for each term is decompressed in a streaming fashion, using an iterator design pattern. Moreover, this had an additional benefit in permitting support for Document-at-a-Time retrieval (DAAT) strategies. Indeed, DAAT retrieval strategies are advantageous in that the number of document score accumulators is markedly reduced compared to Term-at-a-Time (TAAT) scoring, ensuring an efficient retrieval process.

On the internationalisation front, we have been working on extending Terrier to index and retrieve from East Asian corpora. In particular, in Terrier 3.5, we refactored the way documents are processed, such that text parsing and tokenisation are now fully separated operations. As a result of this refactoring, Terrier now supports pluggable tokenisers for different languages, adding to the overall adaptability of the platform. In a first test of the new tokenisation architecture, Terrier delivered out-of-the-box state-of-the-art retrieval performance on news and web corpora for both Chinese and Japanese [20].

4. ROADMAP FOR TERRIER

In the following, we highlight new functionalities developed for Terrier, which we plan to release in future open source versions. Each of these functionalities is key to improving one or more dimensions of EESA within the Terrier platform. In particular, the massive scale and heterogeneity of current corpora and the increasingly complex information needs of search users limits the effectiveness of traditional ranking approaches based on a single feature. Instead, effective retrieval is increasingly moving towards machine-learned ranking functions combining multiple features [11].

Feature-based retrieval (effectiveness and efficiency).

Terrier will support the extraction of query-independent features at indexing time, as well as the efficient extraction of query-dependent features with a single pass over the inverted index at retrieval time. The latter is enabled by an improved matching mechanism that keeps track of the actual postings for documents that might end up among the top retrieved. Another valuable source of evidence, which conveys how a document is described by the rest of the Web, is the anchor text of the incoming hyperlinks to this document. Integrating anchor-text extraction to the indexing pipeline of Terrier will provide a unified solution for leveraging this rich evidence as an additional feature for effective retrieval.

Non-global configuration (adaptability).

An important direction for improving the adaptability of Terrier is to enable multiple instances of its indexing and retrieval pipelines to run concurrently. A crucial development in this direction is a configuration system that admits non-global, instance-specific setups. For instance, a typical search scenario that may require multiple instances of a retrieval process is search result diversification. In particular, effective diversification can be attained by ensuring that the produced ranking covers multiple aspects of an ambiguous query, represented as query reformulations [18, 19].

Dynamic pruning (scalability and adaptability).

Dynamic pruning strategies, such as WAND [4], can increase efficiency by omitting the scoring of documents that can be guaranteed not to make the top-k retrieved set—a feature known as safeness. These pruning strategies rely on maintaining a threshold score that documents must overcome in order to be considered in the top-k documents. Each term is associated with an upper bound stating the maximal contribution of the weighting model to any document's relevance score. By comparing upper bounds on the scores of the terms that have not been scored to the threshold, i.e. the current k-th document, the pruning strategy decides on to whether to skip the scoring of documents during retrieval. In addition, in WAND, skipping is also supported by underlying posting list iterators in order to reduce disk IO and further increase efficiency. However, traditionally in the literature, the upper bounds are pre-calculated at indexing time and stored in the index. As a result, the generated index is only suited for efficient retrieval using one particular weighting model, where all the returned search results of queries are ranked using a single weighting model. Instead, in Terrier, we avoid tying up the generated index to a particular weighting model, by deploying safe upper bound approximations for various retrieval models, which can be calculated on-the-fly at query execution time [12]. By doing so, Terrier supports the recent trend of deploying selective retrieval approaches, where the retrieval strategy varies from a query to another [7, 19].

Distributed and real-time search (scalability).

While a MapReduce indexing process can efficiently index the 1.2B documents of the ClueWeb09 corpus, retrieval from a monolithic single index shard is not efficient. Document-partitioned distributed indexing [5] ensures that efficient retrieval can be attained regardless of the index size. Recently, search in real-time scenarios such as live search over tweets have become popular [21]. Real-time search poses different challenges to traditional retrospective retrieval tasks. In particular, documents are not contained within a static corpus, but rather arrive over time in a streaming fashion.

Moreover, both indexing and retrieval operations occur in parallel, hence index structures must always be searchable, thread safe and up-to-date. Furthermore, from a research perspective, there is a need to support the 'replaying' of a stream, facilitating reproducible experimentation. The development of real-time search within Terrier is well advanced and is targeted for merging into the next major open source release. Terrier's real-time infastructure is also being further developed as part of the SMART EC project to enable low latency distributed indexing and retrieval [1].[7]

Crowdsourcing for relevance assessment (effectiveness).

Researchers rely on document relevance assessments for queries to gauge the effectiveness of their systems. Evaluation forums such as TREC and CLEF play a key role by providing relevance assessments for many common tasks. However, these venues cannot cover all of the collections and tasks currently investigated in IR, resulting in the burden of relevance assessment generation falling upon individual researchers. This is an important problem, as relevance assessment generation is often a time-consuming, difficult and potentially costly process. For many IR-related tasks, crowdsourcing has been shown to be a fast and cheap method to generate relevance assessments in a semi-automatic manner [16], by outsourcing to a large group of non-expert workers. CrowdTerrier is a soon to be released open source extension to Terrier that leverages crowdsourcing to provide researchers with an out-of-the-box tool to achieve fast and cheap relevance assessment [15].

Plugin Expansions (adaptability).

The growth of the Terrier platform into exciting new areas such as crowdsourcing entails increased functionality, but also platform complexity. To avoid software bloat, we are moving from a monolithic release structure, to a system of periodic core releases and timely plugin expansions. This will enable Terrier to continue to grow and evolve to tackle future challenges in the dynamic field of IR in line with the interests of the community.

5. CONCLUSIONS

In this paper, we described the philosophy that has guided the development of the Terrier IR open source platform since its first release in 2004. We described some recent developments in the Terrier IR platform, as well as a comprehensive roadmap for its forthcoming releases, intended to ensure that the platform remains extensible and effective, while providing a robust, modern and efficient grounding for building next generation search engine technologies. The last decade has witnessed a dramatic shift in the scale and nature of experiments IR researchers are increasingly being required to conduct to test and evaluate new search technologies. Our vision for Terrier is to continue empowering researchers and practitioners in IR with up-to-date, effective and scalable tools, allowing them to build and evaluate the next generation IR applications. We hope that many will join us in working together towards such an objective.

Acknowledgements

The authors express their gratitude to all members of the Terrier community who have contributed to the platform over the last decade.

[7] `http://www.smartfp7.eu/content/twitter-indexing-demo`

6. REFERENCES

[1] M.-D. Albakour, C. Macdonald, I. Ounis, A. Pnevmatikakis, and J. Soldatos. SMART: An open source framework for searching the physical world. In *Proc. of OSIR*, 2012.

[2] G. Amati. *Probability models for information retrieval based on Divergence From Randomness.* PhD thesis, University of Glasgow, 2003.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. of SIGIR*, pages 372–379, 2006.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of CIKM*, pages 426–434, 2003.

[5] F. Cacheda, V. Plachouras, and I. Ounis. A case study of distributed information retrieval architectures to index one terabyte of text. *IP&M*, 41(5):1141–1161, 2005.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of OSDI*, pages 137–150, 2004.

[7] X. Geng, T.-Y. Liu, T. Qin, A. Arnold, H. Li, and H.-Y. Shum. Query-dependent ranking using k-nearest neighbor. In *Proc. of SIGIR*, pages 115–122, 2008.

[8] B. He and I. Ounis. A query-based pre-retrieval model selection approach to information retrieval. In *Proc. of RIAO*, pages 706–719, 2004.

[9] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, 54(8):713–729, 2003.

[10] D. Hiemstra. *Using Language Models for Information Retrieval.* PhD thesis, University of Twente, 2001.

[11] T.-Y. Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, 2009.

[12] C. Macdonald, I. Ounis, and N. Tonellotto. Upper-bound approximations for dynamic pruning. *ACM TOIS*, 29(4):1–28, 2011.

[13] C. Macdonald, V. Plachouras, B. He, C. Lioma, and I. Ounis. University of Glasgow at WebCLEF 2005: experiments in per-field normalisation and language specific stemming. In *Proc. of CLEF*, pages 898–907, 2006.

[14] R. McCreadie, C. Macdonald, and I. Ounis. MapReduce indexing strategies: Studying scalability and efficiency. *IP&M*, 2011.

[15] R. McCreadie, C. Macdonald, and I. Ounis. Crowdterrier: Automatic crowdsourced relevance assessments with terrier. In *Proc. of SIGIR*, 2012.

[16] R. McCreadie, C. Macdonald, and I. Ounis. Identifying top news using crowdsourcing. *Information Retrieval*, 2012.

[17] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proc. of TREC*, 1994.

[18] R. L. T. Santos, C. Macdonald, and I. Ounis. Exploiting query reformulations for web search result diversification. In *Proc. of WWW*, pages 881–890, 2010.

[19] R. L. T. Santos, C. Macdonald, and I. Ounis. Selectively diversifying Web search results. In *Proc. of CIKM*, pages 1179–1188, 2010.

[20] R. L. T. Santos, C. Macdonald, and I. Ounis. University of Glasgow at the NTCIR-9 intent task. In *Proc. of NTCIR*, 2011.

[21] I. Soboroff, D. McCullough, J. Lin, C. Macdonald, I. Ounis, and R. McCreadie. Evaluating real-time search over tweets. In *Proc. of ICWSM*, 2012.

[22] H. Zaragoza, N. Craswell, M. J. Taylor, S. Saria, and S. E. Robertson. Microsoft Cambridge at TREC 13: Web and Hard tracks. In *Proc. of TREC*, 2004.

[23] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM TOIS*, 22(2):179–214, 2004.

Yet another comparison of Lucene and Indri performance

Howard Turtle
Center for Natural Language Processing
Syracuse University
Syracuse, NY 13078
turtle@syr.edu

Yatish Hegde
Center for Natural Language Processing
Syracuse University
Syracuse, NY 13078
yhegde@syr.edu

Steven A. Rowe
Center for Natural Language Processing
Syracuse University
Syracuse, NY 13078
sarowe@syr.edu

ABSTRACT

We present results that compare the performance of Lucene and Indri at two points in time (2009 and 2012) using data from TREC 6 through 8. We compare indexing throughput, index size, query evaluation throughput, and retrieval effectiveness. We also examine the degree to which the results produced by the two systems overlap with an eye toward estimating the performance increase that might be expected by combining the results of the two systems.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Search Process; H.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms

Open source search engines, information retrieval, performance evaluation, fusion of search results

1. INTRODUCTION

We have used a number of open source and proprietary search engines to support research at the Center for Natural Language Processing often combining the results from multiple engines to good effect [2]. The two engines we most often use are Lucene/Solr (http://lucene.apache.org/solr/) and Indri [6].

Lucene/Solr is attractive because it is a relatively full feature package that makes it easy to field Web-based applications. Indri is attractive because it offers better search results and because it offers a highly expressive query language that allows very fine grained control of a search. The engines are also natural choices because we are familiar with them. One author (Rowe) is a Lucene contributor and chair of the Lucene Project Management Committee. A second author (Turtle) has contributed to the development of Indri and Lemur.

In order to test the assertion that Indri produced better rankings, to assess the likelihood that combining Lucene and Indri results would improve overall performance, and to improve our understanding of the relative performance of the two engines we ran a series of experiments in 2009 to compare performance on TREC data. Both engines have evolved since 2009 so we reran the tests this year to evaluate any change. We present the results of both sets of experiments here.

SIGIR 2012 Workshop on Open Source Information Retrieval, August 16, 2012, Portland, Oregon, USA
ACM X-XXXXX-XX-X/XX/XX.

2. PRIOR WORK

Lin [3] compared the performance of Lucene and Indri as part of a study of the impact of retrieval quality on the performance of question answering systems and concluded that there was no significant difference between the ranking quality of the two systems. Lin used relatively long queries, which may account for the performance similarity as Indri performance is known to degrade with query length. This study also uses early versions of Indri and Lucene.

Middleton and Baeza-Yates [4] conducted an early survey of the features of 17 search engines and conducted extensive performance tests with 12 of those engines, including Lucene and Indri. Their tests used TREC data (Disk 4, WT10g) and reported indexing time, index size, query evaluation time (one and two word queries), and retrieval effectiveness although not all engines participated in all of the tests. The Middleton and Baeza-Yates study used early versions of Lucene (1.9.1) and Indri (2.4); both engines have changed significantly since their study.

Perea-Ortega et al [5] compare the ranking performance of three retrieval systems (Lucene, Lemur, and Terrier) when used in a Geographical Information Retrieval (GIR) system. They used the GeoCLEF 2007 data and ran both mono- and bilingual queries. They conclude that Lemur works best for monolingual queries and that Terrier works better for bilingual queries.

Armstrong et al [1] compared the retrieval effectiveness of five search engines, including Indri and Lucene (version 2.4), using TREC data from 1994 to 2005. Queries were based on title plus description fields, similar to the long query experiments described in Section 3. They found a somewhat smaller difference between the two systems than reported here – for TREC 6 and TREC 8 data they report that Lucene's Mean Average Precision (MAP) scores are 3% to 4.5% lower than Indri whereas our experiments show MAP scores to be 5.6% lower. Differences in the Lucene version used and details of the experimental setup (e.g., stopwords, stemmer) likely account for the difference.

3. EXPERIMENTS

Two sets of experiments were run to compare Indri with Lucene/Solr performance at two points in time. The first set, originally run in October of 2009 but repeated on more modern hardware, compares the versions of Indri and Lucene that were current at the time. The second set compares the

	TREC Disk 4	TREC Disk 5	Total
Number of documents	293,710	262,367	556,077
Collection size (Mb)	1,194	945	1,344
Number of queries	150	150	

Table 1: Collection statistics

versions of Indri and Lucene that were current in June of 2012. Out-of-the-box settings were used for both systems with no tuning or special query formulation.

While our focus is on the performance of the Indri and Lucene search engines, the experiments are run using their respective wrappers, Lemur and Solr. In 2009, the current version of Lemur was 4.10 which used Indri version 2.10. The current version of Solr was 1.4 which used Lucene version 2.9.1. By June 2012, the Lemur software had been repackaged so that the wrapper software and search engine were combined in a single distribution, Indri 5.3. In 2012, Solr and Lucene remained separate packages but the version numbers had been aligned so the current version of Solr was 3.6 which used Lucene version 3.6.

We collected performance information on indexing speed, index size, query evaluation times for two query sets, ranking performance for those queries (using trec_eval), and overlap between the results produced by the two systems. All experiments were run on an Intel(R) Xeon(R) CPU E5335 @ 2.00GHz (four cores) running Debian Squeeze v6.0.4 and Java 1.6 (Oracle). While the test system was a four core system, all tests were single threaded. The test system is equipped with 16Gb of memory but both Indri and Solr were only given 1Gb.

3.1 Data

We used a single data set consisting of TREC disks 4 and 5 for both sets of experiments. The Porter stemmer and the default Solr stop word list (35 words) was used for both the Indri and Lucene collections. Collection statistics are shown in Table 1.

3.2 Queries

Two sets of queries were generated from TREC topics 301-450 (TREC 6 through 8). The first query set (short queries) consists of the text from the title element of the TREC topics. The short queries average 2.6 words per query. The second set (long queries) consists of the text from both the title and description elements with an average length of 18.7 words per query. The queries were completely unstructured and made no use of proximity or other special query language features.

4. RESULTS

4.1 Indexing

The results of the indexing experiment are shown in Table 2. The index sizes remained the same for both systems between 2009 and 2012. Both systems produce indexes that are roughly twice the size of the source file. Indri produced a more compact index; the Solr index is roughly 17% larger than the Indri index. The indexing time results are quite different. Indri 5.3 indexing time increased from Indri 4.1 by about 10% whereas Solr indexing time decreased between the two versions by about 24%. Indri indexing is faster than Solr for both experiments but the difference is greatly reduced, Solr indexing was slower by a factor of 1.7 in 2009 but only by a factor of 1.2 in 2012.

4.2 Query evaluation

Query evaluation times are shown in Table 3. There is little difference in the query throughput of the two engines for short queries and no change in performance between 2009 and 2012 for short queries. For long queries there are significant differences. Indri is significantly slower than Lucene for long queries. The increase in time for evaluating long vs short queries is between 20 and 25 for Indri (long queries are roughly 7 times longer than short queries) and only a factor of 2 for Lucene. Indri query evaluation for long queries slowed between 2009 and 2012.

Note that for these tests the entire index for each of the systems was cached by the operating system as the test machine was equipped with 16Gb of memory and the combined size of the two indexes is only 5.2 Gb. Each system was run once to prime the OS cache then run 3 to 5 times to gather timings. Comparing performance when the the collections must be read from disk is for future work.

Indri is much more processor intensive than Lucene. During the experiments Indri used essentially 100% of a CPU core whereas Lucene used roughly 50%. Indri generally used less memory – for the 2012 versions running long queries Indri used up to 45Mb of memory whereas Lucene used 150 to 300Mb.

4.3 Retrieval effectiveness

Retrieval effectiveness results are shown in Tables 4 (2009) and 5 (2012). For short queries, Indri produces a significantly better ranking. Performance as measured by MAP is 44% less for Lucene. Using precision at fixed ranks of 10 and 20, Lucene performance is roughly 30% lower, using bpref Lucene is 26% lower. For long queries, the differences are smaller but Indri still produces noticeably better rankings – Solr is 16% lower using MAP and 14% lower using bpref.

The change in retrieval effectiveness between 2009 and 2012 is shown in Tables 6 (Indri) and 7 (Solr). For both systems the change is small. Indri showed no change for short queries and mixed results for long queries (slight increase in P10 and P20). Solr showed small improvements for short queries and mixed results for long queries.

4.4 Overlap

	Short queries		Long queries	
	Indri 5.3	Solr 3.6	Indri 5.3	Solr 3.6
P(in OL)	0.2076		0.4653	
P(+|in OL)	0.4824		0.4688	
P(-|in OL)	0.4363		0.4546	
P(?|in OL)	0.0813		0.0767	
P(+)	0.3721	0.2587	0.4167	0.3813
P(-)	0.5112	0.5967	0.5127	0.5547
P(?)	0.1167	0.1447	0.0707	0.0640
P(+|not OL)	0.3577	0.2057	0.3596	0.3040
P(-|not OL)	0.5253	0.6450	0.5697	0.6429
P(?|not OL)	0.1170	0.1493	0.0707	0.0531

Table 8: Overlap in top 10 ranks

	2009		2012	
	Lemur 4.1	Solr 1.4	Indri 5.3	Solr 3.6
Index size (gigabytes)	2.4 (1.8x)	2.8 (2.1x)	2.4	2.8
Indexing time (sec)	863	1,461	942	1,113
Throughput (Mb/sec)	1.6	0.9	1.4	1.2

Table 2: Indexing results

	2009		2012	
	Lemur 4.1	Solr 1.4	Indri 5.3	Solr 3.6
Short queries (sec)	10	13	10	13
(sec/query)	0.07	0.09	0.07	0.09
Long queries (sec)	200	25	251	25
(sec/query)	1.33	0.16	1.67	0.16

Table 3: Query evaluation times

	Short queries			Long queries		
	Lemur 4.1	Solr 1.4	Change	Lemuri 4.1	Solr 1.4	Change
MAP	0.1951	0.1092	−44.1	0.2235	0.1840	−17.7
Precision at 10	0.3713	0.2573	−30.7	0.4053	0.3827	−5.6
Precision at 20	0.3247	0.2173	−33.1	0.3500	0.3220	−8.0
bpref	0.2219	0.1645	−25.9	0.2449	0.2081	−15.0

Table 4: Indr vs. Solr retrieval effectiveness (2009)

	Short queries			Long queries		
	Indri 5.3	Solr 3.6	Change	Indri 5.3	Solr 3.6	Change
MAP	0.1948	0.1098	−43.6	0.2224	0.1856	−16.1
Precision at 10	0.3707	0.2607	−29.7	0.4167	0.3813	−8.5
Precision at 20	0.3243	0.2207	−31.9	0.3590	0.3183	−11.3
bpref	0.2219	0.1645	−25.9	0.2433	0.2087	−14.2

Table 5: Indr vs. Solr retrieval effectiveness (2012)

	Short queries			Long queries		
	Lemur 4.1	Indri 5.3	Change	Lemur 4.1	Indri 5.3	Change
MAP	0.1951	0.1948	−0.2	0.2235	0.2224	−0.5
Precision at 10	0.3713	0.3707	−0.2	0.4053	0.4167	+2.8
Precision at 20	0.3247	0.3243	−0.1	0.3500	0.3590	+2.6
bpref	0.2219	0.2219	0.0	0.2449	0.2433	−0.7

Table 6: Change in Indri retrieval effectiveness over time

	Short queries			Long queries		
	Solr 1.4	Solr 3.6	Change	Solr 1.4	Solr 3.6	Change
MAP	0.1092	0.1098	+0.5	0.1840	0.1856	+0.9
Precision at 10	0.2573	0.2607	+1.3	0.3827	0.3813	−0.4
Precision at 20	0.2173	0.2207	+1.6	0.3220	0.3183	−1.1
bpref	0.1645	0.1645	0.0	0.2081	0.2087	+0.3

Table 7: Change in Solr retrieval effectiveness over time

The overlap between the results produced by both systems is shown in Tables 8 and 9 (+ means document judged relevant, − means document judged not relevant, ? means document not judged, OL means in overlap). These numbers are important for two reasons. First, effectiveness results can be biased in favor of a system that has been used extensively in the TREC experiments if many of the documents retrieved by the other system have not been judged

	Short queries		Long queries	
	Indri 5.3	Solr 3.6	Indri 5.3	Solr 3.6
P(in OL)	0.2356		0.4973	
P(+|in OL)	0.4042		0.4026	
P(-|in OL)	0.5211		0.5325	
P(?|in OL)	0.0747		0.0649	
P(+)	0.3274	0.2207	0.3590	0.3183
P(-)	0.5562	0.5880	0.5763	0.6133
P(?)	0.1163	0.1913	0.0647	0.0683
P(+|not OL)	0.3026	0.1515	0.3093	0.2199
P(-|not OL)	0.5750	0.6334	0.6283	0.7119
P(?|not OL)	0.1224	0.2151	0.0624	0.0681

Table 9: Overlap in top 20 ranks

and will therefore be treated as not relevant. The results in Table 8 suggest that this is not a factor in these experiments – the number of unjudged documents is small for both systems with between 85% and 90% of all documents judged for short queries and between 90% and 95% for long queries.

Second, they provide an indication of how much improvement might be achieved by combining the results of the two systems. If the overlap is large then the combined result can have little increase in recall so the primary source of improvement is the reordering of the documents based on combined score. If the overlap is smaller then the probability that a randomly selected document from the overlap is relevant is an indication of how well a simple voting strategy might work. For example, in Table 8 roughly 20% of the documents retrieved by the two systems with short queries were the same. The probability that a document retrieved by both systems is relevant is 0.4824 which is significantly higher than the probability of relevance achieved by either system individually (0.3721 for Indri and 0.2587 for Solr).

5. CONCLUSIONS

The results presented here allow direct comparison of the two search engines. It also allows comparison of the changes in the two engines between 2009 and 2012.

Index size for the two engines did not change between 2009 and 2012. Indri produces a somewhat smaller index (1.8 times as large as the source collection) than Lucene (2.1 times). In terms of indexing throughput, Indri declined between between 2009 and 2012 (from 1.6Mb/sec to 1.4Mb/sec) whereas Lucene performance improved (from 0.9Mb/sec to 1.2Mb/sec). In 2012, Indri still enjoyed a slight advantage over Lucene (1.4Mb/sec vs 1.2Mb/sec).

In terms of query throughput there is little difference between the two engines for short queries but Indri is significantly slower than Lucene for long queries.

In terms of retrieval effectiveness, Indri results are significantly better than Lucene results especially for short queries. Using precision at rank 20, Lucene rankings are roughly 30% worse for short queries and roughly 10% worse for long queries. Retrieval effectiveness did not change significantly for either engine between 2009 and 2012, at least for the simple queries used in these experiments.

The overlap results show that the documents retrieved by the two engines are significantly different, especially for short queries. Using the top ten documents retrieved, for short queries roughly 80% of all documents retrieved appear in only one of the two rankings. For long queries, about half of the documents retrieved appear in only one ranking. The overlap results also show that even simple strategies for combining results can yield significant improvements in retrieval effectiveness.

6. REFERENCES

[1] T. G. Armstrong, A. Moffat, W. Webber, and J. Zobel. Has adhoc retrieval improved since 1994? In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, pages 692–693, New York, NY, USA, 2009. ACM.

[2] J. W. Keeling, E. E. Allen, S. A. Rowe, A. M. Turner, J. A. Merrill, E. D. Liddy, and H. R. Turtle. Development and evaluation of a prototype search engine to meet public health needs. In *Proceedings of the American Medical Informatics Association*, pages 693–700, 2011.

[3] J. Lin. The role of information retrieval in answering complex questions. In *Proceedings of the COLING/ACL on Main conference poster sessions*, COLING-ACL '06, pages 523–530, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[4] C. Middleton and R. Baeza-Yates. A comparison of open source search engines. Technical report, Universitat Pompeu Fabra (Barcelona, Spain), Oct. 2007. Available at: http://wrg.upf.edu/WRG/dctos/Middleton-Baeza.pdf.

[5] J. Perea-Ortega, M. Garcia-Cumbreras, M. Garcia-Vega, and L. Urena-Lopez. Comparing several textual information retrieval systems for the geographical information retrieval task. In E. Kapetanios, V. Sugumaran, and M. Spiliopoulou, editors, *Natural Language and Information Systems*, volume 5039 of *Lecture Notes in Computer Science*, pages 142–147. Springer Berlin / Heidelberg, 2008.

[6] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language-model based search engine for complex queries. Technical Report IR-407, CIIR, Department of Computer Science, University of Massachusetts Amherst, 2005.

Phonetic Matching in Japanese

Michiko Yasukawa* J. Shane Culpepper† Falk Scholer†

*Gunma University, Gunma, Japan
michi@cs.gunma-u.ac.jp

†RMIT University, Melbourne, Australia
{shane.culpepper, falk.scholer}
@rmit.edu.au

ABSTRACT

This paper introduces a set of Japanese phonetic matching functions for the open source relational database PostgreSQL. Phonetic matching allows a search system to locate approximate strings according to the sound of a term. This sort of approximate string matching is often referred to as fuzzy string matching in the open source community. This approach to string matching has been well studied in English and other European languages, and open source packages for these languages are readily available. To our knowledge, there is no such module for the Japanese language. In this paper, we present a set of string matching functions based on the phonetic similarity for modern Japanese. We have prototyped the proposed functions as an open source tool in PostgreSQL, and evaluated these functions using the test collection from the NTCIR-9 INTENT task. We report our findings based on the evaluation results.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation, retrieval models, search process*; I.7.1 [**Document and Text Processing**]: Document and Text Editing—*languages, spelling*; I.7.3 [**Document and Text Processing**]: Text Processing—*index generation*

General Terms

Open Source RDBMS, Approximate String Matching, Fuzzy String Matching, Phonetic Matching, Japanese Information Retrieval

1. INTRODUCTION

One interesting but cumbersome problem in IR and NLP research is mismatches between the spelling of words. A simple question for this problem is: What if two words have the same meaning but different spellings? This is the issue explored in this paper. A related but more difficult problem are homographs, or words with the same spelling. The latter requires word sense disambiguation (WSD)[7], which is the task of identifying the meaning of words in a context. In this paper, we focus only on words with the same meaning and different spellings. These words are categorized into the following two types.

- *Synonyms* – words with the same (or, nearly the same) meaning, different spellings and different pronunciation.
- *Spelling variation* – words with the same meaning, different spellings and the same (or, nearly the same) pronunciation.

Synonyms are words that are similar in a semantic sense. Approaches such as Latent Semantic Indexing (LSI) [2] and word clustering[9] can be used to alleviate this problem. Spelling variants are slightly more difficult to classify, and present a difficult problem in ranked document retrieval systems depending on keyword queries.

Recent efforts to improve the effectiveness of IR systems have included web search result diversification [10]. In order to increase search effectiveness in this task, an innovative solution to spelling variants is needed. The goal of this paper is take a first step towards providing an open source tool to help with this problem in the Japanese language.

The rest of this paper is organized as follows: Section 2 presents string matching methods for English and Japanese; Section 3 describes our proposed phonetic matching approach for Japanese; Section 4 reports our findings based on a preliminary experimental study; and Section 5 presents conclusions and future work.

2. RELATED WORK

Phonetic matching is a type of approximate string matching. As with other approximate pattern matching methods, edit distance[6] and character-based n-gram search[8] are commonly used. The seminal approach to phonetic matching is the Soundex Indexing System[5]. It can accurately assign the same codeword to two different surnames that sound the same, but are spelled differently, like SMITH and SMYTH. The basic coding rules of Soundex are shown in Figure 1. Both SMITH and SMYTH are encoded as S530 by Soundex. In English and other languages, revised versions and alternatives of Soundex have been proposed. A set of these string matching functions, generally referred to as "fuzzy string matching", are deployed in an open source programming language. These functions are assembled in an open source relational database, PostgreSQL[1] as well and are applied to the objects in the database or combined with other relational operations.

[1]http://www.postgresql.org/

SIGIR 2012 Workshop on Open Source Information Retrieval.
August 16, 2012, Portland, Oregon, USA.

Step-1 Retain the first symbol and drop all vowels.
Step-2 Replace consonants with the following **code**.

b, f, p, v	→	**1**	l	→	**4**
c, g, j, k, q, s, x, z	→	**2**	m, n	→	**5**
d, t	→	**3**	r	→	**6**

Step-3 Remove the duplication of code numbers.
Step-4 Continue until you get three code numbers. If you run out symbols, fill in **0**'s until there are three code numbers.

Figure 1: The Soundex Indexing System.

For English, phonetic matching for IR systems has been studied extensively[1, 3, 11]. If Japanese documents are transliterated into a Latin alphabet, English methods can be applied. However, transliteration contains its inherent problems. Some input characters are lost due to the lack of correspondence between Japanese and Latin alphabets. How to reduce the transliteration errors is an interesting related problem, and will be considered in our future research. The issue of effective transliteration was explored by Karimi et al.[4]. Our research interest in this paper is to develop native matching functions for Japanese search engines without transliteration.

3. OUR METHODOLOGY

In this section, we present our approach to symbol grouping and phonetic matching in Japanese.

3.1 Japanese writing system and syllabary

In the Japanese language, the number of phonetic sounds is relatively small, and simply expressed in the 5×10 grid in Table 1. It shows the Japanese syllabary, "五十音(Gojūon)" meaning "Fifty Sounds" in English. In the table, both Hiragana symbols (the rounded syllabic symbols) and Katakana symbols (the angular syllabic symbols) are displayed. Vowel symbols, such as ア (a) and イ (i), do not have corresponding consonants, and this is represented as ϕ in the leftmost column. In the table, the gray cells are vacant because the symbols become lost over time. In modern Japanese, there is no symbol for Y+I (yi), Y+E (ye), W+U (wu), and the pronunciation for them are the same as ϕ+I (i), ϕ+E (e), ϕ+U (u), respectively. The symbols for W+I (wi) and W+E (we) are outdated, but can still be used in modern Japanese. They are normally replaced with symbols for ϕ+I (i) and ϕ+E (e) because of the similarity of the pronunciation. In addition to the symbols in the table, the Japanese writing system takes an additional symbol, ン (n, a syllabic nasal), symbols with a voiced/semi-voiced sound mark, such as ガ (ga) or パ (pa), lower-case symbols, such as ッ (tsu, a double consonant), or ャ (ya, a contracted sound), and a diacritical mark for a prolonged sound, such as ー (a macron). For web queries and documents, classical spellings, such as the usage of obsolete symbols, ヱ (we) or an uncommon usage of lower-case symbols, e.g, ヶ (ke) substituting for ケ (ke), may appear in web queries and documents for stylistic reasons, or simple mistakes. Japanese phonetic matching must account for such anomalies.

Table 1: The Japanese Syllabary (Fifty Sounds).

	Hiragana Symbol					Katakana Symbol					
	A	I	U	E	O	A	I	U	E	O	
ϕ	あ	い	う	え	お	ア	イ	ウ	エ	オ	1
	a	i	u	e	o	a	i	u	e	o	
K	か	き	く	け	こ	カ	キ	ク	ケ	コ	2
	ka	ki	ku	ke	ko	ka	ki	ku	ke	ko	
S	さ	し	す	せ	そ	サ	シ	ス	セ	ソ	3
	sa	si	su	se	so	sa	si	su	se	so	
T	た	ち	つ	て	と	タ	チ	ツ	テ	ト	4
	ta	ti	tu	te	to	ta	ti	tu	te	to	
N	な	に	ぬ	ね	の	ナ	ニ	ヌ	ネ	ノ	5
	na	ni	nu	ne	no	na	ni	nu	ne	no	
H	は	ひ	ふ	へ	ほ	ハ	ヒ	フ	ヘ	ホ	6
	ha	hi	hu	he	ho	ha	hi	hu	he	ho	
M	ま	み	む	め	も	マ	ミ	ム	メ	モ	7
	ma	mi	mu	me	mo	ma	mi	mu	me	mo	
Y	や		ゆ		よ	ヤ		ユ		ヨ	8
	ya		yu		yo	ya		yu		yo	
R	ら	り	る	れ	ろ	ラ	リ	ル	レ	ロ	9
	ra	ri	ru	re	ro	ra	ri	ru	re	ro	
W	わ	ゐ		ゑ	を	ワ	ヰ		ヱ	ヲ	10
	wa	wi		we	wo	wa	wi		we	wo	
	1	2	3	4	5	1	2	3	4	5	

3.2 Symbol groups in Japanese

Table 2 shows symbol groups for our phonetic matching functions. The symbol groups are assembled based on the similarity of Japanese speech sounds. Each symbol group is given a unique identifier (ID). In the table, text in parentheses expresses a commentary on each group and the corresponding consonant, e.g., K for カ (ka) and S for サ (sa). Vowel symbols, such as ア (a) and イ (i), do not have corresponding consonants, and hence ϕ is in the parentheses.

Table 2 composed of 3 distinct parts: "Fifty Sounds," "Voiced Sounds," and "Additional Sounds." As a whole, the table covers all speech sounds in modern Japanese that are writable with Katakana symbols in UTF-8 character encoding. Different from English phonetic matching that uses the Latin alphabet or Arabic numeral for input/output strings, our matching functions take Katakana symbols for input strings and use Hiragana symbols for output strings. In UTF-8 character encoding, Katakana symbols are from ァ (E382A1) to ヶ (E383B6), and the number of Katakana symbols is 86. Hiragana symbols are from ぁ (E38181) to ん (E38293), and the number of Hiragana symbols is 83. The three symbols, ヴ (E383B4), ヵ (E383B5), ヶ (E383B6) are special symbols. They are defined in the Katakana part only, and there is no corresponding Hiragana symbol for these symbols.

In Table 2, Katakana symbols for "Fifty Sounds" (F-01 to F-11) are mostly the same as those in Table 1 with the exception of Katakana symbols for *wi*, *we* and *wo* because the Katakana symbols, ヰ (wi), ヱ (we), and ヲ (wo) have the same pronunciation as イ (i), エ (e), and オ (o), respectively, in modern Japanese. Hence, F-02 are incorporated into the same group as F-01 in Table 2. Similarly, V-03 are separated from V-04, and incorporated into V-02 because the Katakana symbols, ヂ (di) and ヅ (du) in modern

Table 2: **The Symbol Groups for Japanese Phonetic Matching.**

ID	Fifty Sounds [in]	Code [out]	ID	Voiced Sounds [in]	Code [out]	ID	Additional Sounds [in]	Code [out]
F-01	アイウエオ (ϕ) →	`E38182` あ				A-01	ァィゥェォ (lower-case, ϕ) →	`E38182` あ
F-02	ヰヱヲ (obs., ϕ) →	`E38182` あ				A-02	ー (macron, ϕ) →	`E38182` あ
F-03	カキクケコ (K) →	`E3818B` か	V-01	ガギグゲゴ (G) →	`E3818C` が	A-03	ヵヶ (lower-case, K) →	`E3818B` か
F-04	サシスセソ (S) →	`E38195` さ	V-02	ザジズゼゾ (Z) →	`E38196` ざ			
			V-03	ヂヅ (obs., Z) →	`E38196` ざ			
F-05	タチツテト (T) →	`E3819F` た	V-04	ダデド (D) →	`E381A0` だ	A-04	ッ (lower-case, T) →	`E381A3` っ
F-06	ナニヌネノ (N) →	`E381AA` な				A-05	ン (syllabic nasal, N) →	`E38293` ん
F-07	ハヒフヘホ (H) →	`E381AF` は	V-05	バビブベボ (B) →	`E381B0` ば			
			V-06	ヴ (V) →	`E381B0` ば			
			V-07	パピプペポ (P) →	`E381B1` ぱ			
F-08	マミムメモ (M) →	`E381BE` ま						
F-09	ヤユヨ (Y) →	`E38284` や				A-06	ャュョ (lower-case, Y) →	`E38283` ゃ
F-10	ラリルレロ (R) →	`E38289` ら						
F-11	ワ (W) →	`E3828F` わ				A-07	ヮ (lower-case, W) →	`E3828F` わ

Japanese have the same pronunciation as ジ (zi) and ズ (zu), respectively.

Each Katakana symbol for "Voiced Sounds" (V-01 to V-07) is a symbol with a voiced/semi-voiced sound mark. In the table, voiceless and voiced (e.g., K and G), voiced and semi-voiced (e.g., B and P), and Japanese voiced and foreign voiced (e.g., B and V) are all distinguished and separated into different symbol groups. "Additional Sounds" (A-01 to A-07) cover the rest of Katakana symbols and the diacritical mark, ー (a macron).

3.3 Japanese Phonetic Matching

In the same way as the Soundex Coding System in English, the matching function in Japanese also encodes symbols according to symbol groups. Essential steps in the matching function are described as follows:

Step-1 Encode all strings in text DB in advance.

Step-2 Encode a query string on arrival.

Step-3 Output a matching set of encoded symbols.

The greatest challenge in designing phonetic matching functions is deciding how to group similar phonetic symbols. Our approach accomplishes this challenge empirically rather than theoretically by using the actual speech sound in modern Japanese. Our first phonetic matching function (Japanese phonetic matching; jppm) is as follows:

jppm1 Retain the first symbol, and encodes the rest as encoded symbols according to Table 2. The encoded symbols are a sequence of Hiragana symbols in UTF-8 character encoding. While it categorizes Japanese speech sounds in a rigorous manner, the output codewords tend to be too verbose, and consequently cause mismatches with similar strings.

In order to reduce the number of codewords used, we derive three revised versions from **jppm1**. They are altered from the initial version as follows:

jppm2 In order to simplify output code symbols, drop all symbols if they are vowels (F-01 and F-02) or an "Additional Sound" (A-01 to A-07) in Table 2. The first symbol is always retained in this function.

jppm3 In order to simplify encoded symbols, merge groups in the same line (the same row) in Table 2. To be more specific, incorporate V-01 into F-03, incorporate V-02 and V-03 into F-04, incorporate V-04 and A-04 into F-05, incorporate V-05, V-06 and V-07 into F-07, incorporate A-06 into F-09. The first symbol is always retained in this function.

jppm4 In order to simplify output codewords, drop A-01, A-02, A-04 and A-06 in Table 2. The first symbol is still retained as is.

3.4 Implementation

Our aim is to provide an open source component of Japanese phonetic matching. We prototyped the phonetic matching functions in Japanese as an extended version of the `fuzzystrmatch` module in the open source relational database, PostgreSQL. We call the new module `jpfuzzystrmatch` and provide its source code under an open source license[2]. To help understanding, an example of the each of the 4 functions (jppm[1,2,3,4]) and an example of English phonetic encoding systems with a transliteration system are attached with the source code. The module contains user-defined C-Language functions, and is compiled into dynamically loadable objects (shared libraries). It is distinguished from PostgreSQL internal functions, and can be loaded by the server on demand.

4. EXPERIMENTS

Evaluation of phonetic matching is an important problem to consider. In contrast to ad hoc evaluation in IR systems, there are no standard test collections for phonetic matching functions. In our experiment, we adapted a standard test collection for IR systems, the NTCIR-9 Japanese Intent task, for evaluating phonetic matching functions in Japanese. To build a text database to query, 84 million index terms were extracted from 67 million Japanese documents in the `ClueWeb09-JA` collection. For document processing, we employed MeCab[3] as a morphological an-

[2]http://www.daisy.cs.gunma-u.ac.jp/jpfsm/
[3]http://mecab.sourceforge.net/

Table 3: Experimental Results (Average).

	CodeLen	#Results	EditDist	P_{score}
jppm1	8.2	6.0	1.88	3.06
jppm2	5.3	131.2	3.73	24.00
jppm3	8.2	9.2	2.06	4.29
jppm4	6.3	25.5	2.74	7.19

alyzer in Japanese, and Indri[4] as an indexing module to extract index terms from documents. In order to analyze how well the new functions work, sufficiently long Katakana words from topic words in the test collection were used as query strings. Specifically, we used all Katakana words which consists of 7 or more Katakana symbols; there were 10 such words in the test set.

Since the judgments in the test collection were not developed to carry out phonetic matching, how to evaluate the developed functions needs to be considered. Generally speaking, it is difficult for human assessors to judge if two strings are phonetically similar. For example, Zobel and Dart[11] report a high level of inconsistency among assessors in such a judgment proces. We therefore adopted an automatic evaluation approach.

In order to evaluate phonetic matching in an automatic manner, we make the following assumptions.

- If the discrepancy between two spellings is small, phonetic similarity is naturally large.
- If many matching strings are returned, the shortened sequence of symbols is sufficiently generic.

Based on the above assumptions, we define the following score, P_{score} to measure the performance of phonetic matching functions.

$$P_{score} = \frac{\#Results}{1 + avg(EditDist)}$$

Here, *#Results* is the number of strings returned by a phonetic matching function, and $avg(EditDist)$ is the average edit distance between the query string and each returned string. Table 3 shows the experimental results. The average values are across all 10 test queries. In the table, **jppm2** has the highest P_{score}. However, manual inspection of the results showed that this function returned many strings that are phonetically similar, but are not spelling variants of the query strings. For example, **jppm2** obtained "matui ryousuke" (a male's name in Japanese) and "mattress queen" (a product name) for the query "matoriyosika" ("Matryoshka doll"). Since **jppm[1,3]** have different grouping features from **jppm2**, some obtained strings are different among these functions. The results of **jppm4** are a subset of those of **jppm2** because the grouping features are common. Putting together obtained strings, phonetic matching can be a viable solution to find potential spelling variants. However, the obtained strings still need to be filtered using further semantic analysis. In future work, we intend to evaluate these functions in the context of a full IR task.

[4]http://sourceforge.net/apps/trac/lemur/

5. CONCLUSION

In this paper, we have presented a set of new Japanese phonetic matching functions. We do not attempt to address the bigger issue of "word sense disambiguation" (WSD) and synonyms, but rather present a simple approach to capturing similar Japanese terms for ad hoc queries. A phonetic matching function takes an input sequence of symbols and converts it into a generic shortened sequence of symbols in order to find as many fuzzy matches as possible. Some of the generic codewords may gather too many matches, including correct ones (the same meaning) and wrong ones (different meaning). However, this approach can be used as a first pass filter to find potentially relevant documents in large Japanese document collections. This subset of documents can then be further refined using relevance feedback or more computationally expensive ranking metrics in subsequent retrieval steps. In future work, we intend to investigate the broader issue of WSD and synonyms using phonetic matching, and explore other applications of these simple matching functions.

6. ACKNOWLEDGMENTS

The first author was supported by JSPS KAKENHI Grant-in-Aid for Young Scientists (B) 21700273. The second author was supported by the Australian Research Council.

7. REFERENCES

[1] Amir, A., Efrat, A., and Srinivasan, S. Advances in phonetic word spotting. In *CIKM '01 Proceedings* (2001), ACM, pp. 580–582.

[2] Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. Indexing by latent semantic analysis. *JASIS 41*, 6 (1990), 391–407.

[3] French, J. C., Powell, L., and Schulman, E. Applications of approximate word matching in information retrieval. In *CIKM '97 Proceedings* (1997), ACM, pp. 9–15.

[4] Karimi, S., Scholer, F., and Turpin, A. Machine transliteration survey. *ACM Comput. Surv. 43*, 3 (2011), 17:1–17:46.

[5] Knuth, D. E. *The Art of Computer Programming: Volume 3.* Addison-Wesley, 1973.

[6] Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv. 33*, 1 (2001), 31–88.

[7] Navigli, R. Word sense disambiguation: A survey. *ACM Comput. Surv. 41*, 2 (2009), 10:1–10:69.

[8] Okazaki, N., and Tsujii, J. Simple and efficient algorithm for approximate dictionary matching. In *Coling Proceedings* (2010), pp. 851–859.

[9] Slonim, N., and Tishby, N. Document clustering using word clusters via the information bottleneck method. In *SIGIR '00 Proceedings* (2000), pp. 208–215.

[10] Song, R., Zhang, M., Sakai, T., Kato, M., Liu, Y., Sugimoto, M., Wang, Q., and Orii, N. Overview of the NTCIR-9 INTENT task. In *Proceedings of NTCIR-9 Workshop Meeting* (2011), pp. 82–105.

[11] Zobel, J., and Dart, P. W. Phonetic string matching: Lessons from information retrieval. In *SIGIR '96 Proceedings* (1996), pp. 166–172.

ISBN 978-0-473-22025-9

9 780473 220259 >