Reverse Engineering Solr: Apache's Open-Source Enterprise Search Platform

COMP 529 Sofware Architecture: Project Submission 2

Alexander Messina
260526155
alexander.messina@mail.mcgill

Andrew Fogarty
260535895
andrew.fogarty@mail.mcgill

Kyle Brumsted
260454054
kyle.brumsted@mail.mcgill

ABSTRACT

When one thinks of a search engine, websites like Google, or Bing are the first that come to mind. However, search engines do not necessarily need to be for finding web sites. If a store has a large catalog of products, providing a good search feature is important to meet customer needs. A search engine will generally have an internal document representation, some functionality to receive user input, some functionality to return results, and that is without including the search algorithms. Additionally, if the service is very popular and often comes under heavy load, some mechanism to distribute search services becomes very important. These are not trivial tasks to accomplish.

Since information retrieval is such a common problem, many projects have surfaced seeking to create platforms which offer out of the box search functionality. One of these is Apache Solr, an open source project which offers fast, reliable, fault-tolerant, scalable search. It offers a wide variety of features such as hit highlighting, faceted search, and real-time indexing. While there is a lot of documentation pertaining to its use, there is little on its internals. In this paper, we aim to present the architecture of Solr, and provide a detailed analysis of its benefits and limitations.

1. INTRODUCTION

In many web applications and data centered systems, the problem of storing and searching data in a fast and reliable manner becomes an important one to tackle. Traditional relational database systems, such as Microsoft SQL Server or PostgreSQL, are often ill-equipped to handle these kinds of search problems.[5]

Large stores of information are often unstructured and hard to fit within the confines of relational databases.[5] For example, document titles, authors, descriptions and other types of metadata may be needed in some project. Search engines work particularly well on these data sets because they are made for information retrieval from the ground up.

Apache Solr provides out-of-the-box search engine functionality and can be used as a component in larger applications. Solr instances are highly customizable. Therefore, Solr can be implemented to handle a variety of problem domains. These problem domains range from industrial software applications to academic research applications.[1]

For example, the Distributed Digital Music Archives and Libraries Laboratory (DDMALL) of McGill University uses Apache Solr to create and store search-able representations of Optical Music Recognition (OMR) data. The DDMALL develops client applications that interface directly with Solr. These client applications provide graphical interfaces for musicological researchers to be able to query the OMR data. Solr's easily-programmable software interface makes these client applications possible.[3]

This example demonstrates how Solr can be used as both a standalone application and a single component within a larger integrated system. The example also demonstrates that Solr can be used to store both traditional textual data as well as string-serialized representations of non-traditional data from a variety of application domains.[1]

Solr was created in 2004 by Yonik Seely at CNET Networks as a search service for its own website. CNET eventually gave the source code to Apache, which released it as an open source project in 2008. Since then, it has gained widespread adoption, being used by companies such as Instagram, NASA, and even the White House website.[2] It has about 100 commits per month and a large community developing plugins and new features.

Solr is not the only Search Platform built on Lucene; elasticsearch was released in 2010, touting better distributed functionality and a simpler setup as distinguishing features. Both applications have evolved a similar feature set today, and the changes in Solr's feature set have been partly due to the extra competition.

Solr offers quality attributes such as reliability, scalability, and performance. It is an open source search platform written in Java which offers some NoSQL features.[7] Solr is built on top of Apache Lucene, a powerful search library which provides an API for indexing and searching as well as other features such as spellchecking and highlighting. Queries are made in a RESTful format, with HTTP requests, while returned documents can be in many formats including XML and JSON.[1]

Solr's behaviour can mostly be managed using XML configuration files, while its state can be monitored from an administration UI. Data can be indexed from XML, JSON, CSV, and binary files. Rich text formats can also be indexed using Apache Tika.[1]

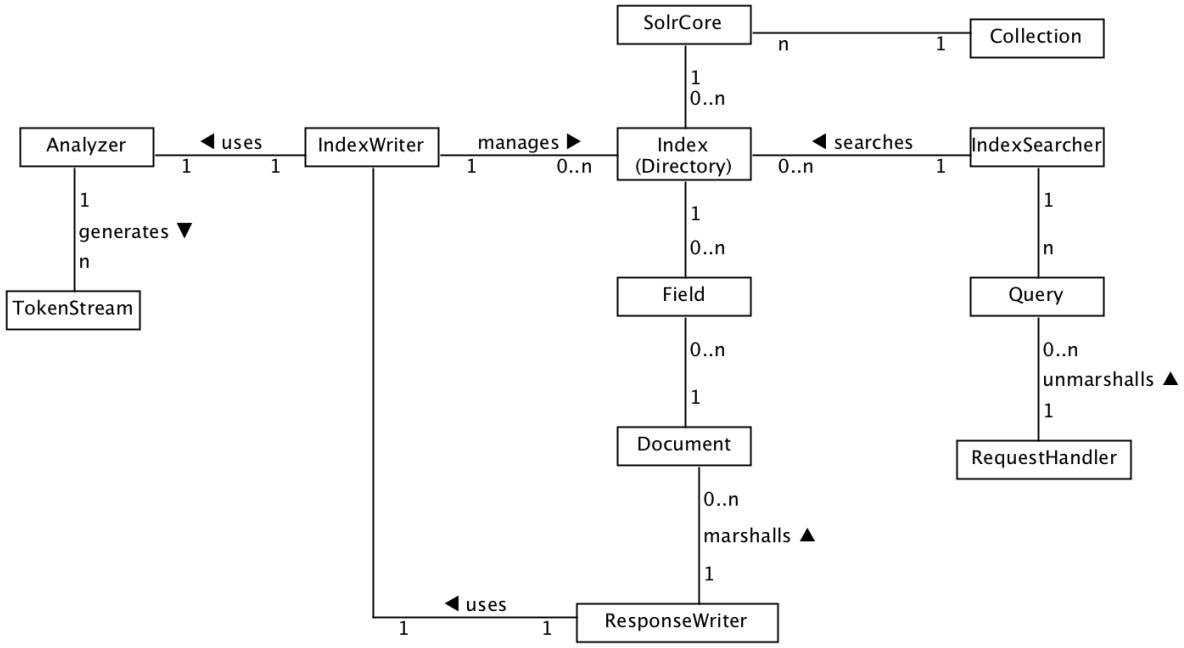

Figure 1: A view of Solr's Domain Model

For high volume applications, Solr has a distributed mode called SolrCloud which uses Apache Zookeeper to manage many cores and indexes. Data sets can be separated into shards for higher throughput, the management of which is handled with Zookeeper. Because of this feature, Solr is now becoming popular as a NoSQL database.[4]

Solr has been designed with plugins in mind in order to better adapt to different production settings. To name a few, custom `RequestHandlers`, `Analyzers`, `SearchComponents` and `Fields` can be supplied in order to get specific functionality out of Solr. These custom classes will generally implement interfaces for whichever element they replace.[6]

These qualities give us an indication of the solution space for the design of good search engines. An ideal search engine would be easy to integrate with the rest of the application, provide a flexible range of features which developers may use as they desire and be fast and reliable. Our goal is to describe the architecture of Solr and determine if it maps onto the desired quality attributes for this problem space.

2. TECHNOLOGY

Solr is first and foremost a Java based application. It is built on top of Lucene and uses other Apache tools such as Tika and Zookeeper.[1] Apache Tika allows Solr to index documents in rich text formats such as Word documents or PDF. Apache Zookeeper is a framework to configure distributed services. It consists of nodes which contain the state of an application's nodes. Solr use Zookeeper's services to manage much of it's distributed function.

The code to handle its state, querying, indexing and searching are all written in Java. Design Pattern wise, there are Factory Patterns, Composite Patterns, Decorator Patterns and Strategy Patterns. Some examples of the Factory design pattern can be found in the `Analyzer` and `Query` classes. Composite and Decorator can be found in `Tokenizer` and `TokenFilter`. Strategy can be found in different searches as well as in the merging of indexes. These design patterns are implemented with modifiability in mind.

XML is used to configure various parts of Solr. `Solrconfig.xml` is the file used to define and configure the state of Solr while `schema.xml` will specify the fields which documents can contain and how they will be handled when indexed or queried. Solr offers a REST API to get and modify the information contained in `schema.xml` although there is no such API for `Solrconfig.xml`.

Solr utilizes several bash scripts in order to carry out tasks which require assistance from the operating system. The most important functionality driven by the scripts is of course starting and stopping Solr, but they also enable system administrators to manage Solr and its associated resources in a flexible manner. Options that these scripts enable include running the process in the foreground or background, enabling SolrCloud mode, switching ports or specifying a specific startup time. Additionally, scripts are provided to create extra cores (for SolrCloud) as well as initializing indexes.[1]

Solr can be run in two different ways; either as a web application or a standalone desktop application. As a web app, it is compiled into a WAR file (Web application ARchive file) which then must be deployed from within a servlet container

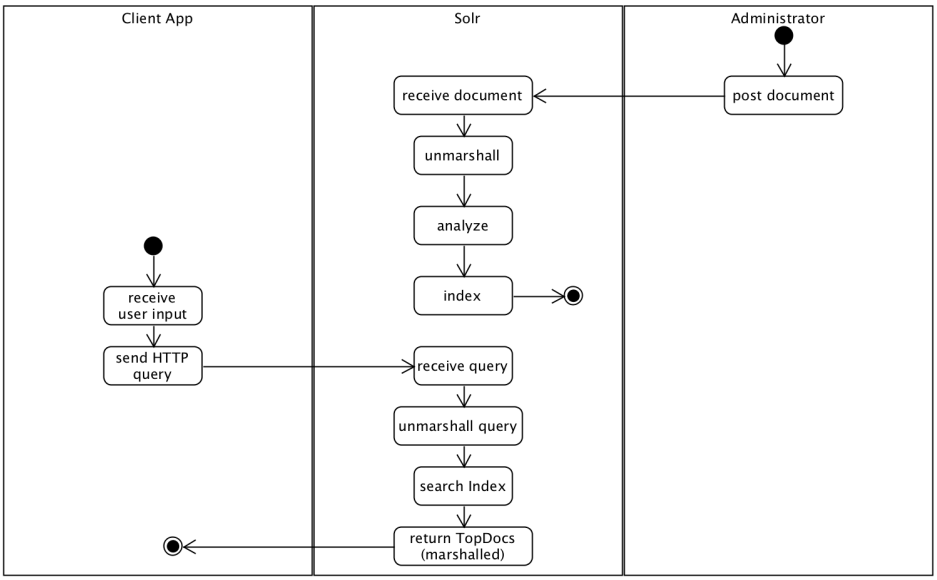

Figure 2: The two principle modes of interacting with Solr

such as Apache TomCat.

Solr can be organized broadly into two pipelines. One of the pipelines would be for indexing data, in which files in a number of formats are analyzed and then stored. The product of this is an inverted index of terms or phrases which point to all documents that contain them. The second pipeline is the query. In this case, the query itself is analyzed and is used as an input for a search algorithm.

This is of course, a simplification. Elements of the Data Centric Style, Data Flow Style, Hierarchical Style and Distributed Style are also present. For example, Solr is being used as a NoSQL database in many instances and in cases where large sets of data need to be indexed or searched, SolrCloud provides a solution. For the Data Centric style, Solr uses a type of Repository Architecture.

The Solr nodes themselves resemble a Master-Slave Architecture from the Hierarchical Style, although a few differences distinguish them. To allow for high availability, data sets can be replicated across nodes. A leader node is elected by Zookeeper, but the leader has the same responsibilities as the replicas with the added task of distributing documents to be indexed to the replicas and then reporting back on the success. More about how each of these architectures are implemented is explained below.

3. INSTALLATION OVERVIEW

The Apache Solr source code repository is divided into two top-level packages: Lucene and Solr. The Lucene package contains the complete source code for Apache Lucene. The Lucene package can be compiled independent of whether or not Solr has been compiled. The Solr package depends on Lucene in order to be compiled.

Apache Ant is used to configure the particular instance of the code repository to be built on the host machine. For example, Ant can generate project definition files so that Solr can be imported into and compiled with an integrated development environment (IDE) such as Eclipse or IntelliJ IDEA. To prepare the Solr source code for use with IntelliJ IDEA, we first ran the command `ant ivy-bootstrap` to generate the Ant build configurations and then `ant idea` to generate the IntelliJ IDEA IDE configuration files. To generate Eclipse project configuration files, we would run `ant eclipse`. The project definitions include configurations in order to run the complete Lucene/Solr JUnit unit test suites. Ant can also generate Apache Maven build configuration files if needed.

The Lucene package contains the analysis, benchmarking, classification, codec, core, demo, expressions, facet, grouping, highlighter, join, memory, misc, query, queryparser, replicator, sandbox, site, spatial, suggest, and tools modules. Lucene's modules are highly decoupled, and each compiles as a standalone Java JAR. Lucene modules can be swapped in-and-out for any particular Solr installation. This decoupled modularity allows for aspects of Lucene to be customized for a particular instance.

The Solr package contains helper scripts for Solr Cloud development, contrib libraries, the Solr Core source code, documentation, examples, server deployment configuration files, the SolrJ source code, the Solr test framework, and the Solr control panel web application source code.

Solr Core is the most important module as it contains the code that handles Solr's main functionality. Solr Core includes the analysis module, the SolrJ embedded client module, the cloud module, the core module, the highlighting module, the CSV encoders and decoders, the logging module, the parser, the request handler, the response generator, the REST module, the schema parser module, the search

module, the servlet module, the spell checker, the storage module, the update handler, and various utilities.

The analysis module invokes Lucene's text analyzer and generates streams of tokenized text. The cloud module allows Solr Core to integrate with Zookeeper for use with SolrCloud. The core module is the "backbone" of Solr Core and initiates all execution. The highlight module highlights the portions of query results that match the query. For example, a client might search for "cat" and receive "*cat*astrophe" as a highlighted response from Solr. The REST module provides Solr with a Representational State Transfer (REST) API that allows client applications to make queries through HTTP requests. The schema module parses the Solr configuration `schema.xml` file.

The search module handles Solr's search execution and core integration with Lucene. The servlet module provides Solr with an HTTP API. The storage module handles saving the Solr database to the file system and caching data in memory and the file system. The update handler module makes changes to the data storage, including adds, deletes, commits, and optimizations.

SolrJ is a Java client library that allows Java applications to interface with Solr in order to make queries. The Solr control panel application is a back-end web application that provides a HTML-based graphical user interface for administrators to be able to make Solr queries and browse Solr logs and statistics. The Solr control panel does not have a user authentication system. When activated, the control panel should not operate on a publicly-accessible port. If the port is publicly accessible then the port becomes an attack vector as foreign clients can make queries.

One of the strengths of Solr's highly-modular design is its simple plugin loading system. Developers can write plugin programs that hook into the Solr Core code to add or modify features. Like the Solr Core, Solr plugins are written in Java and compiled into JAR files. Thus, developers can use Solr's plugin system to create custom interfaces between Solr and other Java software components.

Once a Solr plugin has been compiled into a Java JAR file, the JAR is placed in the `lib` directory of the Solr Core module. Then, the `solrconfig.xml` configuration file must be modified to include a reference to the new plugin JAR. Solr is then re-compiled, producing a new WAR file that contains the new plugin JAR. Solr contains a custom Class Loader that handles the loading of all of the included plugins automatically.[6]

This plugin system addresses the quality attribute of scalability. As a particular instance of Solr scales up, plugins can be written to allow the Solr installation interface with new systems or implement new features. This process can take place without having to re-index the entire database. This aspect of scalability is particularly useful because mission-critical software systems often change drastically over time as new organizational objectives emerge. Developers can use plugins to modify their Solr instance to handle new requirements or quality attributes over a span of time without interrupting the Solr database. Once their newly customized version of Solr has been compiled with the relevant plugins, the developers must only swap out the old Solr WAR file for the new one.

Figure 3 shows an example of Solr deployed as a component of a web application. An Apache Jetty server hosts the Solr instance and the Solr Admin WebApp. A separate Nginx server hosts a public-facing web application that handles customer client PC requests via HTTP. The public-facing application makes REST calls over TCP/IP to the Jetty Server to perform Solr queries. The client-facing application uses the Solr query results to build views that are returned to the customer client PC.

It is important to note that the customer does not interface with Solr directly, but only through the public-facing web application. The Jetty Server can thus filter-out query requests that do not originate from the Nginx Server. Therefore, the public-facing web application on the Nginx server doubles as a security layer that restricts the types of queries that can be executed by the public.

The Jetty Server can also restrict access to the Solr Admin WebApp such that the WebApp only accepts requests from localhost. This introduces another layer of security because only users that have Unix/Linux user permissions on the machine hosting the Jetty Server can access the Solr Admin WebApp. If an admin wishes to connect to the Solr Admin WebApp remotely, they must use SSH to tunnel into the machine hosting the Solr Admin WebApp.

4. DISCOVERY PROCESS SUMMARY

To discover and understand the Solr architecture, we first downloaded and extracted the latest Solr code repository from the Apache Solr website.[1] Exploring the source code repository provided the best, most hands-on access to the Solr architecture. As described in the Installation Overview section, the Solr source code repository is very clearly structured. That structure is designed to present the division between the Lucene and Solr components in a clear manner. Running the Ant build configuration commands described in the Installation Overview section allowed us to import the Solr build configurations into our IDEs and also run the relevant JUnit tests.

Searching through the Source code repository, we found that the source code is well annotated. The Apache developers have filled the Lucene and Solr source code with descriptive comments. There are also files containing descriptions of the modules. The source code is so well annotated that the entire Solr JavaDocs can be generated from the source code. These JavaDocs are at least as descriptive as the JavaDocs for the standard Java Libraries. We ran scripts on the source code repository to generate helpful metrics.

5. CRITIQUE OF EXISTING CONTEXT

The most helpful context for investigating the Apache Solr architecture has been the in-code annotations and the JavaDocs that can be generated from them. In an indirect way, Solr's detailed code annotations address the quality attributes of

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

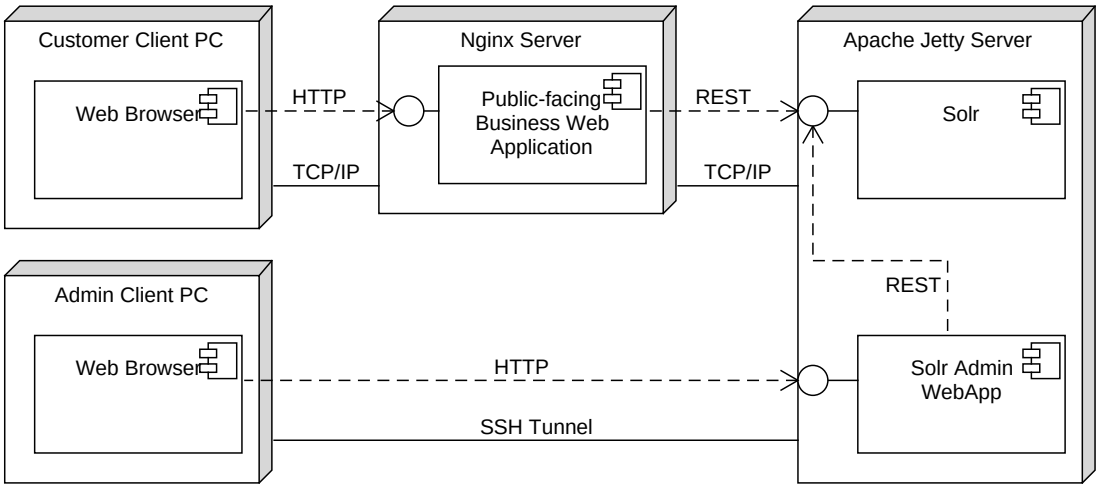

Figure 3: Example Solr deployment structure

reliability, scalability, and performance. Consistently well-annotated source code contributes to reliability because it allows developers to better understand how and why a particular component operates the way that it does. Thus, developers are better equipped to address fixes and improvements that can improve reliability.

Similarly, well-annotated code allows developers to more quickly implement changes that address scalability and performance concerns. Solr is an open-source application at heart, and is designed to be as easily extendable as possible so that it can continue to be used to address new application domains in the future.

The least helpful context has been from the Solr Wiki, which is somewhat incomplete and contains some outdated pages. The structure of the Wiki is also less consistent than the structure of the JavaDocs. The JavaDocs are generated from the code itself, and are therefore as up to date as possible with regards to the structure of the Solr source code. For example, some Solr Wiki pages which contain instructions (either for Solr component installation or customization) do not clearly specify which version of Solr the instructions apply to.

6. ARCHITECTURAL DESCRIPTION

Solr is made up of a mix of architectural styles which tailor its structure to fit the desired quality attributes. These styles include the Data Centric Style, the Data Flow Style, the Implicit Asynchronous Style, the Hierarchical Style and the Distributed Style. In this section we will examine the specific architectures that Apache chose to implement that fulfill these different styles.

At the most basic level, Solr is a data-centered application; in order for a search engine to be useful there must be some large, stable data source to be searched. As such, Solr uses a Data Centric style by implementing the Repository Architecture. At the heart of every instance of Solr is a passive repository housing all of the indexed data ingested by Solr; administrative clients can add data while regular clients can query it. Thus, the clients control the interaction with the data. By implementing a passive repository architecture, it is ensured that Solr can scale to handle any size of data set and that the data is stable, as these are essential properties of any database system.[1]

As described above, clients interact with Solr either by adding more data to the repository, or by executing queries over the data. To ensure that data can be ingested in a flexible manner, Solr utilizes a Data Flow Style by implementing a Batch Sequential Architecture in order to transform these two types of requests.

Every time an administrator wants to add new data, they input Documents (Solr's basic data structure) which are analyzed and transformed by several different loosely coupled transformation modules. The purpose of this is to index these Documents; the administrator specifies Fields (words or phrases) which act as keys for the list of documents in which the fields are found. `Analyzers` first break up the document into tokens (as specified by the administrator in `schema.xml`) which are then sent through any number of `TokenFilters`. After this filtering, the remaining tokens and their associated Document references are passed to `IndexWriters` so that they are added to their relevant indexes.[1]

Queries are also sent through several transformation modules as batched data; `RequestHandlers` receive HTTP queries, which are parsed using `Analyzers` which produce tokens that `IndexSearchers` utilize when searching the data. The final result is then passed to a `ResponseWriter` in order to respond to the client. This canonical example of Batch Sequential Architecture allows administrators extreme flexibility both in the way they index Documents and handle queries, as any of these modules can be deleted or replaced. [1]

The clients interacting with any Solr based application are

all independent and unaware of each other. Elements of the Implicit Asynchronous style are implemented through a Buffered, Message-Based architecture that handles all of these independent clients. The administrative clients can be considered producers while the regular clients are consumers. The administrators add new data to Solr, which gets parsed and indexed as described above and forms the content of the message in this architectural style. The clients who execute queries over the data in Solr then consume these messages as responses to their queries. Both search requests and indexing requests must be handled by buffers in order to ensure the integrity and consistency of the system.

In order to scale up for larger applications, Solr offers SolrCloud; a platform for spreading data over multiple servers in order to handle more data and more queries over this data. In order to manage this cluster of servers, Solr implements a Hierarchical Style using a modified Master-Slave Architecture.

When the index is too large to fit on a single server, SolrCloud shards the data onto multiple servers; each server holds a different part of the data. In order to make sure that queries can still be served if a single server goes down, each unique shard of the index is replicated onto one or more additional servers. However, for each set of servers that houses the same portion of the index, only one of them is the 'leader'; only the leader can decide when data can be written on that shard. This keeps the index consistent across replicated servers. In order to keep track of which servers are up, which are down, which hold which parts of the index and which are the leaders, Apache utilizes a framework called Zookeeper. [4]

The server running Zookeeper is the master node, while all the others are slaves. Zookeeper controls the behaviour of all other nodes in the cluster; it decides which nodes are leaders, how to replicate the shards, and what to do when a given node goes on or offline, including electing new leaders for a shard. Utilizing such an architecture allows the Solr application to be very fast and keeps the data accurate.

This architecture, however, is not a canonical example of Master-Slave. Zookeeper does not hold any locks for writing new data, instead the leader of each shard holds this responsibility. Additionally, Zookeeper does not decide which nodes receive which queries. The queries can land on any given node based on a hashing function, at which point that node parses the query then sends it along to a node holding the relevant shard from which the query wishes a response. Thus, the architecture implemented is more of a mix of Master-Slave and Multi-Tiered.

Finally, Solr utilizes a Distributed Style in order to facilitate communication between the client making queries, and the server(s) holding the index, specifically the Broker Architecture. The application holding Solr acts as a Broker to pass messages between the client making requests and the Solr Servers responding. This use of the Broker architecture aids in load balancing, as well as keeping the interface between client and server simple.

7. CRITICAL ANALYSIS

One main weakness in the Solr architecture comes from its implementation of SolrCloud. Solr provides the SolrCloud functionality for applications whose index or query load is too great to handle on a single server. SolrCloud allows you to shard your index; that is, to distribute the documents across multiple servers in order to load balance. However, in order to ensure integrity, the data on each server must be replicated in case one server goes down.

As was outlined in the Architectural Description, Solr uses Zookeeper in order to manage all of the nodes in a cluster. At any given time, a new node could be coming up, a leader could go down, multiple new documents are being indexed into different replicas of the same shard, all the while queries are being executed. It is necessary that Zookeeper allow such a complex state space to be possible in order to keep the Solr application running smoothly and with maximum flexibility.[4] However this complexity comes with a steep price.

Due to the nearly limitless number of different scenarios that could be happening at a given time, it becomes virtually impossible to test the implementation of SolrCloud and Zookeeper. This inability to perform rigorous testing means that many non-anticipated problems could arise, some small and some large. It is possible that the data between replicas becomes inconsistent and that responses to queries will thus not be correct. Or, much worse, entire portions of the index can be lost depending on how the nodes go down.

The only effective way to test such a system where incredibly complex states can arise is through random killing of nodes. Solr uses Chaos Monkey, a testing framework that randomly selects nodes to take down while applying extreme stress to the system.[4] Although this form of testing can effectively simulate complex scenarios it has two main issues. First, even when Chaos Monkey achieves its goal of getting the entire system to fail, it generally does so by achieving a very complex state which means figuring out the reason for this failure is a non-trivial task. Second, due to the random nature of Chaos Monkey, there is no guarantee that all of the edge cases will be simulated that can cause a crash. Thus, there is not real way to provide and absolute guarantee of the fault tolerance of SolrCloud.

Another point of criticism relates to Solr's component-based paradigm; Solr's modifiability may require a lot of domain knowledge, hampering its effectiveness as a Product Line architecture. An out of the box search engine is often times going to be integrated into an existing application for which search is a part of the solution space, not the solution space in its own right. If a team implementing search is applying an product line architecture for their project, then there is a possibility that they are not experienced Java developers.

Solr presents a great degree of configurability and manageability for its adopters through the strong plugin architecture it uses and the admin UI interface. Configuration is done with XML files, so getting a working instance of Solr indeed does not require knowledge of Java development.

Solr has been built with a very modular architecture meant to swap elements in and out, but this does not map so well

onto configurability for those that are inexperienced with Java development. The system of plugins requires Java to take advantage of Solr's flexibility. This means configurability may be in the hands of plugin developers.

One example of a missing feature having been added is the addition of Nested Documents to Solr. In search, there are some cases where the relations between fields of a document may be important. For example, a database of first and last names would have been flattened into a list of first names and a list of last names. Queries trying to match a person would not work in this case because the relation in between fields is lost.

Some solutions not requiring the use of Java would be to organize the schema into numbered fields so that the first person's first name were assigned to the FirstName1 key and so on. This does preserve the relation between names, but it is not very scalable for a large number of people. Block-join was Solr's solution to nested documents but it was only implemented in Solr 4.5.

One interesting feature is percolation, present in elastic-search. Percolation allows developers to index queries instead of documents, and then run documents through the index to see what matches. This is useful in cases where users would subscribe to a feed and get documents matching their interests. A feature like this has not been implemented yet in Solr.

Solr employs the Product Line Architecture in that it can be reused in many applications in different domains and in different ways but it becomes difficult to customize easily if the developer does not have sufficient domain knowledge and is inexperienced with Java.

Another one of Solr's weaknesses is a lack of included profiling or schema optimization tools. When deploying Solr, one must rely on web server profiling tools to generate any performance data about Solr. Solr could be improved by including performance profiling tools in the source code repository that could run without deploying the Solr instance to a live server.

Another one of Solr's weaknesses is that the included Java unit test suite is not entirely multi-platform. Some of the included unit tests will not run on Windows platforms. Considering that one of the strengths of Java is that Java supports multiple host platforms for the JVM, this platform incompatibility is surprising.

8. ALTERNATIVES

The biggest weakness present at the architectural level of Aolr is the implementation of the modified Master-Slave Architecture for handling the distribution of the index and load balancing. At the price of integrity and testability, the Apache engineers chose this architecture for its speed and concurrency, as it allows near real time indexing, as well as simultaneous updating and querying all the while keeping multiple replicas of the data in case a node fails.[4]

However, because this architecture is a mixed one, it allows many complex states to be achieved such that testing is nearly impossible. The implementation of a pure Master-Slave architecture would drastically cut back the complexity of node management and data updating and querying, thus making testability (and therefore integrity) a much smaller problem. With a pure Master-Slave Architecture, instead of having a leader per shard, there would be a single leader that holds all of the write privileges for all of the shards. This would eliminate many concurrency related issues. Additionally, having a single master node handle all of the load balancing could potentially help prevent nodes from going offline. However, these benefits that are gained come with the addition of more bottlenecks in the system.

When designing any architecture, especially for a system as large and complex as Solr, there will always be tradeoffs that are made in terms of the desired quality attributes. In the case of Solr, the engineers chose a more complex clustering system with the disadvantage of integrity of testability, so that they could guarantee faster response times and faster data updating. Alternate architectures, such as the pure Master-Slave, solve some of the problems present in Solr, but introduce new ones. Therefore, choosing any architecture must boil down to deciding which quality attributes are the most important and for Solr, that is sacrificing a bit of integrity for the sake of speed.

9. CONCLUSION

Apache Solr is a top of the line open-source application. Utilizing multiple architectural styles in order to best fulfill a wide array of quality attributes, Solr is an impressively versatile, yet very manageable application. Solr was clearly designed with easy customization in mind. With highly cohesive, loosely coupled modules, Solr can be extended, transformed or shrunk down as necessary to perfectly fit the needs of any application needing a search engine.

Although the external documentation for Solr was sparse, the code is organized in an easy to understand way, with very helpful and verbose comments throughout. This lack of readily accessible documentation, coupled with a non-intuitive installation process creates a bit of a hurdle for developers just beginning with Solr, but once these initial road blocks are passed the application becomes much easier to deal with.

Perhaps the most important lesson learned during this analysis is the importance of documentation for large software projects. Not only does documentation provide a means of access for those who wish to become involved with the project in one way or another, but if forces the developers to organize their ideas and abstract the structure of their logic into larger, more digestible architectural designs. This can help to create a feedback cycle that improves the development, as the goals of the engineer are more explicit. Thus, we hope that the Solr developers work towards better documenting their project as they move forward.

One of the most important features that has been added recently to Solr is its framework for distributed indexes, SolrCloud. As applications need to handle larger and more complex data all the time, scalability becomes ever more important, and SolrCloud provides this vital feature. Although there are some issues of testing and data integrity

with the current distributed architecture, Solr is still for the most part stable and as development continues it will only improve in these quality attributes.

Overall, Solr should be seen as an example that other open source developers can follow. Solr is constantly adapting to the needs of the ever-growing computing industry through the use of well thought out architectural design and implementation.

10. REFERENCES

[1] Apache solr reference guide. https://cwiki.apache.org/confluence/display/solr/Apache+Solr+Reference+Guide.

[2] Solr powered. https://wiki.apache.org/solr/PublicServers.

[3] A. Hankinson, J. A. Burgoyne, G. Vigliensoni, and I. Fujinaga. Creating a large-scale searchable digital collection from printed music materials. In *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, pages 903–908, New York, NY, USA, 2012. ACM.

[4] Mark Miller. The solrcloud architecture. https://www.youtube.com/watch?v=WYVM6Wz-XTw.

[5] Martin Fowler. Introduction to nosql by martin fowler. https://www.youtube.com/watch?v=qI_g07C_Q5I&list=PLB9uLawXQoggpG9MGz5v9wDodr9f_p4lg.

[6] Solr Wiki. Solr plugins. https://wiki.apache.org/solr/SolrPlugins.

[7] Yonik Seeley. Solr4 the nosql database. https://www.youtube.com/watch?v=WYVM6Wz-XTw.