On the Efficacy of Keyword Searches to Find Meaningful Architectural Knowledge in Open-Source Software Mailing Lists

Bachelor Thesis for Computing Science

Andrew Lalis andrewlalisofficial@gmail.com

Supervised by **Dr. Mohamed Soliman** m.a.m.soliman@rug.nl

June 23, 2022

university of groningen / faculty of science and engineering

Abstract

This section will contain a brief overview of the research and conclusions. Test

Contents

Introduction

In this paper, we'll explore the efficacy of using targeted keyword search queries to find architectural knowledge in mailing lists for open-source software projects. More simply put, we'll build tools, collect data, and analyze that data to qualitatively determine how effective certain keyword-based search queries are at finding useful information in large sets of emails, sent in mailing lists for developers to communicate about large open-source software projects.

Software Architecture

Behind every software system you interact with, including the one used to read this paper, is a set of elements, relationships, and rationale that define the software's architecture. While on the surface, you see the *implementation* of the software, the *architecture* is concerned with what elements, their interactions in order to provide a framework for satisfying the requirements of the system.[7] Continuing our example, the software you're using to read this paper probably uses an architecture that includes elements like a PDF renderer, file reader, and a controller to manage your button clicks and mouse scrolling. Collectively, this set of components, design decisions, rationale, assumptions, and context together are defined as the *architectural knowledge* of the system.[3]

In Perry's 1992 paper, "Foundations for the Study of Software Architecture", he posits that "we have not yet arrived at the stage where we have a standard set of architectural styles with their accompanying design elements and formal arrangements," and that each system is "a new architecture".[7] While we are perhaps nearer now, 30 years later, to that mythical stage of architectural standardization, indeed many new systems are genuinely new architectures, with new elements or entirely novel approaches to communicating between components, where engineers must consult both their acquired skills and the collective knowledge of their peers, to make a best-effort to build a system to satisfy their requirements.

Despite the plethora of resources available to the modern software engineer, we still often see *architectural knowledge vaporization*[5] because of undocumented decisions engineers took when designing the architecture of a system, or because it's impossible or practically infeasible to extract useful information. This dissipation of knowledge is a function of time and architecture size, and can lead to some serious problems that have already established themselves as hallmarks of poor software design in the industry, such as an increasing cost to improve or upgrade a system,[5,7] and a lack of reusability,[5] and an increased cost of maintenance[6] even when no new features are added. These costs can, and often do, reach a point at which it is simply more effective to start over, abandoning completely any established work on an architecture. Through the years, this evolved from reprogramming low-level batch programs for early computing installations,[6] to today's distributed systems. Succinctly, a loss of architectural knowledge, by conjecture, leads to a loss of value and a loss of efficiency, and this paper aims to add additional tools to our collective defense against such regression.

Searching

TODO: Add more background about what searching has been done already, what possibilities there are.

It is important to explore different avenues for acquiring knowledge, especially as the body of information grows exponentially with time. It is difficult for developers and software architects to make informed decisions about their own projects, because the source of their knowledge is distributed in a variety of disparate sources. If we can reliably glean information about software architecture and the successful (and unsuccessful) decisions that other field experts have made, we can make this knowledge more accessible for all.

Research Questions

The main research question that this paper attempts to answer is summarized in the following question:

What architectural knowledge exists in open-source software development mailing lists?

In addition to the main question we're attempting to answer, this paper will also discuss several other possible questions and answers that may be obtained using the data originally gathered for the main question.

1. Does there exist a relationship in the order in which architectural decisions are discussed, chronologically in an email thread? Is there significance in the order in which architectural decisions are made?
2. Is there a relationship between the content of discussion in emails, and related issues in issue/ticket boards such as JIRA and GitHub issues?
3. How can we use data gathered in this research to improve our search queries?

Related Work

This section will contain an overview of lots of different sources and what they've done, and how what I'm doing is different.

Methodology

For the purposes of this research, we will focus on analyzing the contents of mailing lists from three major open-source projects from the Apache Software Foundation: Hadoop, Cassandra, and Tajo. Mailing list data will be obtained from lists.apache.org, and this data will be indexed and searched over using Apache Lucene. A subset of emails from these mailing lists will be categorized based on the type of architectural design decisions they contain, in order to conjecture about the efficacy of searching.

Note: All software and components developed for this research are available on GitHub.com under the ArchitecturalKnowledgeAnalysis organization. All software is licensed under the permissive MIT license, and may be freely used or redistributed however you see fit.

Choosing Sources

While previous work by den Boon chose to analyze data from both issue tracking boards and mailing lists from open-source software projects,[3] this research will limit the scope of sources to just mailing lists, since that is the focus of our research questions. More specifically, a *mailing list* is a "mechanism whereby a message may be distributed to multiple recipients by sending to one address."[4] Put more plainly, someone may *subscribe* to a mailing list, and in doing so, they will receive any email which is addressed to that list. Likewise, that person may also send an email to the list's address, and it will be broadcast to all other subscribed recipients. In the context of software development, mailing lists have been used extensively since their inception as a way to collectively discuss and make design decisions regarding large open-source projects. Because of this, we can expect mailing lists for software development to be particularly rich in architectural knowledge, and they make a good candidate for analyzing the effectiveness of various search approaches.

Within the domain of mailing list communications, we can further narrow our search down to software projects which we can reasonably expect to contain a large amount of, and large variety of architectural decisions. These would generally be large projects with many moving parts, which have a broad range of applications and societal uses. The Apache Software Foundation (ASF) is a leader in these sorts of projects, with many open-source systems for managing, processing, and searching through big data, which find their usage in various state-of-the-art enterprises and research endeavors, notably NASA, Facebook, Twitter, Netflix, and so on.[2] From the ASF, we chose mailing lists from the following three projects for analysis:

1. Cassandra - An open source NoSQL distributed database trusted by thousands of companies for scalability and high availability without compromising performance.
2. Hadoop - A framework that allows for the distributed processing of large

data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

3. Tajo - A robust big data relational and distributed data warehouse system for Apache Hadoop. Tajo is designed for low-latency and scalable ad-hoc queries, online aggregation, and ETL (extract-transform-load process) on large-data sets stored on HDFS (Hadoop Distributed File System) and other data sources.

Project descriptions obtained from the respective projects' homepages.

For each of the above projects, we obtained one or more mailing lists dedicated to the discussion of the project's internal development, thus excluding irrelevant discussions about usage and user issue reports.

For Cassandra, we chose the following mailing lists:

- dev@cassandra.apache.org

For Hadoop, we chose the following mailing lists:

- common-dev@hadoop.apache.org
- hdfs-dev@hadoop.apache.org
- mapreduce-dev@hadoop.apache.org
- yarn-dev@hadoop.apache.org

For Tajo, we chose the following mailing lists:

- dev@tajo.apache.org

Fetching and Processing Sources

The first step to being able to analyze and identify architectural knowledge in mailing lists is, of course, to obtain the emails from the mailing lists in the first place. For this purpose, the EmailDownloader utility library was developed, and for parsing and preparing data for use in our datasets, the MBoxParser utility library was developed.

Downloading

The EmailDownloader utility offers an asynchronous interface for downloading a series of MBox (email archive format) files to a directory. Because all of the mailing lists used by this research come from the Apache Software Foundation, the library includes an implementation for downloading from the ASF's mailing list internal API at https://lists.apache.org/api/mbox.lua, with support for rate-limited downloading and the ability to detect long periods of inactivity and short-circuit prematurely to save time.

While this library is primarily designed to be included in other Java projects for a complete workflow, it can also be run as a standalone program that offers the same interface on the command-line, if you wish to use it like this.

The end result is that we are able to download a large archive of all emails sent in a mailing list, since its inception, to today.

Processing

Once we have downloaded a large amount of MBox files, we must parse their contents to extract the individual emails for use in our analysis. This is done by the MBoxParser utility library, which offers an interface for parsing a collection of MBox files and issuing a callback handler for each email that was obtained.

The end result is a collection of many Email objects that are ready for further use, whose format is described in the code snippet below:

```java
public class Email {
  public String messageId;
  public String inReplyTo;
  public String sentFrom;
  public String subject;
  public ZonedDateTime date;

  public String mimeType;
  public String charset;
  public String transferEncoding;
  public byte[] body;
}
```

Note that the MBoxParser library performs a few rudimentary operations to attempt to clean up ill-formatted emails. This includes sanitizing various date formats, and stripping superfluous characters from an email's MESSAGE_ID.

For more details, please see nl.andrewl.mboxparser.EmailContentHandler.

Dataset Format

Once a sufficient number of emails have been downloaded as described in the previous section, they can be ingested by the EmailIndexer API to generate a *dataset*. This dataset is simply a directory containing the following components:

1. An embedded relational H2 database file named `database.mv.db`, which stores all emails, tags, and other key information for the dataset. A schema diagram is provided in this section, and the full schema definition can be found in the appendix.

2. An `index` directory in which the indexes generated by Apache Lucene are stored.

3. A `metadata.properties` file that holds some meta information about the dataset. As of the time of writing, the only metadata stored in this

file is the dataset version, which, again at the time of writing, is version 2.

Database Schema

The relational schema is the core of the email dataset, and defines how we operate on and organize the data. It is purposefully designed to be as simple as possible, to make it easy to use the dataset in a variety of applications and environments.

The schema consists of three entities: *emails*, *tags*, and *mutations* (although mutations are not strictly required).

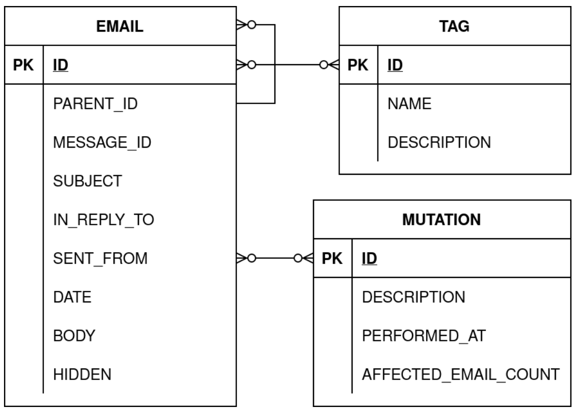


Figure 1: A diagrammatic representation of the email dataset's relational schema.

Emails are, as you'd probably expect, the entity that represents a single email message, sent by someone, in a mailing list. While emails *do* have their own unique `MESSAGE_ID` attribute,[8] they are identified by an 8-byte integer primary key. This reduces the dataset size, improves search performance due to the monotonic nature of incrementing primary keys, and allows for easier referencing of emails by their id, instead of a long, randomly-generated message id.

Also of note is an email's `PARENT_ID`, which is a self-referencing foreign key that references another email entity by its id. This foreign key is the manifestation of the *many-to-one* relationship between an email and its parent, where many "child" emails may refer to one parent email. The parent id is derived, when processing emails, from the email's "In-Reply-To" field. According to RFC 5322, the in-reply-to field, if present, must reference another email's message-id, and this format of replies forms the basis of an email thread's structure.[8]

Tags are entities which can be attached to any number of emails, to offer a method of categorizing emails. Originally in version 1 of the dataset format, tags were not entities but simple texts that were attached to each email, but it was discovered during the development of the tools for this research that there are several benefits to having tags as their own entities: the ability to edit a tag's name, provide a description, and reduce the total dataset size by referencing an 4-byte integer id instead of keeping a copy of a string for each tag applied to an email.

There exists a *many-to-many* relationship between **emails** and **tags**, which one could verbalize as, "A tag may be applied to many emails, and an email may have many tags applied to it."

Mutations are entities that record some change to the dataset which is used as a historical record of how a the emails in a dataset are filtered or modified. Mutations are used at the discretion of any third-party application which interfaces with the EmailIndexer API, and are not required to be used.

There exists a *many-to-many* relationship between **mutations** and **emails**, which one could verbalize as, "A mutation may involve many emails, and an email may be involved in many mutations." It is not required that a mutation be linked to any emails.

Indexing with Lucene

Alongside the relational database resides the `index` directory, as mentioned at the start of this section. This contains the indexes produced by the Apache Lucene indexing library when we build an index using the emails in the database. This index is then used for executing query searches over the database by users.

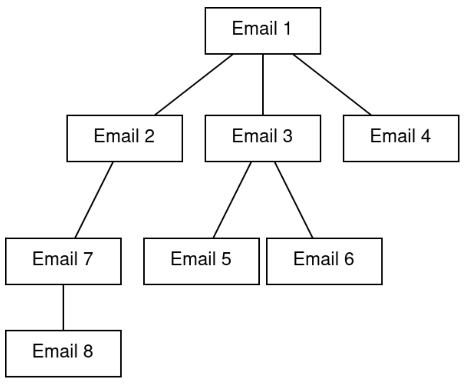
For each email in the dataset, we index the following fields as non-stored string-type fields using the `DOCS_AND_FREQS_AND_POSITIONS` index options provided by Lucene:[1]

- The `SUBJECT` field.
- The `BODY` field.

In addition to these indexed fields, we store the email's id and pre-compute the *root id* (id of the first email in the current email's thread), as these are essential for producing actionable results from any query search on the index.

Email Thread Structure

As previously mentioned, emails have a self-referencing *many-to-one* relationship defined by their "In-Reply-To" field. It is important to understand the consequences of this structure, and how it impacts the dynamics of processing data. In any information system, a self-referencing, many-to-one relationship that an entity participates in, can be interpreted as a *tree* data structure.



Each email can be thought of as a node in the tree, where each reply to that email forms a branch. There is no limit to how wide or deep the tree can become, because there is no universal hard limit on the number of replies to an email, nor on how many replies may be sent in a thread.

In previous research by den Boon, this natural tree structure of email threads was artificially flattened into a sequential list, presumably to facilitate more efficient methods of working with the data.[3] However, in this research, the tree structure of the emails is preserved in the dataset, because it more accurately depicts how real users would navigate a set of emails using any number of existing programs, and therefore can provide a better model for determining the effectiveness of searches. This choice does come with some drawbacks, however.

1. Because emails form a tree structure, navigating the dataset is more computationally intensive, and more complex for a user, than if one could simply traverse a flat list.
2. Analysis is more complex, due to the fact that most algorithms will need to incorporate tree traversal, either iteratively or recursively.

Constructing the Dataset

Now that I've explained all this bullshit, you can put it together to make a dataset.

Results

This section will show visualizations and aggregate data for results.

Conclusion

This section will answer the research questions and discuss further research.

References

[1] URL: https://lucene.apache.org/.

[2] Aleem Akhtar. "Role of Apache Software Foundation in Big Data Projects". In: *arXiv e-prints* (May 2020). DOI: 10.48550/arXiv.2005.02829.

[3] Tom den Boon. "Exploring the effectiveness of search engines for finding architectural knowledge in open source repositories". In: *University of Groningen Student Theses* (2021), pp. 3–30. URL: https://fse.studenttheses.ub.rug.nl/25813/.

[4] R. Gellens. *Mailing Lists and Internationalized Email Addresses*. Internet Engineering Task Force (IETF). Oct. 2010.

[5] Anto Jansen et al. "Software Architecture as a Set of Architectural Design Decisions". In: *Proceedings - 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005* 2005 (Jan. 2005), pp. 109–120. DOI: 10.1109/WICSA.2005.61.

[6] Peter Naur and Brian Randell. "Software Engineering: Report on a conference sponsored by the NATO Science Committee". In: *Working Conference on Software Engineering* (1969).

[7] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture". In: *ACM SIGSOFT Software Engineering Notes* 17 (4), pp. 40–52. DOI: 10.1.1.40.5174. URL: http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf.

[8] Pete Resnick. *Internet Message Format*. RFC 5322. Oct. 2008. DOI: 10.17487/RFC5322. URL: https://www.rfc-editor.org/info/rfc5322.

Appendix

This section will contain larger bits of text or code or figures that aren't well suited to being placed inside the body of the paper.

Email Dataset Schema

The following SQL snippet defines the relational schema which is used for email datasets in version 2 of EmailIndexer API.

This DDL script is written using the H2 dialect of SQL.

```sql
CREATE TABLE EMAIL (
  ID BIGINT PRIMARY KEY AUTO_INCREMENT,
  PARENT_ID BIGINT NULL DEFAULT NULL REFERENCES EMAIL(ID)
    ON UPDATE CASCADE ON DELETE SET NULL,
  MESSAGE_ID VARCHAR(255) UNIQUE,
  SUBJECT VARCHAR(1024),
  IN_REPLY_TO VARCHAR(255),
  SENT_FROM VARCHAR(255),
  DATE TIMESTAMP WITH TIME ZONE,
  BODY LONGTEXT,
  HIDDEN BOOL NOT NULL DEFAULT FALSE,
  CHECK (PARENT_ID IS NULL OR PARENT_ID <> ID)
);
CREATE INDEX IDX_EMAIL_DATE ON EMAIL(DATE);
CREATE INDEX IDX_EMAIL_HIDDEN ON EMAIL(HIDDEN);

CREATE TABLE TAG (
  ID INTEGER PRIMARY KEY AUTO_INCREMENT,
  NAME VARCHAR(255) UNIQUE,
  DESCRIPTION MEDIUMTEXT NULL DEFAULT NULL
);
CREATE INDEX IDX_TAG_NAME ON TAG(NAME);

CREATE TABLE EMAIL_TAG (
  EMAIL_ID BIGINT NOT NULL REFERENCES EMAIL(ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  TAG_ID INTEGER NOT NULL REFERENCES TAG(ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  PRIMARY KEY (EMAIL_ID, TAG_ID)
);

/* Utility tables to record all changes made to the dataset. Each mutation should
↪  affect one or more emails. */
CREATE TABLE MUTATION (
  ID BIGINT PRIMARY KEY AUTO_INCREMENT,
  DESCRIPTION LONGTEXT NOT NULL,
  PERFORMED_AT TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP(0),
  AFFECTED_EMAIL_COUNT BIGINT NOT NULL DEFAULT 0
);
CREATE INDEX IDX_MUTATION_DATE ON MUTATION(PERFORMED_AT);

CREATE TABLE MUTATION_EMAIL (
  MUTATION_ID BIGINT NOT NULL REFERENCES MUTATION (ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  EMAIL_ID BIGINT NOT NULL REFERENCES EMAIL(ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  PRIMARY KEY (MUTATION_ID, EMAIL_ID)
);
```