Electronic Mail Database System

Christopher League

CS437b final report
15 May 2000

1. Introduction

Electronic mail systems typically have a concept of a *folder* or *mailbox*, which is simply a collection of messages. New messages appear in a specially-designated folder, often called the *inbox*. From there, a user might manually move certain messages into other folders, based on their content, sender, priority, or the current phase of the moon. Alternatively, a sophisticated user might invoke an auto-sorter (such as procmail or slocal [8, 9]), which will automatically file new messages in particular folders, based on their content, sender, priority, or the current phase of the moon. In this case, the user also needs a tool, such as flist [8], to find the set of folders that contain *unseen* messages.

Folders are generally implemented in one of two ways. First, a folder might be a directory in the underlying file system, with each message in a separate file. This is used by MH and its front-ends, including xmh, exmh, mh-e, and mew [8]. Second, a folder might be a single file, with each message concatenated using some designated separator. The Unix mail program, Netscape Messenger, and many others use *mbox* format, where a specially-formatted "From" line serves as the separator. Another instance, called *MMDF,* introduces a sequence of control characters as the separator.

Each implementation has its disadvantages. An mbox with thousands of messages can cause an unacceptable latency when the system needs to display a summary or delete a message. Likewise, building a summary for an MH directory with thousands of messages requires at least that many system calls and disk accesses. (Not to mention that directories with so many files eat up *inodes* and make your system administrator unhappy!) With either format, clients invariably must create a summary or cache file for each folder, to help overcome these latencies.

Apart from these implementation concerns, the model itself has important limitations; namely,

- a message can belong to only one folder at a time, and
- search, select, and mark operations are always performed within a single folder.

I have several conventions for assigning methods to folders: by the sender/recipient, by mailing list, by subject matter. None of them works perfectly. Suppose Stefan sends a

message to our research group, via the `flint-group` mailing list. I (or my auto-sorter) might place it in the `flint` folder. Now suppose that Zhong replies to the message, but not to the whole group; he replies to Stefan and me only. My auto-sorter would place the reply in the `zhong` folder.

Then, perhaps Valery replies to `flint-group` with some information that might be useful for a later phase of my dissertation research. My auto-sorter would put it in `flint`; I might move it to either `valery` or `research`. A year from now when I need to find it, will I know where to direct my mail client? No, probably I will use `grep` instead! All these messages logically belong to the same thread, but with the way they have been scattered, MH is unlikely ever to show them together again.

My intuition is that an email system which can overcome these problems with the folder model will essentially contain (or at least emulate) a relational database system. This project is an experiment to determine the feasibility of storing email in an RDBMS.

The intended system architecture is illustrated in figure **??**; in many ways, it is similar to that of IMAP. Various kinds of clients connect to a server, which stores the messages. Unfortunately, the folder model is alive and well on the IMAP server. My idea is to design a more flexible protocol, which will free us from the limitations of the folder model. The server itself (tentative name: *Meba*, for Message Base) is implemented using an SQL server, which probably (but not necessarily) will run on the same machine.

In this report, I will describe an initial data model (§2), its implementation (§3), and some performance results (§3.4).

2. Data model

The primary *entity* in my data model is, naturally, a message. A message has a sequence of *headers* (key-value pairs) and a body, which, for now, is just a string of text. The MIME standard defines a structure on the body, so it can contain a hierarchy of various types of data; this is a possible extension, but it would complicate the data model.

Headers that are relevant for searching include:

- the addresses (`From`, `To`, `Cc`),
- `Date`, `Subject`, `Message-Id`,
- `In-Reply-To`, and `References`.

The address fields are multi-valued (except for `From`) and can have pairs of real name and email address. There are several different formats, but they are all easy to recognize and parse.

The `Message-Id` and related fields require some explanation. Sendmail and other MTAs generate this identifier before releasing the message onto the network. Intended to be globally unique, the `Message-Id` typically incorporates the hostname, date and time, and perhaps some random bits (*e.g.*, ⟨`115CF8C60D8ED2118DF00008C75D4456011B3471@cfnyexch02.ny.cantor.com`⟩). The `Message-Id` is probably not reliable enough to use as a primary key;

we assign arbitrary integers for that purpose. It can, however, be very useful for linking together related messages.

Specifically, when a recipient replies to my message, her mailer will insert an `In-Reply-To` header, which may contain my message's `Message-Id`. In this way, we can determine which messages refer to which other messages. Unfortunately, the contents of `In-Reply-To` are not standardized, and vary widely from one mailer to the next. The `References` header is similar, but came from the USENET world [5]. Some new proposals ask mailers to use `References` instead of `In-Reply-To`. When it exists, the format of this field is more reliable.

Finally, for each message, there is a fair amount of user- or client-generated markup:

- flags indicating whether the message has been read, replied to, forwarded, etc.,
- marks (sequences in MH) indicating that certain messages are slated for further processing, and
- names of folders to which the message belongs.

We introduce the notion of a *tag* to serve all these purposes. A tag is simply a string; each message has a *set* of tags associated with it. Certain tags can be designated (by clients, or by the protocol) as meaning *unseen, unprocessed, high-priority,* and so on. User-designated tags stand for sequences or (virtual) folders. This way, a message can easily belong to multiple folders.

Let us summarize this section by presenting the E-R diagram in figure **??**. *message* is an entity set with primary key *message_num* and simple attributes *subject*, *date*, *message_id*, *headers*, and *body*. (*headers* contains the complete header part of the original message.) *from*, *to*, and *cc* are composite attributes which feature the email address and full name as their components. *to*, *cc*, *tag*, and *reference* are multi-valued attributes. (*reference* includes any message IDs we could find in either the `In-Reply-To` or `References` headers.)

There is a peculiar kind of relationship set in this model: a message is related to many other messages through the *reference* attribute. It is unclear how to represent this relationship in the E-R diagram, however, because *reference* relates to other messages by *message_id*, which is *not* a candidate key. Indeed, it is acceptable to have "dangling pointers" in *reference* and possible to have duplicate or empty *message_id*s in the entity set. Nevertheless, as we will see in section 3.2.2, we can self-join on the *reference* attribute to find a message's referents (parents) or follow-ups (children).

In the next section, we explore the implementation of this model, from the SQL code to create and access it, to the Perl code for parsing message headers.

3. Implementation

The first step in implementing this model is to create the SQL tables. As described in [11, page 56], new tables must be created for multi-valued attributes. In addition to the `message` table, we will create tables called `reference`, `tag`, and `recipient`. `recipient` combines the *to* and *cc* attributes by using an additional `kind` field.

message

Field	Type	Key
message_num	int(10) unsigned	Pri
from_addr	varchar(200)	Mul
from_name	varchar(200)	
subject	varchar(200)	Mul
date	datetime	Mul
message_id	varchar(200)	Mul
headers	text	
body	mediumtext	

recipient

Field	Type	Key
message_num	int(10) unsigned	Pri
kind	enum('To','Cc')	Pri
address	varchar(200)	Pri
name	varchar(200)	

reference

Field	Type	Key
message_num	int(10) unsigned	Pri
message_id	varchar(200)	Pri

tag

Field	Type	Key
message_num	int(10) unsigned	Pri
tag	varchar(200)	Pri

Figure 1: Descriptions of the SQL tables created for the data model. We currently allow 64k for the headers and 16M for the message body.

We create indices for any fields on which we might search: `subject`, `date`, `message_id`, and all the addresses. MySQL will use indices for both exact and prefix matches. Descriptions of all the tables are given in figure 1.

I have written a Perl script, setup, to create these tables, given the name of a database, hostname, username, and so on. The script does not change any database permissions or create a new database. Table names can optionally be prefixed by some string, allowing multiple instances of the message tables to coexist in one database. The script writes several configuration parameters to a file, which is then read by the other scripts in the suite. A companion script, destroy, drops all the tables created by setup.

3.1. Insertion

The insert script reads a single message from standard input, or from files named on the command line. It is meant to be invoked by sendmail whenever a new message arrives for a particular user, but it can also easily insert the files (messages) from an MH tree. If something goes wrong with the insertion (*e.g.*, the script cannot connect to the database), then sendmail can be instructed to requeue the message and try again later.

Message parsing is largely performed by Graham Barr's MailTools for Perl [2]. This package will separate the headers from the body, and it can parse the contents of certain headers, including name and address lists (in `From`, `To`, and `Cc`) and the date, including time zone (*e.g.*, `Tue, 22 Feb 2000 16:11:35 -0500`).

Barr's package does not understand `References` or `In-Reply-To`, so I had to write parsers for those. `References` is standardized in the USENET specification [5]. If present, it contains a string of angle-bracketed message IDs, separated by white space:

```
References: <ucg0udwc8v.fsf@kahlua.cs.yale.edu> <38BADF98.AA2AD9EA@yale.edu> ...
```

`In-Reply-To`, on the other hand, is not standardized at all. Jamie Zawinski, a lead developer for Netscape Messenger, surveyed a collection of 22,950 messages with `In-Reply-To`

Return-Path: ⟨league@scire.pair.com⟩
Received: from scire.pair.com ...
Received: from crufty.src.bell-labs.com ...
X-Envelope-To: ⟨league@contrapunctus.net⟩
Message-Id: ⟨204041.LA21@nsl.cs.bell-labs.com⟩
From: "John H. R." ⟨jhr@something.com⟩
To: Dave M. ⟨dbm@something.com⟩
Cc: Edjard ⟨edjard@something.br⟩,
sml-nj@something.com
Subject: Re: documentation
In-Reply-To: Your message of "Tue, 04 Apr 2000"
⟨20046.AA10@nsl.cs.bell-labs.com⟩
Date: Tue, 04 Apr 2000 11:51:05 -0400

I'm in the process of producing new documentation using the ML-Doc tool and the first HTML version of this should be on the web later this week. Right now, it is mostly just the signatures, but I'll be adding descriptions as timer permits.

– John

```
INSERT INTO message VALUES (
  "jhr@something.com",
  "John H. R.",
  "Re:  documentation",
  "2000-04- 4 11:51:05",
  "204041.LA21@nsl.cs.bell-labs.com",
  "Return-Path:  <league@scire...",
  "I'm in the process...");
⟹ 574
INSERT INTO recipient VALUES (
  574, "To",
  "dbm@something.com",
  "Dave M.");
INSERT INTO recipient VALUES (
  574, "Cc",
  "sml-nj@something.com", "");
INSERT INTO recipient VALUES (
  574, "Cc",
  "edjard@something.br",
  "Edjard");
INSERT INTO reference VALUES (
  574,
  "20046.AA10@nsl.cs.bell-labs.com");
INSERT INTO tag VALUES (
  574, "unseen");
```

Figure 2: A sample message, and the SQL code used to insert that message into the database.

headers, and found that "the most common forms of `In-Reply-To` seem to be:

31%	`NAME's message of TIME <ID@HOST>`
22%	`<ID@HOST>`
9%	`<ID@HOST> from NAME at "TIME"`
8%	`USER's message of TIME <ID@HOST>`
7%	`USER's message of TIME`
6%	`Your message of "TIME"`
17%	hundreds of other variants (average 0.4% each?)" [16]

His advice, which I followed, is to select the first angle-bracketed text in the line, and assume it is a message ID. This works 70% of the time, if you assume Zawinski's numbers are accurate.

Figure 2 shows a sample message and the actual SQL generated by insert for that message.

3.2. Retrieval

Stuffing messages into a database is useless if we cannot retrieve them! I have written scripts to demonstrate two different ways of displaying messages, assuming the user already knows which message numbers she wants. `scan` outputs a one-line summary of each message whose unique number is given on the command line. The summary contains the message number, date, sender, and as much of the subject and body as will fit on the rest of the line. By default, the messages are ordered by date.

```
$ ./scan 46 47 52 100 ...
  100 02/15/99 Christopher League   mini-Java semantics<>Hello. I have defined
  107 02/16/99 Valery Trif          Re: mini-Java semantics<>Looks good. A few
  108 02/16/99 Christopher League   Re: mini-Java semantics<>* Valery Trif
   46 02/16/99 Valery Trif          Re: mini-Java semantics<>Christopher League
   47 02/16/99 Valery Trif          Re: mini-Java semantics<>One more thought..
   52 02/17/99 Valery Trif          Re: mini-Java semantics<>This is a multi-pa
```

show outputs selected headers and the full body of each message whose unique number is given on the command line. It can pipe the message through a pager, such as `more` or `less`.

```
$ ./show 100
[Message 100]
From: Christopher League <league@contrapunctus.net>
Subject: mini-Java semantics
To: trif@SOMETHING.EDU, sho@SOMETHING.EDU
Date: Mon, 15 Feb 1999 19:44:23 -0500

  Hello.
  I have defined the syntax and semantics of a miniature Java-like
--More--(67%)
```

Now, how to determine which message numbers we want? In my proposal, I envisioned a script called pick, which would take some form of search predicate on the command line (*e.g.*, "(from Chris or to Zhong) and date $>$ (now $-$ 2 weeks) and unprocessed"), translate it to one or more SQL queries, execute them, and output a list of matching message numbers.

I have not written this script, largely because I could not decide on a precise language for the search predicates. It seems to be more an issue of the protocol design, and I did not want to invest too much effort on that front when the script itself would just be a disposable prototype.

Nevertheless, we must show how to use SQL to retrieve message that meet some intuitive criteria. In the following section, we will investigate various kinds of retrieval. All of them can be tested by typing them into a MySQL session.

3.2.1. Queries

Queries by subject, date, and from address are quite simple:

```
SELECT message_num FROM message WHERE date = ...;
SELECT message_num FROM message WHERE subject = ...;
SELECT message_num FROM message WHERE from_addr = ...;
```

In MySQL we can also do wildcard (glob pattern) matches using the `LIKE` operator, where `%` is the wildcard. This is useful, for instance, for searching on a particular date and allowing the time to vary:

```
SELECT message_num FROM message WHERE date LIKE "2000-01-28 %";
  ⟹ 1368 1369 1370 1371
$ ./scan 1368 1369 1370 1371
 1368 01/28/00 jongcoam@BIOMED.MED. hi<>Hey there: I hope you've had a good day
 1369 01/28/00 Christopher League   Re: hi<>i didn't know it was today or tomor
 1370 01/28/00 jongcoam@BIOMED.MED. Re: hi<>Hi again I don't care if you invite
 1371 01/28/00 jongcoam@BIOMED.MED. hi<>I just faxed a letter officially withdr
```

If the search string has a known prefix (*i.e.*, it doesn't start with a wildcard) then MySQL can still use the index for fast retrieval. Unfortunately, subjects often *begin* with `re:`, `fwd:`, or other uninformative annotations. Perhaps these could be stripped out on insertion.

Searching the message body text for keywords even fits into this framework, although it will probably be pretty slow: bodies are not indexed, and even if they were, the keyword is prefixed by a wildcard.

```
SELECT message_num FROM message WHERE body LIKE "%java%";
  ⟹ 9 20 37 39 45 46 48 ...
```

Note, however, that it probably isn't any slower than using `grep` on my entire MH tree.

Retrieval by recipient address is straightforward, particularly if we do not care whether the message was sent `To` or `Cc` the recipient:

```
SELECT DISTINCT message_num FROM recipient WHERE address = "league@cs.yale.edu";
  ⟹ 25 45 48 49 50 60 66 ...
$ ./scan 25 45 48 49 ...
   25 05/26/98 Sixdegrees           Evan Otis<>Name: Chris League E-Mail Addre
   45 02/05/99 Valery Trif          [Fwd: ECOOP99 Notification]<>This is a mult
   48 02/16/99 Valery Trif          classes vs interfaces<>Zhong noted that the
   49 02/17/99 Valery Trif          Re: Drossopoulou papers<>Drossopoulou et al
```

But we can also specify that we only want messages `Cc`'d to my address:

```
SELECT DISTINCT message_num FROM recipient WHERE kind = "Cc"
                                            AND address = "league@cs.yale.edu";
  ⟹ 109 111 112 123 133 134 ...
$ ./scan 109 111 112 123
  109 09/03/98 Zhong Sho            Re: Question concerning CS421:as1<> Hi, Sam
  111 10/04/98 Zhong Sho            Re: Assignment 4 - expressions that return
  112 10/05/98 Zhong Sho            Re: Pretty Printer<> |> I am not sure what
  123 10/06/98 Christopher League   Assignment 3 grades<>----Next_Part(Tue_Oct_
```

Either way, an index will be used.

If I want to retrieve all correspondence with my friend Evan, whether it was sent `To`, `Cc`, or `From` him, I need to join two tables:

```
SELECT DISTINCT message.message_num
FROM recipient LEFT JOIN message USING (message_num)
WHERE message.from_addr = "otis@something.edu"
   OR recipient.address = "otis@something.edu";
  ⟹  1 15 16 21 22 32 35 1937
```

As requested, the result set contains messages sent from Evan to me (15), from Evan to a bunch of people (1), from me to Evan (16), and from Nick to Evan (1937).

```
$ ./scan  1 15 16 21 22 32 35 1937
    1 12/02/97 Evan Otis            Anybody out there?<>Uh, hey guys... I'm bac
   15 03/27/98 Evan                 Quick question<>Hey Chris, Is a natural lan
   16 03/28/98 Christopher League   Re: Quick question<>|| Is a natural languag
   21 04/01/98 Evan                 References<>Hey guys! As I'm sure you are a
   22 04/01/98 Christopher League   Re: References<>|| Nevertheless, I was wond
 1937 05/26/98 Nick Charo           It's a boy!!!<>Joseph Nicholas Charo
   32 09/15/98 Evan                 Okay Guys..<>You have another chance to vis
   35 10/07/98 Evan                 Re: hb<>Hey Chris, Wow, thanks for the birt
```

(The message numbers are not meaningful; Evan's are small and Nick's are large because, in creating this particular database, I inserted Evan's folder from my MH tree first, and Nick's folder last!)

Next, we attempt to retrieve messages by their tags. Single tags and disjunctions are similar to cases we've seen so far. Conjunctions require a self-join (this would be true of recipients as well; it is not specific to tags). Differences require two queries.

- Retrieve *unseen* messages:

```
SELECT message_num FROM tag WHERE tag = "unseen";
  ⟹ 1900 1901 1902 1903
```

- Retrieve both unseen messages and *cool* messages:

```
SELECT message_num FROM tag WHERE tag = "unseen" OR tag = "cool";
  ⟹ 128 801 1900 1901 1902 1903
```

- Retrieve unseen messages that are also cool:

```
SELECT t1.message_num FROM tag=t1 LEFT JOIN tag=t2 USING (message_num)
WHERE t1.tag = "unseen" AND t2.tag = "cool";
  ⟹ 1901 1902
```

- Retrieve unseen messages that are *not* cool:

```
✘ SELECT t1.message_num FROM tag=t1 LEFT JOIN tag=t2 USING (message_num)
✘ WHERE t1.tag = "unseen" AND t2.tag <> "cool";
    ⟹ 1900 1901 1902 1903   (Wrong!)
```

The last query attempts what is essentially a set difference, but it is incorrect. The query returns instances of messages (such as 1901) where both t1 and t2 are, in this case, *unseen*. This cannot be prevented in general. A workaround is to use two queries: retrieve messages that are unseen, and then remove from that list those messages that are both unseen and cool.

Sophisticated combinations of all these kinds of queries are, of course, possible. The difficult part is deciding what tables need to be joined in which ways. As an example, consider the English query quoted previously: "(from Chris or to Zhong) and date $>$ (now $-$ 2 weeks) and unprocessed:"

```
SELECT DISTINCT message.message_num
FROM message LEFT JOIN recipient USING (message_num)
             LEFT JOIN tag USING (message_num)
WHERE (from_addr = "league@contrapunctus.net"
       OR (kind = "to" AND address = "sho-zhong@something.edu"))
  AND date > "2000-04-27"
  AND tag = "unprocessed";
  ⟹ 2092 1948 2094
$ ./scan 2092 1948 2094
 2092 05/05/00 Christopher League   Re: Fascinating idea<>* "Rupa Athreya" <rup
 1948 05/05/00 Christopher League   Re: [flint] [Linda.Joyce@yail.edu: summer s
 2094 05/05/00 Christopher League   Re: Fascinating idea<>Adam, did you ever ca
```

(Although not shown in this output, all these messages are marked as unprocessed; all are from me, but 1948 is also to Zhong.)

3.2.2. Threads

We have yet to discuss message IDs and threads. One of my design goals was to allow retrieval of all messages that belong to a given thread. The threading algorithms in most mail clients and news readers use `In-Reply-To`, `References`, and, to some extent, the `Subject` to arrange the messages of a thread into a hierarchy. These algorithms, of course, can only work with what they're given! If part of the thread happened to be in a different folder, it would not be included. Starting with a given (possibly singleton) set of messages, we would like to retrieve all messages in the database that belong to the same thread as any messages in the initial set. It will still be the job of the client to arrange the messages into a hierarchy.

We can find the parent(s) of a particular message by joining `reference` with `message` on `message_id` (instead of on `message_num`). It is possible (likely, even) that some message IDs in `reference` won't match anything in `message`. That is okay; we are interested in those that do match. The following query retrieves the parent of message number 117:

```
SELECT message.message_num FROM reference LEFT JOIN message USING (message_id)
WHERE reference.message_num = 117;
  ⟹ 390
```

Similarly, we can retrieve all the children of (follow-ups to) message 117:

```
SELECT reference.message_num FROM reference LEFT JOIN message USING (message_id)
WHERE message.message_num = 117;
  ⟹ 124
```

These three messages are:

```
$ ./scan 117 390 124
  390 10/01/98 Christopher League   Re: 421: Assignment 2 grade<>----Next_Part(
  117 10/05/98 Ian Schlaff          Re: 421: Assignment 2 grade<>This message i
  124 10/06/98 Christopher League   Re: 421: Assignment 2 grade<>|| I think my
```

Unfortunately, in this model, we cannot retrieve the entire thread in a single query. We must iterate these two kinds of queries until we reach a fixed point (*i.e.*, when no new messages are added to the set). We demonstrate this starting with the set retrieved so far: numbers 117, 390, and 124:

```
SELECT message.message_num FROM reference LEFT JOIN message USING (message_id)
WHERE reference.message_num IN (117, 390, 124);
  ⟹ 390 117 386      one new parent
SELECT reference.message_num FROM reference LEFT JOIN message USING (message_id)
WHERE message.message_num IN (117, 390, 124, 386);
  ⟹ 117 124 389 390      one new child
SELECT message.message_num FROM reference LEFT JOIN message USING (message_id)
WHERE reference.message_num IN (117, 390, 124, 386, 389);
  ⟹ 390 117 NULL 386 386      no more parents
SELECT reference.message_num FROM reference LEFT JOIN message USING (message_id)
WHERE message.message_num IN (117, 390, 124, 386, 389);
  ⟹ 117 124 389 390      no new children
```

When have reached a fixed point, and the thread contains the following messages:

```
$ ./scan 117 390 124 386 389
  389 10/01/98 Christopher League   Assignment 2 grades<>----Next_Part(Thu_Oct_
  386 10/01/98 Ian Schlaff          Re: 421: Assignment 2 grade<>Where are the
  390 10/01/98 Christopher League   Re: 421: Assignment 2 grade<>----Next_Part(
  117 10/05/98 Ian Schlaff          Re: 421: Assignment 2 grade<>This message i
  124 10/06/98 Christopher League   Re: 421: Assignment 2 grade<>|| I think my
```

In this example, the initial set was a singleton, but in general, I might want to begin with some larger set (*e.g.*, all messages sent to the `flint-group` mailing list) and then add to that any parents and follow-ups (which were not sent to `flint-group`, but perhaps sent privately to me).

3.3. Deletion

Deleting messages is quite trivial in this data model. Simply remove all rows (in all tables) whose message number matches that of the message to be deleted.

As part of the protocol design, however, I might suggest having the client simply mark those messages slated for deletion, perhaps using a designated tag. Then, some other process would periodically expunge the marked messages.

3.4. Measurements

Now I will attempt to quantify some of the overheads of using MySQL for email. I expected this system to be, on the whole, larger and slower than, say, MH operating on a local file system. Still, we want to know *how* big and slow it is, and whether it is acceptable or can be tuned for better performance. (Please remember that these numbers are for the tables precisely as shown in figure 1. I have not tuned the model or the server as suggested in the MySQL manual.)

The first result pertains to the size of the database. I inserted 9,999 messages from two of my MH trees, a total of 38,130k. The size of the MySQL files for this database were 59,740k, or 57% larger. The factor seemed consistent over different amounts of data. I don't believe it's anything to worry about; I'm happy to be within a factor of 2.

The next result is the (wall clock) time for inserting messages. I inserted 9,999 messages (38,130k) in two batches. The first batch was 5889 messages (20,273k, or an average of 3.44k per message). This took 0.087 seconds per message. The second batch was 4110 messages (17,857k, or an average of 4.34k per message). This batch took 0.108 seconds per message, or about 24% longer. I expected the second batch to take longer because the index files had grown, etc. But it turned out that the difference in time could be explained by the increased average size of messages in the second batch.

All the remaining results will assume the same database, with just under ten thousand messages. Now, we consider searching through *all* messages for a particular keyword in the message body. In conventional systems, this can *only* be accomplished with an external utility, such as `grep`. Although I previously predicted that this search would be fairly slow (because it cannot use indices), it is still faster than using `grep`. Here is how I would search for the word "facilitate" in my two MH trees, `~/Mail` and `~/mail`:

```
$ time find mail Mail -name '[0-9]*' -exec grep -i facilitate {} \;
```

This takes 49.5 seconds, wall clock time. (I wanted to invoke grep only once, with all the message names on its command line, but the command line becomes too long!) The analogous query in SQL:

```
$ time mysql --batch -A -q -e '
    SELECT message_num FROM message WHERE body LIKE "%facilitate%";'
```

This takes about 1.4 seconds.

Of course, I have chosen a test case where MH is particularly bad. Suppose we want all messages with subject containing "weekend" in the folder for Art J. In MH this would be:

```
$ time pick +./mail/people/artj -subject weekend
```

This command searches through the 1,160 messages in the folder and returns four matches in 0.39 seconds. In my data model, I would search for messages either to or from Art, with subjects matching "weekend":

```
$ time mysql --batch -A -q -e ' SELECT message.message_num
    FROM recipient LEFT JOIN message USING (message_num)
    WHERE (message.from_addr = "jongcoam@biomed.something.edu"
           OR recipient.address = "jongcoam@biomed.something.edu")
     AND message.subject LIKE "%weekend%"'
```

It searches through 2396 messages and returns the same four matches in 1.37 seconds. This is not too bad, considering that MySQL touched twice as many messages as the MH version, and was not allowed to use an index for `subject`.

4. Conclusion

My results are preliminary, but, I believe, promising. It does seem at least possible that a mail server based on an RDBMS could be flexible (overcoming the old folder model) and quick enough. Before extending or tuning the data model, I think I would try to design the protocol and at least sketch a client interface. Those exercises would likely inform the next generation of the data model.

References

[1] IETF Working Group. Detailed revision/update of message standards. Various drafts available at ⟨`http://www.ietf.org/html.charters/drums-charter.html`⟩.

[2] G. Barr. *MailTools*. ⟨`http://search.cpan.org/search?dist=MailTools`⟩.

[3] M. Crispin. RFC2060: Internet message access protocol–version 4rev1, December 1996. ⟨`http://www.faqs.org/rfcs/rfc2060.html`⟩.

[4] D. H. Crocker. RFC822: Standard for the format of ARPA internet text messages, August 1982. ⟨`http://www.faqs.org/rfcs/rfc822.html`⟩.

[5] M. Horton and R. Adams. RFC1036: Standard for interchange of USENET messages, December 1987. ⟨`http://www.faqs.org/rfcs/rfc1036.html`⟩.

[6] The IMAP connection. ⟨http://www.imap.org/⟩.

[7] J. Kent, D. Terry, and W.-S. Orr. Browsing electronic mail: Experiences interfacing a mail system to a DBMS. In *Fourteenth International Conference on Very Large Data Bases*, pages 112–123, Los Angeles, 1988. Morgan Kaufmann. ⟨http://www.vldb.org/dblp/db/conf/vldb/KentTO88.html⟩.

[8] nmh—message handling system. ⟨http://www.mhost.com/nmh/⟩.

[9] procmail and smartlist. ⟨http://www.procmail.org/⟩.

[10] S. Putz. Babar: An electronic mail database. Technical Report SSL-88-1, Xerox PARC, April 1988.

[11] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 3rd edition, 1999.

[12] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly, 2nd edition, 1996.

[13] R. J. Yarger, G. Reese, and T. King. *MySQL & mSQL*. O'Reilly, 1999.

[14] J. W. Zawinski. Intertwingle. ⟨http://www.mozilla.org/blue-sky/misc/199805/intertwingle.html⟩.

[15] J. W. Zawinski. Mail summary files. ⟨http://www.jwz.org/doc/mailsum.html⟩.

[16] J. W. Zawinski. Message threading. ⟨http://www.jwz.org/doc/threading.html⟩.