

Apache Lucene Technology Overview

by Aaron Bannert
aaron@codemass.com



Copyright © 2007 Codemass, Inc.



Except where otherwise noted, this presentation is licensed under the Creative Commons **Attribution-NonCommercial-NoDerivs 2.5 License**, available here: <http://creativecommons.org/licenses/by-nc-nd/2.5/>

This presentation is a brief overview of Lucene, its capabilities and limitations, along with some standard (and novel) architectures within which to develop and deploy Lucene applications.

What is Apache Lucene?



“Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java.”

- from <http://lucene.apache.org/>



Availability

- Freely Available (no cost)
- Open Source
 - Apache License, version 2.0
<http://www.apache.org/licenses/LICENSE-2.0>



Features

- Ranked Searching
- Flexible Queries
 - Phrases, Wildcards, etc...
- Field-specific Queries
 - eg. title, artist, album
- Sorting

Lucene's license is very compatible with commercial use, and can be used within a proprietary service.

Ranked Searching



1. Phrase Matching
2. Keyword Matching
 - Prefer more unique terms first

Lucene is very smart about how it ranks results. Results are not simply returned for every document that has a matching term from the search query. For example, one way that Lucene is smart about scoring and ranking is how it takes into account the uniqueness of each term when determining a document's relevance score.



Flexible Queries

- Phrases
`"star wars"`
- Wildcards
`star*`
- Ranges
`{star-stun}`
`[2006-2007]`
- Boolean Operators
`star AND wars`

This is just a small subset of the types of queries that Lucene can support. Some query types such as wildcard and range queries have a potential to cause heavy load on the Lucene server, so Lucene makes it easy to disable certain types of queries while allowing all others to proceed through the system. This gives programmers better control and allows the system performance to be more predictable.

Field-specific Queries



- For example

```
title:"star wars"  
AND  
director:"George Lucas"
```

Field-specific queries can be used to target specific fields in the Document Index. Note that this example uses a Boolean Query to unify two field-specific queries.

Sorting



- Can sort any field in a *Document*
 - For example, by Price, Release Date, Amazon Sales Rank, etc...

By default, Lucene will sort results by their relevance score. Sorting by any other field in a Document is also supported.

Lucene Internals



Everything is a Document



- A *document* can represent anything textual:
 - Word Document
 - DVD (the textual metadata only)
 - Website Member (name, ID, etc...)

In Lucene, everything is a Document. A Lucene Document need not refer to an actual file on a disk, it could also resemble a row in a relational database.

Each developer is responsible for turning their own data sets into Lucene Documents. Lucene comes with a number of 3rd party contributions, including examples for parsing structured data files such as XML documents and Word files.

Indexes



- Indexes track *term frequencies*
- Every term maps back to a Document

The type of index used in Lucene and other full-text search engines is sometimes also called an “inverted index”. This index is what allows Lucene to quickly locate every document currently associated with a given set up input search terms.

Basic Indexing



1. Create a Document
 - Add fields to the Document
2. Add the Document to an Index
3. Indexer will *Analyze* the Document
 - We can provide specialized *Analyzers*

Lucene comes with a default Analyzer which works well for unstructured English text, however it often performs incorrect normalizations on non-English texts. Lucene makes it easy to build custom Analyzers, and provides a number of helpful building blocks with which to build your own. Lucene even includes a number of “stemming” algorithms for various languages, which can improve document retrieval accuracy when the source language is known at indexing time.

Basic Searching



1. Create a *Query*
 - (eg. by parsing user input)
2. Open an *Index*
3. Search the *Index*
 - Use the same *Analyzer* as before
4. Iterate through returned *Documents*
 - Extract out needed results
 - Extract out result scores (if needed)

It is important that Queries use the same (or very similar) Analyzer that was used when the index was created. The reason for this is due to the way that the Analyzer performs normalization computations on the input text. In order to find Documents using the same type of text that was used when indexing, that text must be normalized in the same way that the original data was normalized.

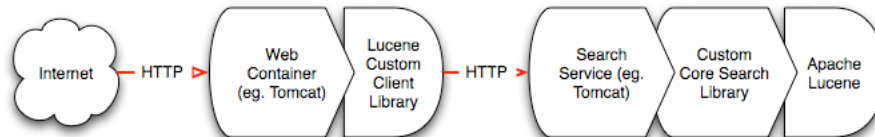
Lucene as SOA



1. Design an HTTP query syntax
 - GET queries
 - XML for results
2. Wrap Tomcat around core code
3. Write a Client Library

One common architecture within which to develop Lucene applications is the SOA, or Service-Oriented-Architecture. This decoupling of Lucene applications from a front-end UI-oriented web application is that it allows each service to scale independently of each other. Also, by using standard protocols such as HTTP and XML to communicate between the web application and the Lucene application, basic building blocks such as load balancers can be deployed to quickly scale up the capacity of the search subsystem.

Lucene as SOA Diagram Single-Machine Architecture



This is an example of a simple Lucene-based Application deployed as a separate internal Service. In this example, the three components that make up the Lucene-based Application are the “Lucene Custom Client Library”, the “Search Service” and the “Custom Core Search Library”. In the ubiquitous MVC pattern that is commonly deployed at the Web Container tier of standard LAMP stacks, the “Lucene Custom Client Library” pictured here would plug directly into its own DAO in the persistence layer.

Lucene Scalability





Scalability Limits

- 3 main scalability factors:
 - Query Rate
 - Index Size
 - Update Rate

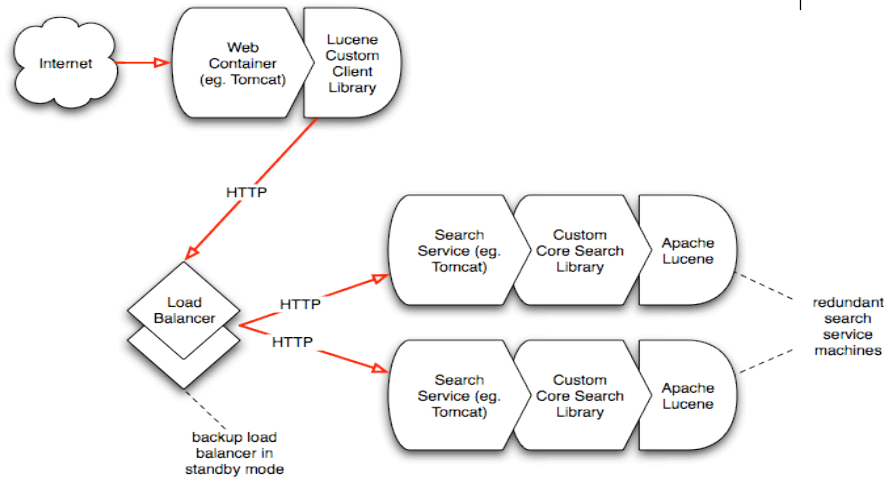
Query Rate Scalability



- Lucene is already fast
 - Built-in caching
- Easy solution for heavy workloads:
(gives near-linear scaling)
 - Add more query servers behind a load balancer
 - Can grow as your traffic grows

Lucene was built from the beginning to be a high-performance full-text search engine. Despite the performance constraints of Java and the JVM, Lucene is blazingly fast.

Lucene as SOA Diagram High-Scale Multi-Machine Architecture



Here is an example showing how a load balancer can be deployed to add additional capacity and redundancy to an internal Lucene-based Search Application. To achieve high-availability, we must simply deploy $N+1$ Lucene boxes, where N is the maximum number of boxes needed to service the peak search load.

Index Size Scalability



- Can easily handle millions of Documents

If you need bigger:

- Built-in multi-machine capabilities
 - Can merge multiple remote indexes at query-time.

Lucene is very commonly deployed into systems with 10s of millions of Documents. Although query performance can degrade as more Documents are added to the index, the growth factor is very low. The main limits related to Index size that you are likely to run in to will be disk capacity and disk I/O limits. Luckily for developers facing these limitations with their massive data sets, Lucene provides built-in methods to allow queries to span multiple remote Lucene indexes.



Update Rate

- Lucene is threadsafe
 - Can update and query at the same time
- I/O is limiting factor

Strategies for achieving even higher update rates...

Update rate is often a limiting factor for Lucene applications which require up-to-the-second index updates. Higher levels of performance can be achieved by reducing the need for instant indexing. Further performance can be achieved through the following strategies:

Vertical Split



For big workloads:

- Centralized Index Building
 1. Build indexes apart from query service
 2. Push updated indexes on intervals

As an intermediate step to reduce load on a Query server, the Indexing can be performed on a separate machine and then the index files can be copied to Query machines when ready.

Horizontal Split



For Huge Workloads:

1. Split data into *columns*
 - Like a *pivot table* in a DB
 - (also known as a *federated database*)
2. Merge columns for queries
3. Columns only receive their own data for updates

A much more involved way to achieving high levels of update performance can be had by dividing the data into separate “columns”, or “silos”. Each column will hold a subset of the overall data, and will only receive updates for data that it controls. This means that each column will only receive a fraction of the overall updates, reducing the required load and improving overall throughput. By taking advantage of the remote index merging query utility mentioned on an earlier slide, the data can still be searched in its entirety without any loss of accuracy and with negligible performance impact.

The End

Thank You

Please see <http://www.codemass.com> for more information

