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

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

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.

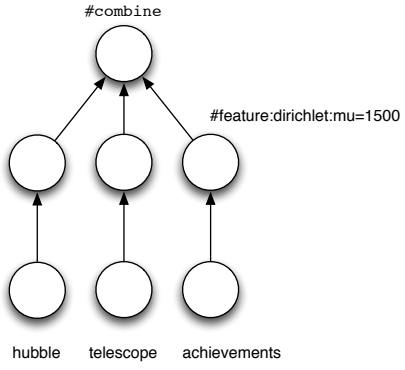

Figure 1: A simple query, represented as a tree. The middle layer of feature nodes in this query tree each convert frequency information about a term into a Dirichlet-smoothed probability.

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`

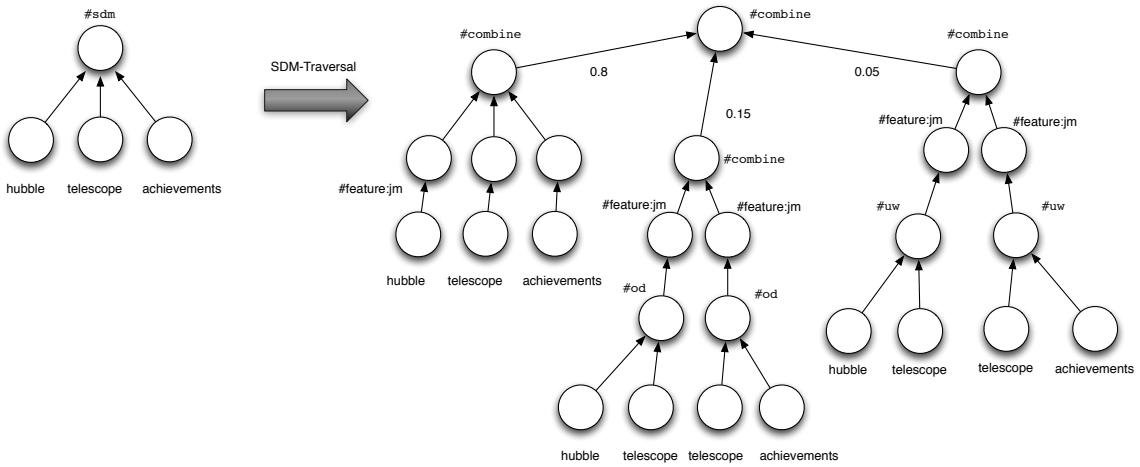

Figure 2: The expansion of the Sequential Dependence Model using a traversal.The layer of feature nodes in this query tree each convert frequency information about a term or a window into a Jelinek-Mercer-smoothed probability.

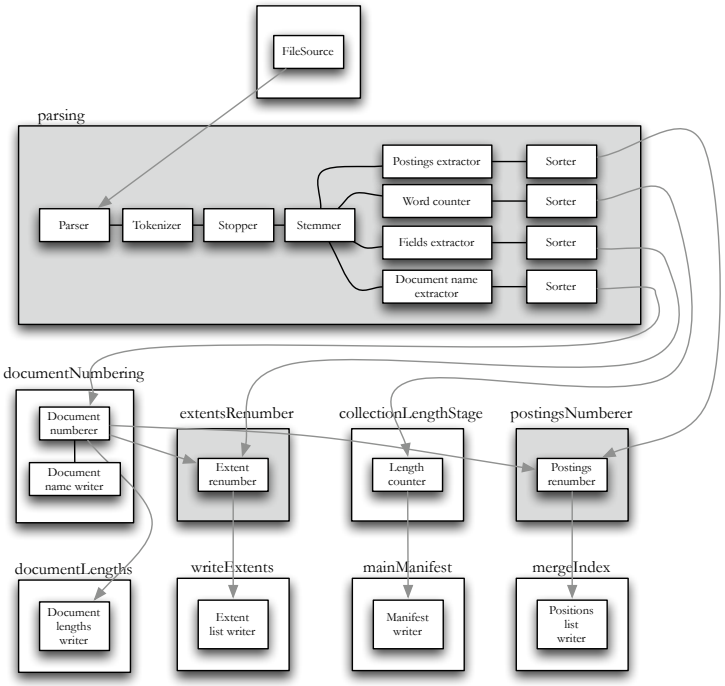

Figure 3: An example of indexing a collection using TupleFlow (generated by Trevor Strohman [12]).

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 obvi-

ous 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.