A Replicable Infrastructure for Empirical Studies of Email Archives

Megan Squire
Department of Computing Sciences
Elon University
Elon, NC, USA
msquire@elon.edu

***Abstract*—This paper describes a replicable infrastructure solution for conducting empirical software engineering studies based on email mailing list archives. Mailing list emails, such as those affiliated with free, libre, and open source software (FLOSS) projects, are currently archived in several places online, but each research team that wishes to study these email artifacts closely must design their own solution for collection, storage and cleaning of the data. Consequently, research results will be difficult to replicate, especially as the email archive for any living project will still be continually growing. This paper describes a simple, replicable infrastructure for the collection, storage, and cleaning of project email data and analyses.**

Keywords—email; archive; mailing list; database; document-oriented database; collection; storage; cleaning

I. Introduction

Email mailing lists are an important artifact of the contemporary software engineering process, especially in transparent, distributed development teams, such as those making free, libre, and open source software (FLOSS). For example, the Apache Software Foundation (ASF) is one of the largest not-for-profit corporations supporting FLOSS development. ASF projects adhere to a policy of transparency in decision-making; specifically, project leaders are directed to "conduct project business on mailing lists"[1]. As such, email archives, and ASF-affiliated emails in particular, are a frequent object of study for researchers interested in the process of software development.

The easy availability of mailing list emails is both a blessing and a curse for research teams. All ASF projects, for example, make their entire mailing lists available for browsing, and many provide raw downloads. However, one of the shortcomings of this high availability is that research teams may not think twice about jumping into collecting and storage of emails for their own analyses, without considering whether it has been done in the past, and how. They may not be aware of the methods that others have already discovered for solving collection and storage problems. The result is that teams may inadvertently duplicate each others' work in the collection, storage and cleaning of the data. Or, they may not design as robust a system for collection and storage as would be required for this large amount of unstructured data. The effect is to increase the amount of time spent in the frustrating collection and storage tasks, thus leaving less time for analyses and sharing of results.

This paper describes a strategy for inverting this situation. The system described here describes a simple, repeatable collection and cleaning infrastructure, using all open source software and commodity hardware, and with code and scripts made available for anyone to use. The system thus provides a lower barrier to entry and a common vocabulary for describing the collection and storage infrastructure. Most importantly, it also provides a common platform for replicating the subsequent analyses. The result is less work in collecting and cleaning, and more opportunities for analysis and sharing.

Section II describes the background of this project, both in terms of a review of existing literature on how emails are used by the research community, and what tools are currently available for this purpose. Section III describes the system that was built, including its system architecture, the data collection procedure, and examples of using the system. Section IV describes several challenges and limitations of the system, including cost and sizing concerns, and a few potential pitfalls in cleaning the data.

II. Background

Email mailing lists from software development projects are useful for learning about project management, team dynamics, and decision-making. For example, a literature survey on how email archives are used to learn about FLOSS processes [2] found a variety of analyses being performed on mailing list archives, including basic statistics (e.g. [3]), social network analysis (e.g. [4]), automated text mining (e.g. [5]) for concept extraction, and mixed methods mining tasks (e.g. [6]) including confirmatory analyses.

A. Limitations of Third-Party Email Archives for Replication and Sharing

There are two main ways that researchers interact with email mailing list archives. For simple browsing and searching, they might use a web-enabled searchable archive like Gmane.org, Mail-Archive.com, or MarkMail.org (e.g. [7]) These web sites collect and store mailing list messages, such as from FLOSS projects (including the ASF projects), and provide a searchable front end to their collection. The searches are typically keyword based. Browsing on these sites consists of drilling down on clickable list names, author names, and the like. Mail-Archive.com and MARC.info both archive email and provide a web interface for searching by keyword. Gmane is a service that archives emails, and then funnels the emails into newsgroups. MarkMail has a fuller feature set, including some dynamically-generated graphs for message counts. It is owned by MarkLogic corporation, created as a proof of concept for one of their commercial projects, MarkLogic server. The FAQ states, "One reason we built the site is to show what MarkLogic Server can do" [8]. However, the FAQ is outdated and the discussion lists and blog have not been active since 2011. The last MarkMail announcement came out in 2010.

Aside from relative recent inattention from some of the site owners, there are no APIs for interacting with any of the services mentioned above. Researchers cannot run regular custom analyses. Results are only occasionally downloadable and formats are not customizable. No claims are made as to the future longevity of the services. All of these shortcomings are probably due to cost concerns (especially in terms of bandwidth), since none of these projects are profiting financially from hosting email archives. However, this means that for any type of analysis more complicated than a conducting a keyword search or generating a graph of post counts, researchers will have to download their own email and build a set of tools for storing and analyzing it.

B. *Limitations of Home-Grown Email Archives for Replication and Sharing*

The second way that researchers interact with email archives is to download the messages, clean and store them in a local database or file system, and then analyze them. The processes and software used for conducting these activities, even the collecting and cleaning activities, will vary widely between teams and will be difficult to describe in sufficient detail to be replicable by another researcher.

To illustrate the point, here are the "how the data was collected" descriptions from three different research papers, each studying mailing lists in different ways, but each starting with the same type of raw data (mbox mailing list archive files).

Example 1: "We process each message in a mailing list repository using a semi-automated approach similar to [ref]. We remove attachments, duplicate messages, convert HTML emails to plain text, and extract the email header information (such as From and Date). Then, we identify and merge multiple email addresses that belong to the same person." [9]

Example 2: "A Python script implementing the schema was used to extract data from the mbox files. For a given input list, the script traversed each mbox to extract a record for each msg_id (primary key). The output for each run was parsed into a MySQL database containing two tables, one for each list. SQL queries were used to extract the information necessary for data analysis.... The data cleaning process involved removing messages in the following categories...." [10]

Example 3: "For every email, we extracted from the email header the message identifier, the sender, the sent time, and the identifier of the message (if any) to which this message was a reply. When a reply-to header was found, the sender s of the reply was someone who found the initial message of interest; and so the sender s was marked as a recipient of the original message.... A small proportion of messages could not be parsed, because of malformed headers. Approximately 1.3% of the messages were in this category." [11]

When faced with the requirement to design and construct a system for collecting and storing mailing list messages, it is natural that each research team will approach this differently. These differences are due to familiarity with various design paradigms, languages and technologies as well as the requirements coming from particular research goals. For example, in Example 3, the researchers only kept the headers they needed and were very concerned with finding threads and replies. Compare this to Example 1, where the researchers kept the entire message but focused on removing duplicate addresses. In many papers, there is insufficient information describing the cleaning and storage, which makes it nearly impossible to recreate the data set (much less to replicate the entire study). In other cases (e.g. [6]), details and rationale are given, but code is not. These differences may be due to page limitations, different levels of expertise and familiarity, and different expectations of the particular research community.

The methods chosen for cleaning and storage are not the only area of concern with transparency and replicability; researchers' standards for quality of the results will also vary widely, and can be affected by choice of technique. Bettenberg, *et al.* [12] discuss the likelihood of potential failure in processing email archives using known information retrieval techniques (many of which were not developed for use with email), and the risks to quality from each step taken to collect, store, and clean the data. For example, they state that even the simple foundational activity of extracting messages from mbox archives still has a measure of risk (albeit a low one), in that there have been differing mbox specifications over time. They acknowledge that other email cleaning tasks, such as duplicate identification and threading, are considerably more risky and difficult to standardize. They conclude with a recommendation that the research community work to develop standards for mailing list processing techniques.

C. *Goal of this Paper*

This paper is a step towards that goal of a standardized, sharable, reproducible infrastructure. The next section describes a system designed to standardize some of the choices about how emails are processed, thus lowering the barrier to entry and providing transparency to research groups studying email. The scripts and techniques also improve replicability by providing a common language for communicating about how the messages were collected and stored.

III. The FLOSSmole Apachemail Platform

The system described here is named *Apachemail* (as it has been deployed and tested using ASF emails) although it will work on any collection of email in mbox format, including personal email. The entire system (code and procedures) has been donated to the FLOSSmole data commons [13]. The system has a few key characteristics that are intended to improve the logistics of sharing between researchers, while decreasing the tedious collection and cleaning work.

A. *Requirements*

First, because the data is collected from a third party, and is growing all the time, the system should support regular, incremental additions in the native format with minimal or no adjustments. Second, because the email artifacts have both structured and unstructured parts (headers can be conceived as structured data, whereas content is usually unstructured text), the data storage architecture is designed to be as flexible as possible in terms of new or unknown headers, while also being entirely searchable within the content. Third, this system can be deployed in a central, accessible Internet location for use by many distributed researchers. Finally, the system supports a way for users to query the system via JavaScript views (discussed in Section D), and to describe those queries to each other, thus supporting sharing.

B. *Collection*

As discussed in Section I, many FLOSS software projects provide the archives of their email mailing lists in either a browsable interface, or in downloadable files. This system was deployed with ASF mailing list files, which are available online, archived by project (e.g. Apache Httpd Server) and list name (e.g. 'dev' for developers), then by year and month (e.g. '199502').

The archives are stored in compressed UNIX mbox format [14]. Mbox files are just text files of email messages with all headers, content, and attachments intact. Each message starts with a From: header, followed by any number of other headers set by the email client. Each message is separated from the next message by a blank line. The mailing list that the message was sent to is shown in the List-Id: header (for newer messages), or in the Mailing-List: header (for older messages).

The larger, older ASF projects have raw mbox files easily accessible for downloading. (Some newer, smaller projects appear in the browsable interface [15] but the location of the raw mbox files is less obvious. If the approach from this paper proves to be popular, the next step will be to expand the collection by seeking out those hard-to-find raw mbox files or by writing additional collection procedures.) For the purposes of this paper, a few simple shell scripts were written to download raw mbox files from Apache Httpd Server developers list (called *httpd-dev* for the rest of this paper). It had 131,093 messages over the 18 years from 1995-2013. The scripts to download these mbox files are available in the Github repository for this project [16].

C. *Cleaning and Storage*

After downloading and uncompressing the mbox files, the mailbox files were transformed into an array of JSON objects using a short Python script based on work from [17]. This process included stripping out HTML tags and working with various character encodings and quoted printable strings. The scripts for this procedure are available on Github.

The JSON files were then read into a CouchDB [18] nosql document-oriented database server using another Python script (also available on Github). The document-oriented approach is preferable to a relational database for a few reasons. First, there are more than 100 different headers used in 131,000 emails, which means the number of fixed columns that a relational structure would have to have would be quite large, and many of the columns would be sparsely populated. Second, to create a relational schema, it is necessary to know all the possible columns in advance, before reading in the data. This obviously limits flexibility in dealing with new, never-before-seen headers. (Similar systems using a relational format, for example MLStats from MetricsGrimoire[19], solve this problem by only saving a few of the most common and important headers.) However, with a nosql system, every header can be handled as a "key" with an associated "value", and there is no need to know the entire list of keys in advance. Second, although there are numerous known advantages of a relational system, especially in transactions and security, these are less important in an email storage scenario since this is static, public data. In addition, the data set will only be added to, and not deleted from. Finally, the CouchDB system provides a REST-oriented API for access, and multi-master replication (pull or push) [20], both of which were important requirements for this system.

D. *Querying*

After the emails were read into in the database, a few simple JavaScript "views" were created to summarize key features of the data, similarly to the basic counts provided by the other email browsing systems mentioned previously (e.g. MarkMail). This section walks through some options for mailing list message retrieval, summary, display, and interaction in this system.

A simple example of a mailing list query is to count the messages sent to the list over time. This type of query is available on many of the web-based email archives compared in section II. This is accomplished using CouchDB views. CouchDB views are written in JavaScript (or another language if a query server is available and installed), and saved inside CouchDB as design documents. Views use the map/reduce paradigm to emit some subset of keys and values that match the specified map (and optional reduce).

For this project, a simple example was to map the email messages by year, month, and day, and then reduce the results to counts. Fig. 1 shows the result of a view called countByDate. This is the result as shown in the Futon web application administrative interface that comes with CouchDB. For this view, the grouping level was set to 1 (to group by "year"). Grouping level 2 is year-month, and grouping level 3 is year-month-day. For those familiar with the RDBMS paradigm, a map/reduce like this is similar to doing a GROUP BY and COUNT() in SQL. The countByDate view code is too long to reproduce in this paper, but it is available on the github site for this project [16].

Now that this view is written, it is possible to add a CouchDB "list" to format the results for viewing in a browser or other application via a specially-formatted URL. The list can emit HTML along with the results of the view.

Fig. 2 shows the code used to create a list called countByDate. It iterates through each item emitted by the view, and sends it to the browser wrapped in HTML with added simple bullet list (<ol>, <li>) tags and a colon (":") between the year and count (line 9).

The URL to access this CouchDB list will include the name of the design document view, as well as the name of the list itself. It can also contain optional query parameters, such as grouping level or start/end points. The URL to access the countByDate list is shown below:

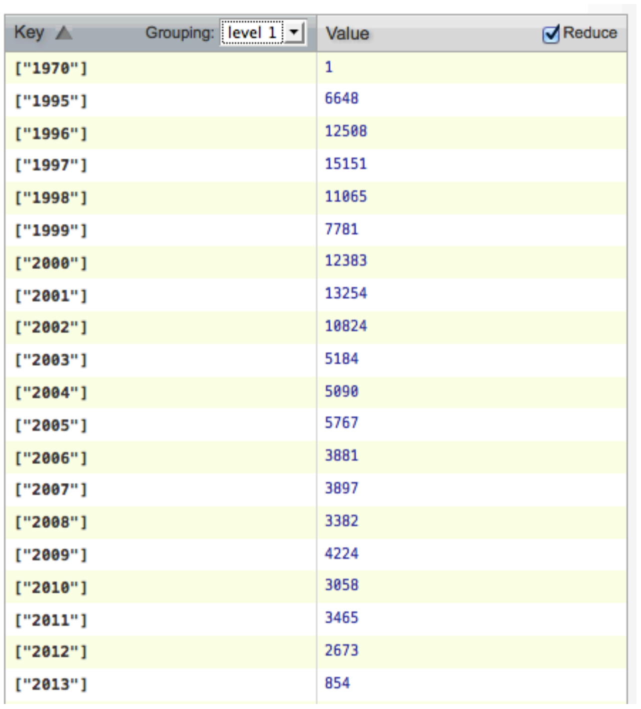
http://hostname:5984/apachemail/_design/countByDate/_list/countByDate/countByDate?group=true&startkey=[%221994%22]&endkey=[%222014%22]&group_level=1

The URL begins with hostname and port number for CouchDB instance, and is followed by the name of the view and list. Optional query parameters are given at the end ("group", "startkey", "endkey", and "group_level"). Note that the URL must be encoded with special characters to represent double quotes (%22). The name of the design document view and list are requested explicitly ("countByDate"), since there will be many views in the system. In this example, group_level of 1 means "year" (2 is month and year, 3 is day, month, and year), so the startkey (1994) and endkey (2014) are the bookends for the year values we wish to show.

Fig. 3 shows what a simple bullet list will look like when accessed in the browser using the URL above. URLs can be shared and manipulated easily by multiple users.

The next logical step is to attach that URL to a web form, and use HTTP GET method to submit user input dynamically. For example the user to could be given a web form to adjust the startkey and stopkey parameters or the grouping level. Fig. 4 shows the sample form. The results look identical to Fig. 3, except that only the years selected by the user will be shown. (The group parameter is passed as a hidden variable.)

Finally, a JavaScript graphing library, such as the Google Charts API [21], can be added to the list design document to display the result as a chart or graph.

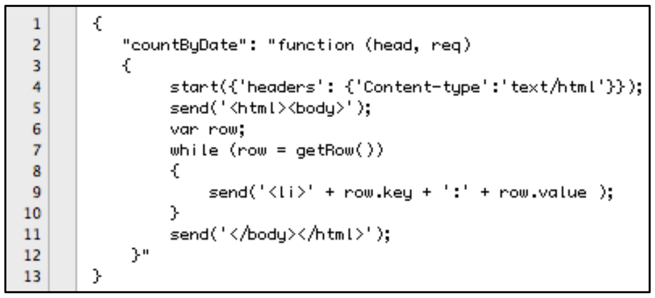


Key	Value
["1970"]	1
["1995"]	6648
["1996"]	12508
["1997"]	15151
["1998"]	11065
["1999"]	7781
["2000"]	12383
["2001"]	13254
["2002"]	10824
["2003"]	5184
["2004"]	5090
["2005"]	5767
["2006"]	3881
["2007"]	3897
["2008"]	3382
["2009"]	4224
["2010"]	3058
["2011"]	3465
["2012"]	2673
["2013"]	854

Fig. 1. The countByDate view, reduced, with grouping set to Level 1

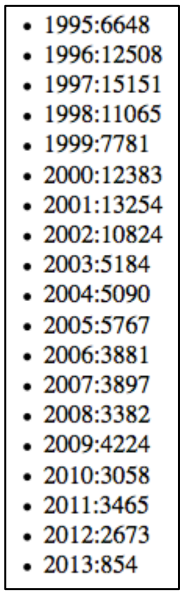
Fig. 5 shows the code added to the list to push the results through the Google Charts API before displaying in the browser. Fig. 6 shows the result as a line graph. (Google Charts API also provides many other visualization types; the JavaScript in Fig. 5 can be changed to emit column charts, pie charts, etc.)

All of the code for creating the views and lists described in this section has been loaded into the same Github repository. [16] All of the scripts and procedures shown can be used on any mailing list, not just the ASF lists used here. Headers that are not of interest can be ignored. New views and lists can be written and shared with others.



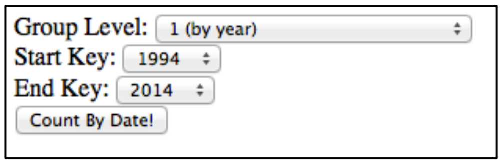
```
{
   "countByDate": "function (head, req)
   {
        start({'headers': {'Content-type':'text/html'}});
        send('<html><body>');
        var row;
        while (row = getRow())
        {
            send('<li>' + row.key + ':' + row.value );
        }
        send('</body></html>');
     }"
}
```

Fig. 2. The countByDate list JavaScript code



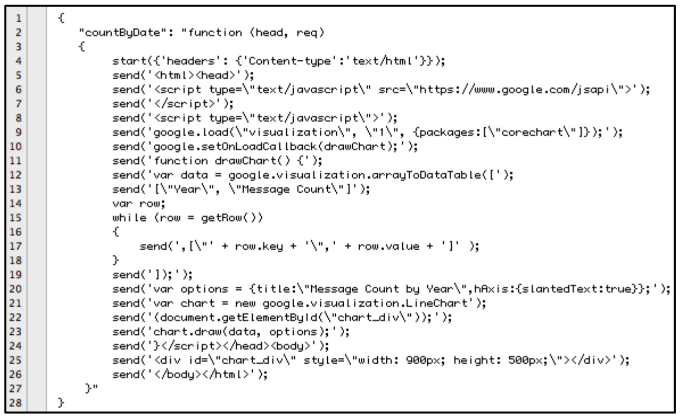
- 1995:6648
- 1996:12508
- 1997:15151
- 1998:11065
- 1999:7781
- 2000:12383
- 2001:13254
- 2002:10824
- 2003:5184
- 2004:5090
- 2005:5767
- 2006:3881
- 2007:3897
- 2008:3382
- 2009:4224
- 2010:3058
- 2011:3465
- 2012:2673
- 2013:854

Fig. 3. The countByDate list, emitted as HTML with bullets (<ol>, <li>)



Group Level: 1 (by year)
Start Key: 1994
End Key: 2014

Fig. 4. Web form for changing parameters in countByDate URL dynamically



```
{
   "countByDate": "function (head, req)
   {
        start({'headers': {'Content-type':'text/html'}});
        send('<html><head>');
        send('<script type=\"text/javascript\" src=\"https://www.google.com/jsapi\">');
        send('</script>');
        send('<script type=\"text/javascript\">');
        send('google.load(\"visualization\", \"1\", {packages:[\"corechart\"]});');
        send('google.setOnLoadCallback(drawChart);');
        send('function drawChart() {');
        send('var data = google.visualization.arrayToDataTable([');
        send('[\"Year\", \"Message Count\"]');
        var row;
        while (row = getRow())
        {
            send(',[\"' + row.key + '\",' + row.value + ']' );
        }
        send(']);');
        send('var options = {title:\"Message Count by Year\",hAxis:{slantedText:true}};');
        send('var chart = new google.visualization.LineChart');
        send('(document.getElementById(\"chart_div\"));');
        send('chart.draw(data, options);');
        send('}</script></head><body>');
        send('<div id=\"chart_div\" style=\"width: 900px; height: 500px;\"></div>');
        send('</body></html>');
    }"
}
```

Fig. 5. Code for countByDate list, emitted as Google Charts API line graph

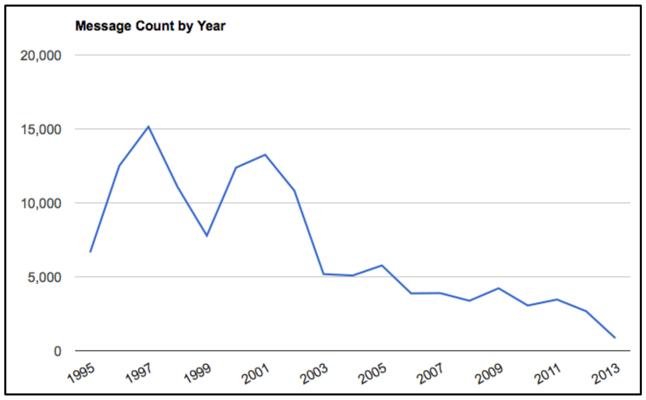
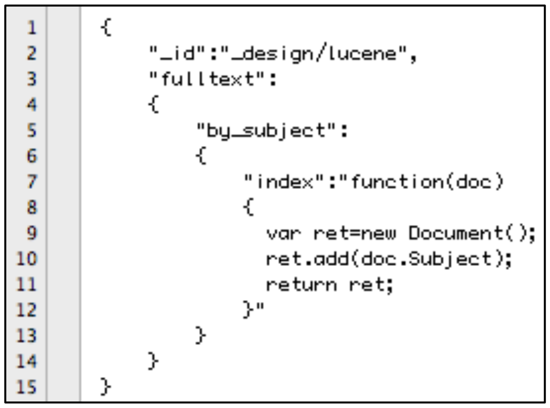
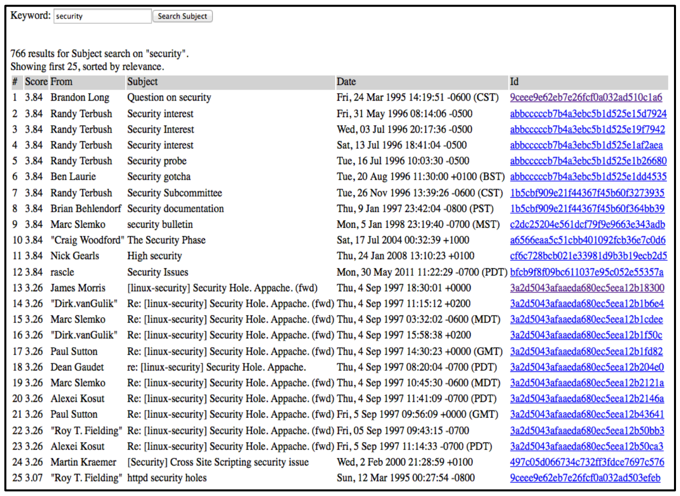


Fig. 6. The countByDate list, emitted as Google Charts API line graph



```
{
    "_id":"_design/lucene",
    "fulltext":
    {
        "by_subject":
        {
            "index":"function(doc)
            {
                var ret=new Document();
                ret.add(doc.Subject);
                return ret;
            }"
        }
    }
}
```

Fig. 7. Design document for a Lucene fulltext index of message subject lines



Keyword: security Search Subject

766 results for Subject search on "security".
Showing first 25, sorted by relevance.

#	Score	From	Subject	Date	Id
1	3.84	Brandon Long	Question on security	Fri, 24 Mar 1995 14:19:51 -0600 (CST)	9ceee9e62eb7e26fcf0a032ad510c1a6
2	3.84	Randy Terbush	Security interest	Fri, 31 May 1996 08:14:06 -0500	abbccccb7b4a3ebc5b1d525e15d7924
3	3.84	Randy Terbush	Security Interest	Wed, 03 Jul 1996 20:17:36 -0500	abbccccb7b4a3ebc5b1d525e19f7942
4	3.84	Randy Terbush	Security interest	Sat, 13 Jul 1996 18:41:04 -0500	abbccccb7b4a3ebc5b1d525e1af2aea
5	3.84	Randy Terbush	Security probe	Tue, 16 Jul 1996 10:03:30 -0500	abbccccb7b4a3ebc5b1d525e1b26680
6	3.84	Ben Laurie	Security gotcha	Tue, 20 Aug 1996 11:30:00 +0100 (BST)	abbccccb7b4a3ebc5b1d525e1dd4535
7	3.84	Randy Terbush	Security Subcommittee	Tue, 26 Nov 1996 13:39:26 -0600 (CST)	1b5cbf909e21f44367f45b60f3273935
8	3.84	Brian Behlendorf	Security documentation	Thu, 9 Jan 1997 23:42:04 -0800 (PST)	1b5cbf909e21f44367f45b60f364bb39
9	3.84	Marc Slemko	security bulletin	Mon, 5 Jan 1998 23:19:40 -0700 (MST)	c2dc25204e561dcf79f9e9663e343adb
10	3.84	"Craig Woodford"	The Security Phase	Sat, 17 Jul 2004 00:32:39 +1000	a6566eaa5c51cbb401092fcb36e7c0d6
11	3.84	Nick Gearls	High security	Thu, 24 Jan 2008 13:10:23 +0100	cf6c728bcb021e33981d9b3b19ecb2d5
12	3.84	rascle	Security Issues	Mon, 30 May 2011 11:22:29 -0700 (PDT)	bfcb9f8f09bc611037e95c052e55357a
13	3.26	James Morris	[linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 18:30:01 +0000	3a2d5043afaaeda680ec5eea12b18300
14	3.26	"Dirk.vanGulik"	Re: [linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 11:15:12 +0200	3a2d5043afaaeda680ec5eea12b1b6e4
15	3.26	Marc Slemko	Re: [linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 03:32:02 -0600 (MDT)	3a2d5043afaaeda680ec5eea12b1cdee
16	3.26	"Dirk.vanGulik"	Re: [linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 15:58:38 +0200	3a2d5043afaaeda680ec5eea12b1f50c
17	3.26	Paul Sutton	Re: [linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 14:30:23 +0000 (GMT)	3a2d5043afaaeda680ec5eea12b1fd82
18	3.26	Dean Gaudet	re: [linux-security] Security Hole. Appache.	Thu, 4 Sep 1997 08:20:04 -0700 (PDT)	3a2d5043afaaeda680ec5eea12b204e0
19	3.26	Marc Slemko	Re: [linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 10:45:30 -0600 (MDT)	3a2d5043afaaeda680ec5eea12b2121a
20	3.26	Alexei Kosut	Re: [linux-security] Security Hole. Appache. (fwd)	Thu, 4 Sep 1997 11:41:09 -0700 (PDT)	3a2d5043afaaeda680ec5eea12b2146a
21	3.26	Paul Sutton	Re: [linux-security] Security Hole. Appache. (fwd)	Fri, 5 Sep 1997 09:56:09 +0000 (GMT)	3a2d5043afaaeda680ec5eea12b43641
22	3.26	"Roy T. Fielding"	Re: [linux-security] Security Hole. Appache. (fwd)	Fri, 05 Sep 1997 09:43:15 -0700	3a2d5043afaaeda680ec5eea12b50bb3
23	3.26	Alexei Kosut	Re: [linux-security] Security Hole. Appache. (fwd)	Fri, 5 Sep 1997 11:14:33 -0700 (PDT)	3a2d5043afaaeda680ec5eea12b50ca3
24	3.26	Martin Kraemer	[Security] Cross Site Scripting security issue	Wed, 2 Feb 2000 21:28:59 +0100	497c05d066734c732ff3fdce7697c576
25	3.07	"Roy T. Fielding"	httpd security holes	Sun, 12 Mar 1995 00:27:54 -0800	9ceee9e62eb7e26fcf0a032ad503efeb

Fig. 8. Results of a keyword search for "security" on message subject lines

E. Searching

One of the key features of web-based public email collections such as MarkMail is the keyword search facility. For this project, the Lucene search engine [22] was configured to work with CouchDB and perform full-text indexing of mailing list message content and subject lines. Lucene is a set of java-based search engine libraries that perform fast, full-text indexing.

After installing Lucene and configuring it to be aware of the CouchDB database, a design document was created to create fulltext indexes on content and subject lines. The subject line document is shown in Fig. 7.

A simple PHP script then provides a keyword search box, issues a request to the database in the form of a URL, and displays the results in a table. Fig. 8 shows an example of the completed Search application. In this example, the Subject lines of the httpd-dev mailing list were searched for the keyword "security", and the results are shown in order by their Lucene score (relevance).

Fig. 8 shows that the document ID is clickable (last column). When clicked, the link will bring up the actual email message. All the CouchDB design documents and the PHP code used to generate this web interface are also available [16].

IV. CHALLENGES AND LIMITATIONS

This paper presents a reusable system for standardizing mailing list collection, storage, and cleaning. The approach presents a few challenges and limitations, which can be considered as opportunities for future work.

A. Storing many lists

The prototype system developed here was written for commodity hardware on a single server. The focus of the work was in collecting and storing ASF mailing lists, particularly the httpd-dev list. As more lists are added to the database, a few problems present themselves: first, organization, then size, and subsequently, cost. With multiple lists, it becomes necessary to have some way of differentiating between the lists so that only the documents (emails) from the correct list are considered in a given query. For example, as described in section III.B., it is usually possible to determine the list by using the List-Id header (if available). Queries, in the form of views, will either have to be written to use this information, or separate databases will need to be kept for each list. If cross-list comparisons are required, then the latter solution is not a good option. Even when the lists are able to be differentiated using a header, the size of the database could grow substantially. The 18 years of the httpd-dev list takes about 310 MB inside CouchDB. But for the ASF alone, there are over 200 projects, each of which have between two and 38 lists. Storing all of these lists into perpetuity could create a requirement for substantial disk space. However, this would be a concern for *any* system of this size and is not affected by the choice of a document-oriented database.

B. *Querying via views*

CouchDB is unlike the traditional RDBMS that many data analysts may be more familiar with, in that it does not allow *ad hoc* querying. Views must be written in advance, and they are actually stored as documents inside the document database itself. This has one drawback, in that it is a new paradigm for query development, and many users of a system like this will be unfamiliar with it. One benefit, however, is that it encourages sharing of views between researchers. Views can be published as documents in the server, and since the server can be mirrored across multiple sites easily (this is in fact one of the benefits of CouchDB, as explained in section III.C.), the result is a highly sharable, replicable data and query store.

C. *De-duplication of addresses*

Bettenberg, *et al.* [12] pointed out the numerous difficulties in cleaning email headers. One cleaning task that is particularly important for many email mining analyses, such as creating social networks or reading dialogue from users, is to standardize the email address formats, and figure out which addresses are aliases. Indeed, for this project, a very simplistic view called CountBySender (similar in structure to the CountByDate examples given in Section III) suffered from this problem. In 18 years of the httpd-dev mailing list, people are going to have multiple email addresses, or multiple spellings for their name, added middle initials, etc. Left uncleaned, these inconsistencies have a detrimental effect on the quality of the view results. For example multiple email addresses will dilute the count impact of a single sender. In other words, if their name is spelled multiple ways it will reduce the count of emails any one of those "senders" sent. Or, for social network creation, name duplication will result in an incorrectly drawn network with more nodes than should exist. This is a problem which will require cleaning steps at the view level to solve. Similarly, the Date headers of emails had to be cleaned and standardized in the CountByDate views to account for pre-Y2K era two-digit years. Once the views are written, they can be shared and improved in the usual way.

D. *Threading*

Another problem with the prototype system as presented here is that it currently lacks threading for emails. When one email of interest is found, it is sometimes helpful to see the entire thread before and after that message. Right now this system does not specifically support threading via views or in the web interface. Threading can be more easily accomplished with newer headers that were added to modern email clients to handle this issue. However, because many of the researchers using this system are likely to be studying many years' worth of email, they are likely to need a more comprehensive and high-quality way to reconstruct threads for older messages, or for when this header is missing. In the future, threading can be added by creating views in CouchDB following the model presented in [17], for example. Subsequently, it will be possible to perform a keyword search, like the one shown in Fig. 8, but to have the option of showing the original messages *and* allow a user to retrieve the entire thread context.

V. CONCLUSION

This paper presents a replicable system, FLOSSmole Apachemail, for collecting and storing email message archives in mbox format in a document-oriented database, for creating views and lists as queries to those messages, and for setting up a search engine and web-based interface to that search engine. This system could be useful to researchers in a variety of fields (including but not limited to empirical software engineering) who need to get a large amount of email collected and stored quickly and easily, and want to share the collection with others. The paper reviews several challenges and limitations of the project, and presents a way forward on each of these.

The system described here improves replicability of email archive-based software engineering research in two ways: first, by standardizing some of the choices about how emails are processed, this system both removes variability between different research groups and lowers the barrier to entry for new research groups. Second, using a common system will provide a familiar and standardized vocabulary for communicating about how the email messages were collected and stored. This improves clarity of communication between research groups.

To get started using the FLOSSmole Apachemail system, and to encourage sharing and replication of this infrastructure, the entire codebase for this system is available for download, on a publicly-available version control system, where changes can be made by anyone and can be shared with others. Of course, this also means that this system could be rolled out in a central location for multiple groups to access and share.

REFERENCES

[1] Apache Software Foundation. "Project Management Committee Policies". http://www.apache.org/dev/pmc.html#mailing-list-naming-policy

[2] M. Squire. "How the FLOSS research community uses email archives." Int. J. Open Source Software Systems & Proeesses, 4(1). 2012. pp. 37-59.

[3] D.S. Pattison, C.A. Bird, and P.T. Devanbu. "Talk and work: a preliminary report." In Proc. 5th Int. Working Conf. on Mining Software Repositories (MSR). 2008. 113-116.

[4] J. Roberts, I.-H. Hann and S. Slaughter. "Communication networks in an open source project". In Proc. 2nd Int. Conf. on Open Source Systems (OSS). 2006. 297- 306.

[5] P.C. Rigby, and A.E. Hassan. "What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List." In Proc. 4th Int. Workshop on Mining Software Repositories (MSR). 2007. 23-31.

[6] A.C. MacLean, L.J. Pratt, C.D. Knutson and E.K. Ringger. "Knowledge homogeneity and specialization in the Apache HTTP Server project." In Proc. of the 7th Int. Conf. on Open Source Systems (OSS). 2011. 106-122.

[7] A. Schilling, S.L. Andreas, and T. Weitzel. "Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in floss projects." In Proc. 45th Hawaii Int. Conf. on System Science (HICSS). 2012. pp. 3446-3455.

[8] MarkMail. "Frequently Asked Questions". http://markmail.org/docs/faq.xqy

[9] W.M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A.E. Hassan. "Should I contribute to this discussion?" In Proc. 7th Int. Working Conf. Mining Software Repositories (MSR). 2010. pp. 181-190.

[10] S.K. Sowe, I. Stamelos, and L. Angelis. "Understanding knowledge sharing activities in free/open source software projects: An empirical study." Journal of Systems and Software, 81(3), 431-446.

[11] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. "Mining email social networks". In Proc. 3rd Int. Workshop Mining Software Repositories (MSR). 2006. pp. 137-143.

[12] N. Bettenburg, E. Shihab, and A. E. Hassan, "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data," In Proc. 25th IEEE Int. Conf. on Software Maintenance (ICSM), 2009, pp. 539–542.

[13] J. Howison, M. Conklin, and K. Crowston,. "FLOSSmole: A collaborative repository for FLOSS research data and analyses". Int. J. Information Technology and Web Engineering, 1(3). 2006. 17–26.

[14] E. Hall. "RFC 4155, The application/mbox Media Type". September 2005. http://tools.ietf.org/html/rfc4155

[15] Apache Software Foundation. "Available Mailing Lists". http://mail-archives.apache.org/mod_mbox/

[16] FLOSSmole Apachemail Github Repository. https://github.com/megansquire/apachemail

[17] M.A. Russell. Mining the Social Web. Sebastopol, CA, USA: O'Reilly. 2011.

[18] Apache CouchDB. http://couchdb.apache.org

[19] MetricsGrimoire. "Mailing List Stats". http://metricsgrimoire.github.io/MailingListStats/

[20] Apache CouchDB Wiki. "How to replicate a database". http://wiki.apache.org/couchdb/How_to_replicate_a_database

[21] Google Charts API. https://developers.google.com/chart/

[22] Apache Lucene. http://lucene.apache.org