Apache Lucene 4

Andrzej Bialecki, 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 Codec 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
Incremental updates are supported and stored in index extents mechanisms for data coding (see the section on Codec API below) and the cost of the variety of encoding schemas that affect the size of the index data a per.

5.2

releases.

package, with additional improvements planned for future

output algebras. Synonym processing, Japanese text analysis, spell

and weighted automata.

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.

5.2.1 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).

5.3.1 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.

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).

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 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.

Figure 3 Indexing Process

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

![Figure 3 Indexing Process](image-url)
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 experimen-
tation with recent advances in inverted index compression algo-
rithms. 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 \( f_1 \) and
term \( t_1 \) the cursor can be advanced along this posting list to the
data for document \( d_1 \), 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 scor-
ing 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 benchmarking 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.cluew-
eb09.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 Haydahl, 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
Blok, Peter Carlson, Tal Dayan, Bertrand Delacretaz, Scott
Ganyo, Brian Goetz, Christoph Goller, Eugene Gluzberg,
Wolfgang Hochsek, 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

http://www.apache.org/licenses/LICENSE-2.0

works.html

http://www.lucidimagination.com/devzone/videos-
podcasts/podcasts/interview-walter-underwood-netflix


