CERIAS Tech Report 2001-84
The Gold Mailer
by Christopher Clifton
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

The Gold Mailer

Daniel Barbará
Chris Clifton[‡]
Fred Douglis

Hector Garcia-Molina[*]
Stephen Johnson
Ben Kao[*]

Sharad Mehrotra[†]
Jens Tellefsen
Rosemary Walsh

Matsushita Information Technology Laboratory
182 Nassau Street, 3rd Floor
Princeton, NJ 08542-7072 USA

Abstract

The goal of the Gold project is to implement a set of tools to interoperate with unstructured and semi-structured data and services. We have selected the electronic mail domain as the first scenario in which to implement and demonstrate some of the tools. This paper describes the Gold Mailer, a system that provides users with an integrated way to send and receive messages using different media (e-mail, faxes, phones), efficiently store and retrieve these messages, and access a variety of sources of other useful information. Our mailer solves the problems of information overload, organization of messages and multiple interfaces. By providing good storage and retrieval facilities, it can be used as a powerful information processing engine covering a range of useful office information. Unlike some recent research efforts ([Ter90, KTO88, LMY89]) which have implemented mailers with relational-like query facilities, the Gold mailer provides a database system interface that fits more naturally. This document explains the current implementation of the Gold mailer.

1 Introduction

The proliferation of electronic communication including computer mail, faxes, and voice mail, has led to a variety of disjoint applications and usage paradigms. For example, in a typical office today, a user might read and send electronic mail on a workstation, communicate with a voice mail system using a touch-tone keypad, and walk to another office to send a fax. No single system allows a user to access all of these forms of communication at once, or to intermix them. Additionally, electronic mail (e-mail) is becoming a central component of modern offices. More and more interactions take place through e-mail, regardless of whether people are geographically separated or not. Many people are finding it convenient to interact with their secretaries or with a colleague next door electronically simply to avoid constant interruptions and to keep an organized record of tasks to be done. At the same time, more and more important office information is arriving electronically, either through mailing lists, or news services such as USENET (net news) [Hor] or bulletin boards. On the other hand, as the storage capacity of modern computers has increased substantially, allowing users to store more and more messages for future reference.

Unfortunately, as the volume of messages grows, the following problems arise:

- There is an information overload: one cannot cope with all the incoming messages. Many of the messages are "junk" [Den82] and need to be filtered out [LMY89]. Others are simply not relevant at the moment, but could become so later. However, the user is still forced to process them in the order they show up, not in the order they are needed.
- Once they are processed, many messages need to be stored or filed. Most mailers have primitive filing facilities, and stored messages are seldom found when needed. (A common question faced by many e-mail users is: "In what file did I stash away that message that came in last month?")
- Users are forced to deal with multiple mail systems. For instance, regular electronic messages come in one way, and FAXes are received another way.

In this paper we describe a new electronic mail system, the Gold mailer, that addresses these problems. A primary goal of the Gold mailer is to develop a

[*]Current address: Stanford University, CS Dept., Stanford, CA 94305
[†]Current address: University of Texas, Austin, CS Dept., Austin, TX 78712
[‡]Current address: Northwestern University, EECS Dept., Evanston, IL 60208

1063-6382/93 $03.00 © 1993 IEEE

system capable of sending and receiving messages using different media, and to store and retrieve messages efficiently. The system will treat all these media in an integrated way. It is clear that some sort of mail database is essential for storing large numbers of messages and later finding them. Current research efforts (e.g., [Ter90, KTO88, LMY89]) have provided mailers with relational-like query facilities. Typically, each stored message is viewed as a tuple with fixed fields such as "from," "to," and "subject." Users can then find messages that satisfy some conditions such as "date > Jan 10" or "from = fred."

Instead, Gold views messages as unstructured and semi-structured sequences of words. Queries can look for messages containing certain words (e.g., find messages with "fat" and "cat"). Queries can also specify proximity searches (e.g., "fat" within x words of "cat"). In this paper we will argue that this unstructured model is more appropriate for message searches and can also encompass a wider variety of message formats, even unknown formats.

Our goal is to find out how this data should be managed. The traditional "database approach" would be: "here is the standard interface for data management (e.g., relations and SQL); let the user figure out how to use it." Our approach instead is to start from the application , and design a database system interface that fits naturally. Hence , our work can be thought of as "database interface" research.

A secondary goal is to allow use of existing interfaces for electronic mail. (The current prototype of the Gold mailer is upwards compatible with *MH* , a previous prototype was compatible with the BSD Mailer.) For Gold we decided to work with existing mailers and add functionality to them. (For instance, adding the possibility of handling multimedia mail.) All old features remain, and users can migrate to the new functionality as they need it. A more modern graphical interface to the Gold mailer is also provided for users who prefer it.

Once a mailer has good storage and retrieval facilities, it can be used as a powerful *information processing engine*. For example, messages from news services and bulletin boards can be automatically stored in the database, without having the user see them. When the user requires information of a given type, he can query the database for relevant messages. This is the same approach followed by systems such as WAIS [Kah89] and the Community Information System [GLB85], except that now a single user interface provides access to both electronic mail and news messages.

The mailer database can also be extended to cover other useful office information. For example, Gold can store "address cards" that contain addresses, phone numbers, and e-mail addresses, and other information. From within the mailer, a user can query these cards in exactly the same way he would search for old messages. For instance, a user can look for the e-mail address of someone at "Computer," "Science," and "Berkeley." The answer will then appear as a set of "virtual" messages that the user can examine. Replying to one of these found cards is equivalent to sending a message to the desired e-mail address. Hence, the entire operation of looking up an address and sending e-mail is done entirely within the mailer, with almost no need for new commands. To achieve this fusion of information (messages, address cards, and other types to be described), it is important that the mailer provide a flexible model for stored objects. This is another reason why we favored our unstructured object approach as opposed to making messages structured with predetermined fields such as "to" and "from."

The environment for which Gold is being developed consists primarily of workstations connected to local-area Networks on the Internet. Workstations handle electronic mail and other data, which may be stored locally or on shared file servers. In addition, phone lines connect devices like phones, faxes and modems. Some of the workstations have interfaces to the phone lines via a fax board or a phone board. These interfaces provide the user with the capability to send and receive faxes and phone calls. Scanners allow the user to input printed documents into his/her system to be handled by the system. Other computers may act as repositories of other sources of information (e.g., articles in NetNews, DowJones activity, library catalogs.) The user will also be able to access these sources using the mailer interface.

The paper is written as follows. In Section 2, we describe our approach and goals. In Section 3 we address the indexing issues that rise in designing the mailer. Finally Section 4 offers conclusions and sketches future goals.

2 Approach

The architecture of the Gold system is given in Figure 1. Users interact with the front end, issuing commands to read new mail messages and retrieve others from the database. When a user decides to store a new message, the front end gives it to the preprocessor; it breaks it up into tokens and gives it to the index engine. The preprocessor can also receive messages directly from a news feed, for automatic indexing. The preprocessor produces a *canonical* file that is used by the index engine to create the appropriate data structures. Messages are stored in an object store or database (currently, each message is stored as a separate UNIX file). [1] The front end also submits queries to the index engine. After a search, the engine gives the front end a list of the matching objects (file identifiers in our case). The front end retrieves the objects for displaying. An index engine, in general,

[1] UNIX is a trademark of UNIX Systems Laboratories.

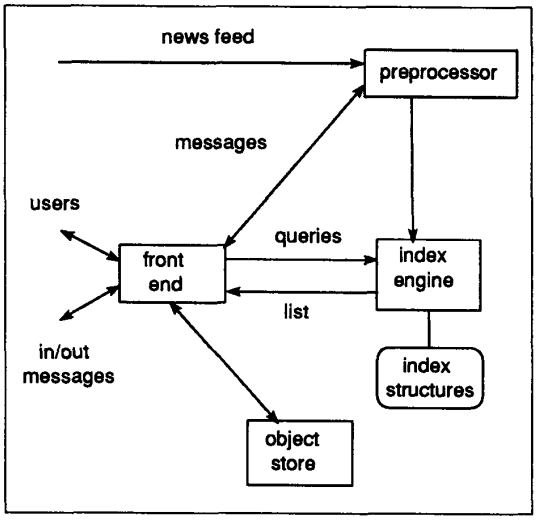

Figure 1: Gold mailer architecture.

can receive queries and storage requests from several front ends, including those on other machines. The implementation of concurrency control issues that arise once multiple front-ends are permitted is ongoing (see Section 3).

2.1 Supporting Mail Message Data

A mail message commonly have two pieces of data:

- *Headers*, i.e., value pairs such as "subject: time for lunch" or "from: joe."
- The body of the message

The user may view the mail message simply as a collection of words, where each word is at a given location in the original message or document. Alternatively, the user may want to view the message as a semi-structured object with separate components such as "subject," "from" and the actual body of the message. Notice that the types of queries that one may ask vary in the two views. For instance, in the first one the user can enquire about the relative position of two words regardless of where they are in the message. In the second, the relative positions of words are likely to refer to the same segment of the message. (Words "time" and "lunch" within the header "subject.")

One of the key design decisions in Gold involves our handling of structured fragments within a document. The key is to be as flexible as possible in the index engine and let the preprocessor decide about the strategy. One can think about the data stored in the index engine as a relation with the following schema:

keyword	document_id	position	header

The attribute document_id points to the indexed object (e.g., message, address card). The attribute position signals the position in which the keyword occurs within the document. The header attribute is provided to support headers if the application decides there is a need for it. In the general case, this attribute can receive a default value (no_header). Two points are worth noticing here:

- This relation is just a way of viewing the data supported by the index engine. The query language does not make use of the attribute names mentioned above.
- The values of attributes in this relation do not have to be atomic. This gives a lot of flexibility to the engine. This fact is indeed exploited in the current implementation of the mailer, as we will see shortly.

To support the first view of a mail message (simply a collection of words), a preprocessor can recognize a colon as a special delimiter between a header and its value, and report the position of the header and what follows in a line as identical. For example, if the "s" in "subject: time for lunch" is at byte 55 in the original message, then the preprocessor will report that the words "subject," "time," "for," and "lunch" are all at position 55. If a user wishes to find messages with keywords "lunch" and "time," then any message with both these words will be retrieved, regardless of whether they are in the subject line or not. If instead a user wishes to search for "lunch" and "time" appearing in a "subject" line, then a query is submitted looking for those three words occurring at exactly the same position (i.e., distance between them is zero). This last query will not retrieve messages that simply contain the words "subject," "lunch," and "time" somewhere in them. However, it will retrieve a message with the line "time: lunch subject ... " or "subject: lunch for time." This can be viewed as the following instance of the index engine relation:

keyword	document_id	position	header
subject	doc_xxx	55	no_header
time	doc_xxx	55	no_header
for	doc_xxx	55	no_header
lunch	doc_xxx	55	no_header

Alternatively, a mail message may be considered as a collection of labeled fragments. Thus, the preprocessor will report the word "subject" as the label for the words "time," "for," and "lunch", and provide byte offsets of the words within the fragment resulting in the following instance of the relation:

keyword	document_id	position	header
time	doc_xxx	0	subject
for	doc_xxx	1	subject
lunch	doc_xxx	2	subject

In the first relation, the right way to retrieve this document is to ask for documents that contain the keywords subject, time, for, lunch (or a subset of them) in the same position. In the second, we should ask for documents that contain the words time, for, lunch (or a subset of them) under the header "subject." The first prototype of the mailer (which used the BSD mailer as the starting point) used the first alternative, while the MH based Gold mailer uses the second. The inclusion of headers allows for storage and retrieval efficiency. Notice however, that the implementation of the index engine remains the same.

The preprocessor is also responsible for identifying certain multi-word tokens and mapping them to a standard representation. For example, strings such as "Dec. 31, 1999," "December 31 1991," "12/31/91" should all map to a single token that represents the given date. The original words should still be indexed, e.g., in case one wishes to search for occurrences of "Dec" or of "31." An additional token for the full date will be inserted (at the position of the first character of the string), to locate the object when we search for a particular date, regardless of its format.

As mentioned previously, the index engine stores a variety of documents. The document stored may be mail messages, address cards or other types of objects discussed later. A user may be interested only in documents of a specific type. For example, a user may wish to retrieve address cards that contain the word "Smith". There is thus a need to store the information about the type of the object. Again, there are many ways of supporting this in the index engine. A simple one is to use the keyword mechanism by letting the preprocessor add the string "type: address_card" to the card. An electronic message is stored with the "type: message" in it. Then, if the front end wishes to retrieve only address cards, it adds to its query the clause "type: address_card." Alternatively, we can exploit the support that the index engine offers for non-atomic attributes and use the header attribute to encode the type of the object. In the current MH implementation, the preprocessor passes a header string that encodes information as follows:

keyword	doc_id	pos.	header
Smith	doc_yyy	55	name#add_card

In this way, the keyword "Smith" is tagged with the word "name" as its header and the word "address_card" as the type of the document to which it belongs. Notice that the index engine remains oblivious to such encodings. It is sufficient to encode the queries in the proper way, so as long as the engine is used by the same application that handled the preprocessing, there will be no confusion. Using non-atomic values for the headers allows the engine to be more efficient in searches. To see that, think that a search for the address card containing the name "Smith" would involve a join operation if the two headers were not encoded in the same attribute. If we want to avoid the join operations and still have atomic values, we would have to have separate attributes for each possible header. Since we want to deal with different kinds of mail formats and do not know beforehand how many different kinds of headers there will be, simply letting the preprocessor decide about the encoding allows us to use always the same view supported by the engine.

The Gold mailer supports the feature of *message annotation.* When a new message is to be stored, the front end allows the user to enter additional words for indexing. For instance, the user may want to associate the words "budget" and "sales" with a message, even though these two words do not appear in the message itself. The front end essentially appends these added index words at the end of the message, and the index engine does not treat them differently from other words in the message. Mailers usually provide the capability of defining folders. A folder is a "bin" with a name in which messages can be collected for future reference. The Gold mailer uses annotation to implement folders, by adding fragments such as "folder: inbox" to the original message.

Notice that the usage of message annotation helps to create a "virtual folder" of messages containing the keywords used for the annotation. However, the user does not get exactly the same functionality as folders. Notice that, for instance, querying the system for all the messages that contain the keyword "databases" will not only retrieve messages annotated by that keyword, but also messages that contain the keyword somewhere in the text. Using both folders and annotation, the user has a lot of flexibility in organizing the messages.

2.2 Supporting Multi-media Documents

The Gold Mailer deals with information coming from media other than electronic mail. In particular, we can receive and store information coming from fax machines. Faxes are first received by a fax board (this board can be connected to any machine in the local area network), where a daemon process sends them to an optical character recognition server (OCR). This software, with the help of an online dictionary, recognizes as many words as possible in the fax and creates a file with these keywords. These words will be used to index the document in the index engine. Notice that perfect OCR is not necessary for this application, and even low hit rates are good enough to produce a handful of keywords. If the addressee is recognized by the

OCR, a message is sent to the user in an electronic mail format containing pointers both to the fax image and to the file that contains the keywords. If it is not possible to recognize the recipient, the daemon sends the message to a predetermined "faxmaster" who will read the fax and reroute it to its final destination. Both the fax electronic image and the keyword file are stored in a fax directory (a directory in the local file system), but only the addressee has reading rights to it. Storing the fax images in a directory allows for easy sharing of faxes when we have multiple recipients, while saving storage space. The same is valid if the user wants to forward the fax to other users. This is true as long as the users are in the same local-area network. (We assume that all the machines in the local area network share a common file system. Of course, if the fax is to be sent electronically to a user not in the local system, a copy should be included in the message.) Faxes are displayed using the X-Windows library. Gold also allows the user to send messages via fax. The user simply composes the message in the terminal and selects fax as the media. The fax number is retrieved from the address card of the recipient.

The Gold Mailer also supports the ability to compose and receive messages that contain different kinds of types, e.g., textual data, images, etc. This is particularly important, for instance, when the user is trying to send a fax to a recipient that does not share a common file system with the sender. In order to implement this, Gold supports the multimedia mail format, MIME (Multipurpose Internet Mail Extensions), which extends the established RFC-822 Mail [Cro82] and Bulletin Board Internet message protocols.

2.3 Query Language Examples

Our goal was to develop a query language that was a simple and flexible as the message model. The simpler and more common a query is, the easier it should be to specify. We also want the language to be compatible with common mailer interfaces. Let us illustrate with examples. (Complete details can be found in [BJM92].) A common command to send mail to users x and y is:

mail x y

Gold provides a new type of mail command that is search based, e.g.,

mail friend, Princeton

It searches through the database for objects with the keywords "friend" and "Princeton." This will locate one or more address cards for people that have both these keywords associated with them. (The user can then send mail to the selected addresses, or to subsets of them.) The command

mail friend, Princeton Joe, Berkeley

searches for cards with either "friend" and "Princeton" *or* "Joe" and "Berkeley" in them. Thus, the space between terms indicates an "or" to match the syntax of the original mail command (where a space means that messages should go to both recipients). We use the comma then to indicate a logical "and" of the terms. The actual character used for delimiting is not critical (it can be easily changed); the key point here is that it is important to match the query syntax to that of the mailer commands.

Since we expect most queries not to involve proximity searches, this should be the default. Thus, in the query above, "Joe" and "Berkeley" can appear anywhere in the address card. If proximity is required for the match, it can be specified by grouping together the terms involved. The notation "[Joe, Berkeley](25)" means the words should be within 25 words of each other. (When the number is left out, a user-changeable default is used. We do not expect most users to care strongly if two terms are within 25 instead of 26 bytes.)

The command

find subject:lunch

is the way to find messages that contain the word "lunch" labeled by the header "subject" (as shown in one of the tables of Section 2.1), while the command

find [subject,lunch](0)

is the way to find messages that contain the keywords "subject" and "lunch" in the same position (as shown in the other table of Section 2.1).

We have developed a graphical front end for the standard MH mailer in both X11 windows for machines running UNIX (or a compatible operating system) and in the NeXT application developer. The same set of commands can be invoked graphically or via the extended version of MH. Figure 2 is the main screen presented to the user when the Gold mailer is invoked. The interface allows the user to incorporate new messages, send messages using the information stored in address cards, edit address cards, and pose queries to the database to retrieve messages.

3 Indexing

In this section, we describe the indexing issues involved in the Gold mailer. For the purpose of efficient query evaluation, the index engine keeps suitable index information about the documents. The index engine basically consists of three components– *server*, *engine*, and the *cache manager*. The server attends requests from the front-end (mailer) at a given UNIX port. The mailer communicates with the server using UNIX sockets. The engine maintains the information corresponding to the messages on the stable storage (the disk) and supports the basic operations of the index engine. The cache manager keeps a cache of the data structures in main memory. Several main memory blocks are allocated to the cache manager when the index engine is invoked. The cache man-

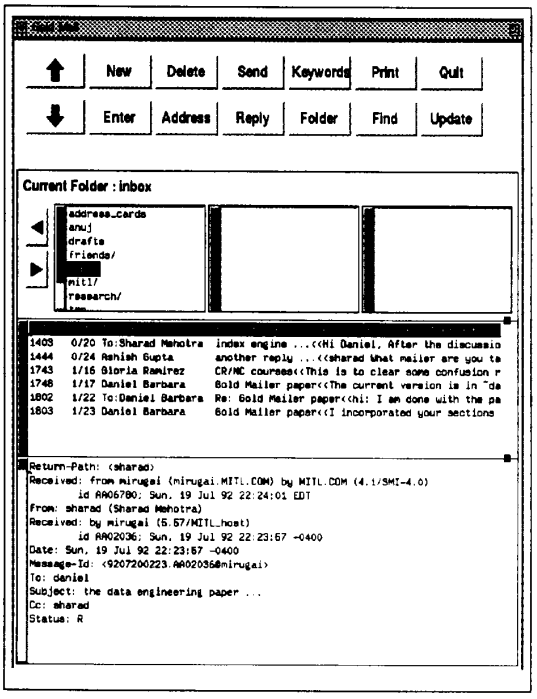

Figure 2: Main Screen

ager brings the blocks containing the data from the secondary storage (the disk) on demand into these main memory blocks and replaces them using the LRU strategy.

As mentioned previously, the information stored by the index engine may be viewed as a set of records (k, d, p, h), where k is the keyword, d the document identifier, p the position, and h the header. The index engine supports the following three operations:

1. *insert d*: This command instructs the index engine to insert all the records for a document d.
2. *delete d*: This command instructs the engine to delete all the records corresponding to the document d.
3. *retrieve query*: This command instructs the engine to find all the documents that satisfy the query and return the list of names to the mailer.

Note that the response time for the evaluation of the queries is critical for the acceptable performance of the mailer. The index structure chosen for the index engine allows for the efficient evaluation of the queries. Although the Gold mailer has the flavor of an information retrieval system (IRS), there are basic differences that influence our design of the index engine. Those are:

- In an IRS, the insertions are usually done at night, when the workload is low. The index gets reconstructed once a day and the new documents are not visible until that happens. In the mailer, however, depending upon the nature of the document, the user may expect to see a newly inserted document immediately.
- In an IRS deletions are never performed. In the mailer, a user might decide that a message is no longer of interest and therefore, should be deleted. Thus, deletions are an issue for the index engine.

In the remainder of this section we will provide an overview of the internal data structures maintained by the engine and the algorithms it uses to handle the requests made by the users. The complete details can be found in [BJM92].

3.1 File Organization and Data Structures

Our choice of the organization of the files and the data structures is dictated by basically two factors: the nature of the queries permitted in the system and the space requirements to store the information. Note that our query language only permits certain types of queries. The following points about the nature of the queries are noteworthy:

- The queries involve searches for message identifiers that contain a given set of keywords. A typical example of a query is to retrieve the set of message identifiers (names) such that the message contains keywords "mitl" and "gold".
- Most of the queries will *not* be range queries. For example, we do not expect to query the index engine for message identifiers of the messages that contain a keyword between "cat" and "dog". This will largely dictate our choice of the index structure.

Since the queries require the index engine to perform keyword searches, we index the records based on keywords. Further, since we do not expect range queries, we store the records as a *hashed file* [Ull88], hashed on the keywords. Thus, the records corresponding to a keyword are kept contiguously on disk blocks resulting in minimal disk access during retrieval. Instead of keeping a separate record for each position at which the keyword appears in the document, for each unique document and header combination, a list of positions in which the keyword appears is kept. This results in substantial space compaction without degradation in performance. The set of positions corresponding to records in the same bucket are also kept contiguously in disk blocks.

Instead of keeping the name of the documents along with the records in the bucket, a list of documents referred to as the *document list* is maintained. This is done to improve the response time for the deletion requests as will become clear later. Finally, the records corresponding to a document in the hashed file are chained together forming a list, referred to as the *next key list* for the document. A pointer to the head of this list is stored with the entry for the document in the document list. The next key list is used to locate records corresponding to a document when deleting the entries of a document from the hashed file.

3.2 Engine Operations

In this section we briefly describe our implementation of the basic engine operations, that is, the insert, delete and retrieve operations.

Insert: If we were to insert the records for a document directly into the hashed file, we will need to access disk blocks on the order of the number of distinct keywords in the message which will be very costly, especially for large message databases. Thus, in order to improve performance and keep disk activity to a minimum during the time the user is requesting operations from the system, we perform the insert operation differently. The records corresponding to a document are inserted into the *overflow lists*. Overflow lists, one per keyword, exist only in main memory. To insert the records of a document into the overflow lists, a block (or more depending upon the size of the message) of main memory is claimed from the cache manager. For each keyword in the document the corresponding record is placed in the acquired block(s) and the record is inserted to the tail of the overflow list. Thus, the records for a document in the various overflow lists are kept contiguously in main memory blocks, unlike the way records are kept in the buckets (in the buckets the records belonging to the same keyword and not the same document are kept contiguously). This makes the deletion of blocks from the overflow list after they are reflected into the disk very simple– we can simply return the blocks occupied by the records of the message to the cache manager. Further, it reduces the number of main memory blocks required for maintaining overflow lists (else a message with n keywords may have required us to request for up to n main memory blocks for the various overflow lists!).

We call a process that implements the above insert algorithm a Mark Insert (MI) process. After the termination of the MI process, the control can be returned to the user executing an insert operation. The insertion is periodically reflected in the disk image by invoking an I process that makes the updates corresponding to a document to the disk.

Delete: A deletion involves finding every keyword in the document and deleting the corresponding entry. If the records for the document are in the overflow list (and thus in main memory), then deletion can be efficiently performed. However, if the records have been inserted into the hashed file on the disk, then similar to the case of insert, deletion of the records for a message requires accesses to various disk blocks which if done online will substantially increase the response time to the deletion request. For this reason, if the records for the document are not on the overflow list, then the entry for the document in the document list is marked as "deleted" and the prompt returned to the user. We call a process that implements the above actions as a Mark Delete (MD) process. As in the case of insert, the index engine periodically updates the disk image by actually deleting the entries in the hashed file. We call a process that implements the actual deletion a D process.

Retrieve: To retrieve the message names that satisfy a query, the engine traverses the buckets in the hashed file. Note that since we keep the records in a bucket contiguously in the disk blocks, the amount of disk access during retrieve is reduced. To retrieve documents for a query, the buckets and the overflow list corresponding to each keyword that appears in the query are traversed. Due to space restrictions we do not discuss our query optimization algorithms details of which can be found in [BJM92]. When returning the answer to the mailer, the engine follows the pointer into the document list to retrieve the name of the document. If a document is marked as deleted, its name is not returned in the target list. We refer to a process that performs a retrieve as an R process.

3.3 Concurrency Control and Recovery

Even though the Gold mailer is a single user system, processes executing concurrently share the data structures and the records stored by the index engine. This is due to the presence of the I and D processes that may be executing in the background along with the other user requested MI, MD, R processes. Presence of concurrent processes sharing common data structures may result in a violation of the consistency of the database and may further result in the user retrieving erroneous data (see [BJM92]).

One way to prevent inconsistency from occurring is to execute each operation sequentially. This, however, is unacceptable since it may require that the user-requested R, MI or MD process be delayed until the termination of an I or a D process (in case an I or a D process began execution before the arrival of the user-requested process). We, therefore, modify our algorithms for the R, MI, MD, I and D processes to

obtain locks in such a fashion that inconsistent scenario does not result. Locks in our algorithms are obtained at the page level.

Another issue in the index engine is that of ensuring atomicity of the processes in presence of system crashes and process failures. To ensure atomicity, in presence of failures, the effects of the partially executed MI, and the MD processes are undone. However, since I and D processes are background processes, and undoing them may result in a substantial loss of work, they are restarted from the point they were at immediately before the crash. For the purpose of recovery, the engine performs both physical logging at the page level, as well as logical logging at the level of the operations (e.g., insert, delete from a bucket).

Due to space limitations, we are not able to describe in details our concurrency control and recovery algorithms, the details of which can be found in [BJM92]. The concurrency control and recovery mechanisms are being added to Gold at this time.

4 Conclusions

We have designed and implemented a system that allows users to store and retrieve their messages efficiently. The retrieval is content-based, thus freeing the user from having to browse in their previously defined folders (a usually burdensome task). Additionally, users can select the preferred media of communication to their peers, without having to remember the specific details such as phone or fax numbers or e-mail addresses. The system is currently in use at MITL.

In the process of designing this system, we have developed an efficient indexing software (the Index Engine), which incorporates issues such as concurrency control and recovery. This piece of software will be a cornerstone for the development of other stages of the Gold project. Due to its simplicity, it can be made to index objects other than files (e.g., databases, library catalogs, etc.).

The mailer corresponds to the first stage of the Gold project. A second stage, currently under development focuses on the notion of accessing other information sources other than the local "database" or e-mail messages. The notion is an extension of the work in [GJSJ91], in which information retrieval techniques are used to allow access to the file system. We intend to use our query language and indexing techniques to let the user find information in a vast array of sources. A long term goal of the project is to consider all these sources as a very large, heterogeneous, multimedia database in which the user will want to run transactions simply and effectively.

References

[BJM92] D. Barbará, S. Johnson, and S. Mehrotra. The design and implementation of the index engine. Technical Report MITL-TR-34-92, M.I.T.L, 1992.

[Cro82] D. Crocker. Standard for the Format of Arpa Internet Text Messages. Technical Report RFC-822, Network Information Center, August 1982.

[Den82] P.J. Denning. Electronick Junk. *Communications of the ACM*, 25(3):163–165, March 1982.

[GJSJ91] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole Jr. Semantic File Systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principl es*, pages 16–25. Association for Computing Machinery SIGOPS, October 1991.

[GLB85] D. Gifford, J.M. Lucassen, and S.T. Berlin. The application of digital broadcast communication to large scale information systems. *IEEE Journal on Selected Areas of Communication*, May 1985.

[Hor] M. Horton. How to Read the Network News. Technical Report UNIX Documentation.

[Kah89] B. Kahle. Wide area information server concepts. Technical Report Anonymous FTP from think.com:/public/wais/wais-8-aXX.tar.Z, 1989.

[KTO88] J. Kent, D. Terry, and W.S. Orr. Browsing electronic mail: Experiences interfacing a mail system to a DBMS. In *Proceedings of the Fourteenth International Conference on Very Large Databases, Los Angeles*, pages 323–336, August 1988.

[LMY89] K. Lai, T.W. Malone, and K. Yu. Object Lens: A "Spreadsheet" for Cooperative Work. *ACM Transactions on Office Information Systems*, 6(4):332–353, October 1989.

[Ter90] D. B. Terry. 7 Steps to a Better Mail System. Technical Report CSL-90-12, Xerox Palo Alto Research Center, 1990.

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems. Volume I.* Computer Science Press, 1988.