Palo Alto Research Center

Walnut: Storing Electronic Mail in a Database

James Donahue and Willie-Sue Orr

XEROX

Walnut: Storing Electronic Mail in a Database

James Donahue and Willie-Sue Orr

CSL-85-9 November 1985 [P85-00028]

© Copyright 1985 Xerox Corporation. All rights reserved.

Abstract: Walnut is an electronic mail storage and retrieval system developed for the Cedar environment. The most novel aspect of Walnut is its use of a general-purpose relational database for storage of messages. This paper discusses the design and implementation of Walnut, with particular attention to the problems of using a database system for this type of application.

CR Categories and Subject Descriptors: H.2 **[Database Management]:** Systems – transaction processing; H.4.3 **[Communications Applications]:** Electronic mail.

Additional Keywords and Phrases: Cedar, integrated applications, background processing.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

designers was the emphasis placed on asynchrony. One of the main goals of the Cedar system was easy switching between tasks and allowing the user to work on new tasks while previously started ones completed. This emphasis imposed several requirements on Walnut:

1. The user can view several components of a Walnut database simultaneously. Thus, Walnut operations must be prepared to keep the contents of multiple windows up-to-date; we guarantee that the effect of any user operation on the database will be consistent with the portion of the database that is currently displayed.
2. More importantly, Cedar users want to leave Walnut running constantly—they don't want to have to restart the system to begin reading their mail again. Much of what we do can be characterized as responding to our mail, either by answering messages directly or incorporating information from our mail into the programs and papers that we write. Because we store Walnut databases on a shared file server, it is important to be able to give up the connection to the server when the user is idle. Providing the consistency guarantee stated above, while not going to the extremes of either holding onto long transactions or redisplaying the contents of several Walnut windows when starting up after a long idle period, was a major design problem.
3. Finally, having plenty of things to do is no help when the only thing that you really want to do takes a very long time. Walnut has two operations that are potentially much more expensive than all of the others: reading mail from mail servers and expunging old mail from the system. New mail retrieval is the more important, because it is done frequently by most users. We currently use background processing to add new mail to the system and have designed (but not yet implemented) the background collection of space consumed by expunged messages. In this way, we were able to smooth out the performance of the system substantially; however, as we discuss below, the cost of these improvements in perceived performance is a rather substantial complication in the internal processing of the system.

2.2 The Mail Distribution Facilities

Walnut is only a mail storage and retrieval system; it relies on the Grapevine message transport system to provide the new mail to store in a Walnut database and to deliver the messages that a Walnut user composes. Grapevine is discussed in [Birrell83]. It is used by Walnut through a GrapevineUser package that runs on the client workstation; for the purposes of this paper, the GrapevineUser package provides four operations:

1. Wait until new mail exists for the current user on one or more of the Grapevine servers,
2. Enumerate the servers on which the current user has a mail box (in which there might be new mail) and establish a connection with each of them in turn,
3. Copy messages from the current server to a file, and
4. Flush previously copied messages from the server. (Copying and then flushing allows

protection against crashes.)

It is easy to write a "new mail retrieval" section of Walnut that would perform correctly: when new mail exists, copy the new mail into the database and then flush the messages from the Grapevine servers. However, there are some performance characteristics of Grapevine that have to be considered when building a responsive system.

1. The Grapevine servers are slower than most of our workstations (and the servers are generally heavily loaded). Thus, fetching new mail is relatively slow—it can take several minutes to retrieve new messages when returning from a vacation or a long weekend.
2. Grapevine server crashes are rare, but they do occur. Unfortunately, a Grapevine server may crash between the time at which the user has been told that new mail exists and the time at which he attempts to fetch it. And when this happens, the error indication does not happen until the connection attempt times out a few minutes later.

Both of these considerations make the cost of retrieving new mail hard to predict. If the servers are up and lightly loaded, and there aren't many messages to fetch, and you are near the server on the network, and ...., it may go quickly. On the other hand, there are frequently situations in which new mail is going to arrive slowly, or possibly not at all. As we discuss later, we use background processing in Walnut to attempt to minimize the variance in responsiveness this causes.

2.3 File Storage and Databases in Cedar

It generally takes three or more machines to read mail: one or more Grapevine servers (where the user's mail boxes reside), one or more file servers (where the user's mail database resides), and the user's workstation (which is running the Walnut program).

2.3.1 Alpine

Walnut stores mail files on Alpine file servers [Brown84]. Alpine is a file server program (written in Cedar) that provides page- and file-level locking and transactions; a general-purpose remote procedure call package is used to transfer pages to and from Alpine servers. Walnut uses Alpine transactions to prevent inconsistent updates due to either concurrent access or server crashes.

Although Alpine can run on a client workstation (it will coexist with the normal Cedar file system), most Walnut users store their files on separate Alpine server machines. The reasons for storing files on a server are simple: backup of the server is done by the administration and the Walnut user is then free to read his mail from any Cedar machine that happens to be available. The ability to easily move among machines is particularly important for our summer interns, who typically do not have personal workstations at their disposal but must share a pool of public machines; using a shared server also makes it possible to have public mail databases. However, using a file server machine introduces some interesting failure modes and makes it necessary to manage the connections with the shared resource.

2.3.2 Cypress

Walnut manipulates Cypress databases [Cattell83]. Cypress is an entity-relationship database system developed in Cedar. The details of the Cypress data model are not particularly important; almost any relational system would have served us as well. Cypress provides a fairly low-level programmer's interface, which is important since we use the database to record and to manipulate a large amount of information about the state of Walnut processing. The database is a natural place to store state information and the Cypress programmer's interface is efficient enough to make this practical.

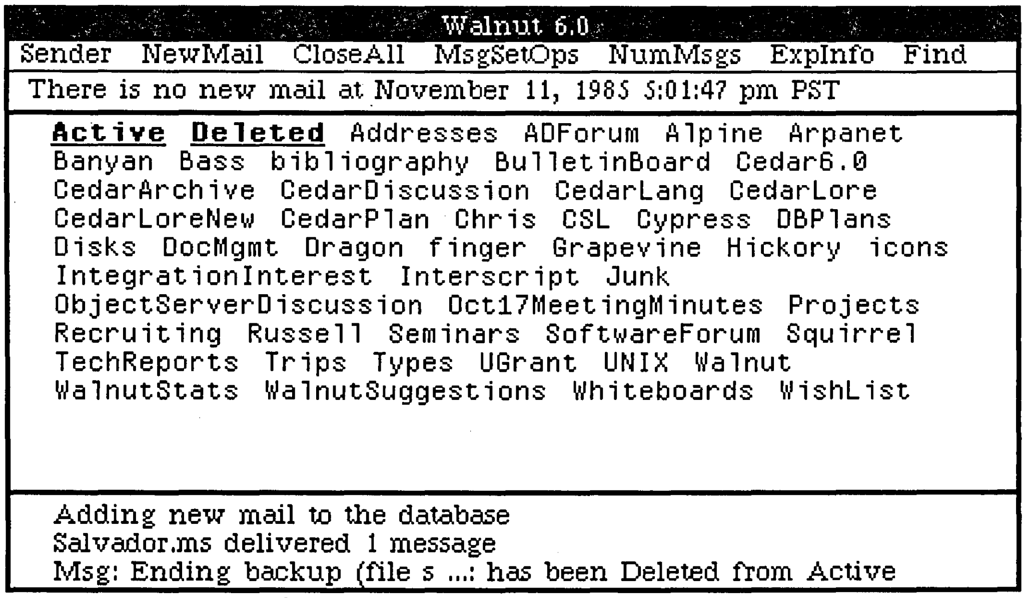
Cypress currently runs on the client workstations, *i.e.*, the workstation running Walnut also runs Cypress. We have plans to change Cypress so that, like Alpine, it can run on a separate server; when this is done, we will be able either to run Alpine and Cypress on the same machine (as a "database server") or to run Alpine on one machine, providing only file service and run Cypress on a separate machine, communicating with both the client machine (running Walnut) and the Alpine servers using remote procedure calls.

Cypress uses Alpine transactions in a straightforward manner: instead of doing logical locking of tuples or relations, Cypress simply locks the pages of the database as tuples are read or written. Because the mapping from Cypress entities and relationships to Alpine pages is not under the control of the Cypress client, the client cannot depend on logically independent operations to be physically independent on the Alpine server. (Cypress attempts to perform some co-location of tuples for the same entity on the same page, but this is done to improve performance rather than to reduce transaction conflicts.)

3. The Walnut User Interface

As we mentioned above, the user's view of a Walnut database is relatively straightforward. A database consists of messages with immutable properties and message sets with changeable contents. This is reflected in the user interface by having three separate types of windows on a Walnut database:

1. The Walnut "control window," which contains the list of all the current message sets (in Figure 1, the built-in message sets Active and Deleted are highlighted), buttons for basic operations such as shutdown and retrieval of new mail, and a typescript that records the history of user interactions.
2. A Walnut message set displayer lists the messages in a message set and provides a menu of operations for them (Figure 2). The message names displayed in italics have not been read; the message name displayed in bold is the "current selection" in the message set. (The choice of these fonts is a user-specified option.) The operations in the menu use the "current selection" as an implied argument; the AddTo and MoveTo operations use the selected message sets in the Walnut control window as arguments. A message in a message



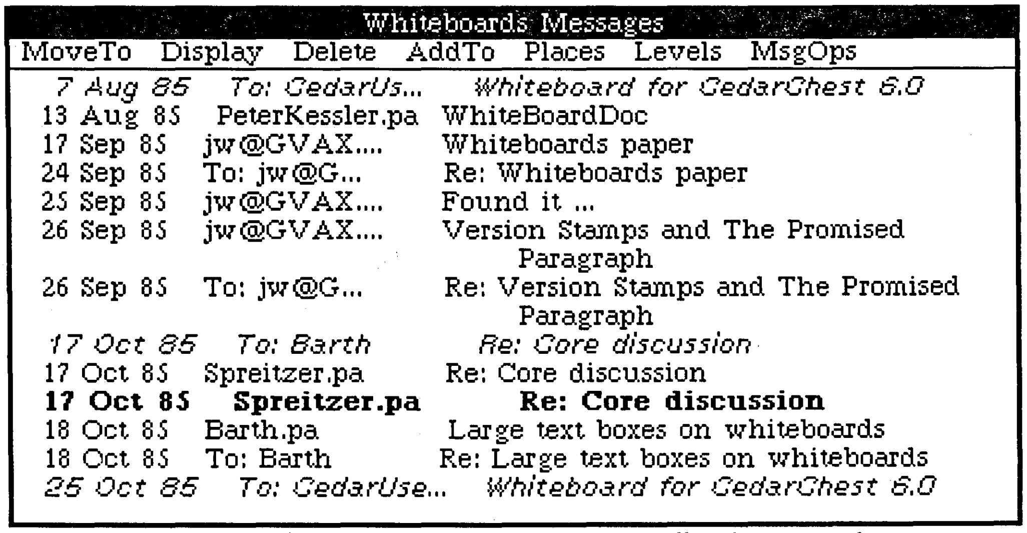
```
There is no new mail at November 11, 1985 5:01:47 pm PST

Active Deleted Addresses ADForum Alpine Arpanet
Banyan Bass bibliography BulletinBoard Cedar6.0
CedarArchive CedarDiscussion CedarLang CedarLore
CedarLoreNew CedarPlan Chris CSL Cypress DBPlans
Disks DocMgmt Dragon finger Grapevine Hickory icons
IntegrationInterest Interscript Junk
ObjectServerDiscussion Oct17MeetingMinutes Projects
Recruiting Russell Seminars SoftwareForum Squirrel
TechReports Trips Types UGrant UNIX Walnut
WalnutStats WalnutSuggestions Whiteboards WishList

Adding new mail to the database
Salvador.ms delivered 1 message
Msg: Ending backup (file s ...: has been Deleted from Active
```

Copyright © 1985 by Xerox Corporation. All rights reserved.

Figure 1. The Walnut Control Viewer

set can be displayed simply by pointing the mouse at it and clicking one of the mouse buttons.

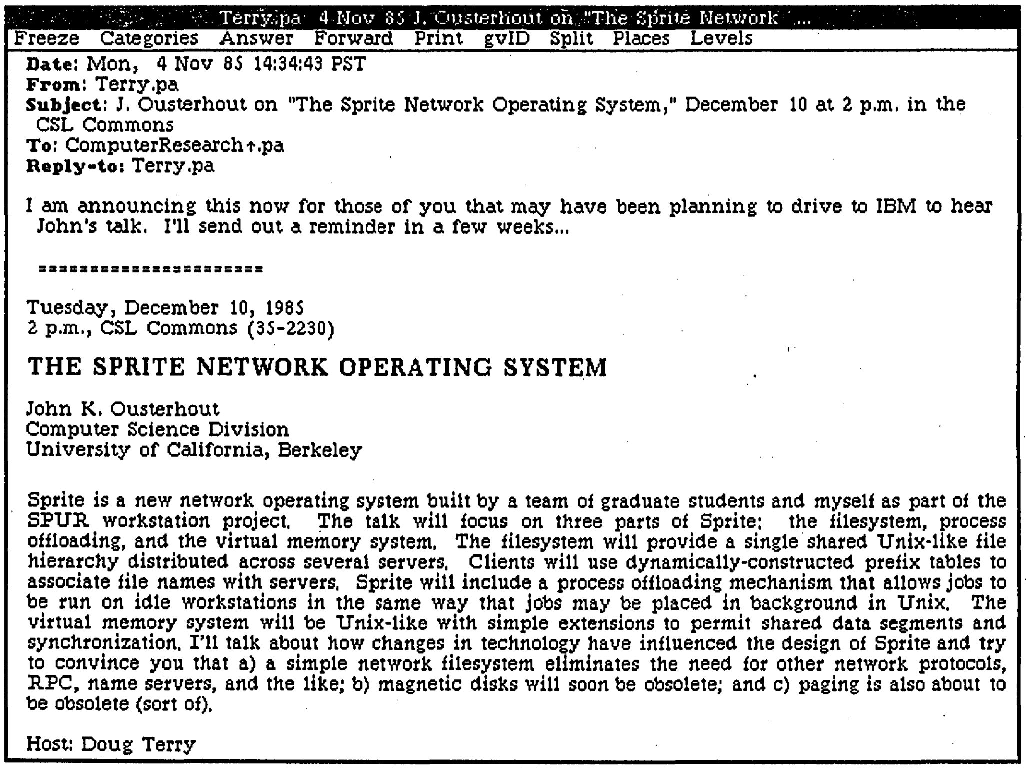
3. A Walnut message displayer, which displays the text of a message (Figure 3). The menu items in the window include operations to generate message-sending windows to Answer or Forward a message, a Print button for producing hardcopy, and operations to manipulate the contents of the message as a normal Cedar document (the "Places" and "Levels" buttons). Note that messages in Walnut can be formatted documents—the mail system uses the same editor as the rest of Cedar.



```
 7 Aug 85   To: CedarUs...      Whiteboard for CedarChest 6.0
13 Aug 85   PeterKessler.pa     WhiteBoardDoc
17 Sep 85   jw@GVAX....         Whiteboards paper
24 Sep 85   To: jw@G...         Re: Whiteboards paper
25 Sep 85   jw@GVAX....         Found it ...
26 Sep 85   jw@GVAX....         Version Stamps and The Promised
                                      Paragraph
26 Sep 85   To: jw@G...         Re: Version Stamps and The Promised
                                      Paragraph
17 Oct 85   To: Barth            Re: Core discussion
17 Oct 85   Spreitzer.pa        Re: Core discussion
17 Oct 85   Spreitzer.pa              Re: Core discussion
18 Oct 85   Barth.pa            Large text boxes on whiteboards
18 Oct 85   To: Barth           Re: Large text boxes on whiteboards
25 Oct 85   To: CedarUse...     Whiteboard for CedarChest 6.0
```

Copyright © 1985 by Xerox Corporation. All rights reserved.

Figure 2. A Walnut Message Set Viewer


Terry.pa 4 Nov 85 J. Ousterhout on "The Sprite Network ...
Freeze Categories Answer Forward Print gvID Split Places Levels

Date: Mon, 4 Nov 85 14:34:43 PST
From: Terry.pa
Subject: J. Ousterhout on "The Sprite Network Operating System," December 10 at 2 p.m. in the CSL Commons
To: ComputerResearch↑.pa
Reply-to: Terry.pa

I am announcing this now for those of you that may have been planning to drive to IBM to hear John's talk. I'll send out a reminder in a few weeks...

=====================

Tuesday, December 10, 1985
2 p.m., CSL Commons (35-2230)

THE SPRITE NETWORK OPERATING SYSTEM

John K. Ousterhout
Computer Science Division
University of California, Berkeley

Sprite is a new network operating system built by a team of graduate students and myself as part of the SPUR workstation project. The talk will focus on three parts of Sprite: the filesystem, process offloading, and the virtual memory system. The filesystem will provide a single shared Unix-like file hierarchy distributed across several servers. Clients will use dynamically-constructed prefix tables to associate file names with servers. Sprite will include a process offloading mechanism that allows jobs to be run on idle workstations in the same way that jobs may be placed in background in Unix. The virtual memory system will be Unix-like with simple extensions to permit shared data segments and synchronization. I'll talk about how changes in technology have influenced the design of Sprite and try to convince you that a) a simple network filesystem eliminates the need for other network protocols, RPC, name servers, and the like; b) magnetic disks will soon be obsolete; and c) paging is also about to be obsolete (sort of).

Host: Doug Terry

Copyright © 1985 by Xerox Corporation. All rights reserved.

Figure 3. A Walnut Message Viewer

Something that is missing from the user interface for Walnut is a more general mail database querying facility than simply being able to display the contents of a message set. A query facility has been built for Walnut, but it is not an integral part of the system (and currently is not widely used). There are two reasons why we did not build a general query processor as part of Walnut:

1. Our first goal was to get the underpinnings of the system right before adding the substantial complexity that a queryer would entail. We were conservative in our ambitions because of the importance our users place on high reliability and performance.
2. More importantly, a good test of the programmer's interfaces to Walnut was whether we could build a query processor on top of Walnut, rather than inside it. Being able to do so made it much easier to modify the queryer as we gained experience with it and made it possible for the Walnut users who did not need or want a query processor to avoid paying any performance penalties. We will discuss below the query processor and the other packages built on top of Walnut.

4. The Walnut Internal Structure

4.1 Files

Instead of storing all of the information pertaining to messages and message sets in a Cypress database, Walnut uses multiple files for storage: a Cypress database file and a collection of log files. Both the logs and the database are stored on Alpine servers. The most important of the log files is the "current log" file, which contains both the contents of all of the messages and a history of all of the operations that a Walnut user has performed (expunging the database compresses this history by discarding entries for deleted messages); thus, the contents of the database can always be reconstructed by replaying the current log. The database contains the current association of messages with message sets and the information necessary to display messages (*e.g.*, the label for a message in a message set viewer and the position in the log of the message body). Additional log files are used to store incoming messages and to execute an expunge operation – the details of their use is given below.

This separation of the information into several files was motivated by several concerns:

1. Cypress, like most database systems, does not handle large varying length strings well. We knew that Walnut would have to handle large messages, so the use of a separate log to store the message bodies was necessary. Messages of approximately half a megabyte have been sent and received by Walnut; the system was designed to handle messages much larger.
2. Initially, there were some worries about the reliability of Cypress; indeed, a few bugs were found in Cypress when Walnut use became heavy. Having a separate log meant that one could always fall back to reconstructing the database rather than having to find ways of recovering data from damaged databases. Reliability was critical in the acceptance of the system; our users were willing to have the system crash if the recovery was simple and certain.
3. Having a log also meant that we had more freedom in changing the database schema, something that we have done during the course of the Walnut development. We can release a new version with a change in the database schema without issuing complicated instructions to our users. Instead, we record in the database a "database schema version stamp" that is checked by each Walnut release; having the wrong version stamp is logically no different than having a mangled database – in each case, recovery is done by replaying the log. (Still, rebuilding databases is not something that we do regularly; most Walnut users rebuild only when forced to do so because of serious hardware or software failures. In particular, we do not depend on rebuilding databases to reclaim file space.)

A "root file" is used to provide a logical connection between the log files and the database. Again, the root file is stored on Alpine. The root file contains the name of the database file and the list of the log files; additionally it records which user may add new mail to the database and a user-specified "key" that is stored in all of the log files and in the database to help guarantee that the logs and database mentioned in the root really do belong together (this key is usually some descriptive phrase, such as "Donahue's mail database").

The fact that the information in a Walnut database is spread over several files is carefully hidden from both the Walnut client programs and (as best we can) from the user. The root file is really the file that describes a Walnut database—it is the responsibility of the low levels of Walnut to interpret the contents of the root file and to make the collection of files described therein behave as if all of the data were stored in a single database.

4.2 Programmer's Interfaces

Walnut makes available three programmer's interfaces. Together, these capture the semantics of a Walnut database as a collection of "abstract datatypes." The lowest level interface, WalnutOps, provides all of the operations that can be legally performed on a Walnut database. The user level interface, WalnutWindow, gives a client program access to the same set of operations that a Walnut user can perform through the buttons on the screen. Finally, the WalnutRegistry interface provides a client program with a history of the WalnutOps operations that are performed so that the state transformations of the database can be monitored.

4.2.1 WalnutOps

The semantics of a Walnut database are *completely* characterized for Walnut clients by the WalnutOps interface. WalnutOps includes all of the operations needed for the Walnut user interface code (such as "get the contents of this message," or "move this message to this message set") and all of the operations that are used to synchronize the various Walnut processes (more about this will be given below). The operations in WalnutOps can be separated into the following categories:

1. *Starting and stopping Walnut*: The StartUp operation takes the name of a root file and initializes the system to operate on the database and log files named in the root; the ShutDown operation shuts down all of the server connections and prohibits all other operations until a subsequent StartUp is performed.
2. *Primitive message set operations*: Given the name of a message set, the operations CreateMessageSet, DestroyMessageSet, and EmptyMessageSet do the obvious things; additional operations are provided to enumerate the messages in a message set, to move messages between message sets, and to add or remove a message from a message set. Although message sets are entities in the Walnut database schema, there is nothing in the WalnutOps interface that commits us to this choice—the interface only uses names. Thus, we could change the Walnut schema to use a different encoding of message sets without changing the abstract characterization provided by WalnutOps.
3. *Primitive message operations*: The only operations currently provided on messages are ones that produce the contents of a message and the message sets to which a message belongs.
4. *New Mail*: WalnutOps contains no operations to retrieve messages from the Grapevine

servers. Instead, it provides operations to obtain a write lock on the new mail log, to mark a portion of the new mail log as information to be preserved, and to copy the contents of the new mail log onto the current log (to fetch new mail). StartNewMail opens the new mail log for writing; EndNewMail operation closes the new mail log. GetNewMail first copies the contents of the new mail log onto the current log and resets the new mail log to be empty, then replays the resulting new "log tail" to add the new messages to the database, and finally returns information about the new messages. Finally, the RecordNewMailInfo operation records in the database the current length of the new mail log. After RecordNewMailInfo finishes, messages that have been written to the new mail log can be flushed from Grapevine—they are guaranteed to eventually be entered into the Walnut database. When restarting Walnut after a crash the new mail log is shortened to the length given by the last RecordNewMailInfo operation (since this is the position at which messages may last have been flushed from Grapevine); truncating the log prevents large numbers of duplicate messages from being created in the event of a crash while reading new mail.

4.2.2 WalnutWindow

For many client programs, WalnutOps is at too low a level; for instance, the Cedar calendar system simply needs to be able to display a named message. WalnutWindow provides program access to all of the user-level operations, such as displaying messages, reading new mail, etc.

The most interesting operation in the interface is QueueCall, which provides a WalnutWindow client with the means of executing a "Walnut user level transaction." QueueCall takes as its argument an arbitrary procedure, which is called atomically—no other concurrent Walnut user activity is allowed. The procedure provided can perform arbitrary WalnutOps operations, so it can change the database in an arbitrary fashion; when the call is finished, WalnutWindow ensures that the current state of the display is consistent with the new state of the database.

4.2.3 WalnutRegistry

Cypress provides no provision for "triggers" to be stored in the database, yet we have found need for other applications to be notified of changes in a Walnut database. For example, we can attach voice annotations to electronic mail; to ensure that the space taken up on a Voice File Server [Swinehart84] is reclaimed when the message to which an annotation is attached is expunged, it is necessary for the voice mail program to track the movement of messages through a Walnut database.

This is done through the WalnutRegistry interface. WalnutRegistry allows a client program to register a collection of procedures to be called when events of interest occur; these events include the starting and stopping of Walnut, the reading of new mail, the movement of messages among message sets, and the creation or deletion of message sets. When a WalnutOps operation is performed that causes an event of interest to occur, the set of procedures registered for the event are invoked.

5. Implementation Problems and Solutions

5.1 Managing Large Objects

The most obvious effect of our concern to handle large messages was the use of a separate log to store the bodies of the messages. This made it possible for us to implement straightforward techniques to manage the storage used for message bodies. Also, this decision also made it easier to build a robust system (for reasons captured nicely in [Lampson84]).

The combination of large size and high volatility makes space reclamation for messages a major concern. Using a separate log file makes it easy to collect unused space by rewriting the log to remove entries for messages that no longer exist in the database. The Walnut expunge code (which does the "garbage collection") goes through the following phases:

1. Destroy all of the messages in the database that belong to the Deleted message set.
2. Read the root file to determine which file is to be the "expunge log." Open the expunge log for writing.
3. Parse the current log and copy to the expunge log each entry in the current log that refers to a message that still exists in the database. When copying a message, update the indices stored in the database to refer to the message's new position in the expunge log.
4. Rewrite the root file to swap the expunge log and the current log. The expunge log now becomes the current log. Set the length of the old current log (the new expunge log) to zero, since its contents are no longer important.

We spent a good deal of time tuning the parsing routines (and designing the format of the log entries themselves) so that expunging is not prohibitively expensive, even though it must scan the entire log. Currently, expunging takes approximately two minutes per megabyte of log (*i.e.*, it takes about ten minutes to reclaim the space on a log of four or five megabytes; there is substantial variation in this figure depending on how busy the Alpine servers are). This is sufficient performance that people can start expunges when they go to lunch or to a meeting and have the expunge completed with certainty by the time they return. However, we also have designed a background collection algorithm that would do the copying as a background process; to handle databases larger than around 20 or 30 megabytes, background collection would be necessary. Below, we sketch some of the basic ideas of how this could be done.

The first idea is to generalize the single "current log" of Walnut to a collection of log files that together represent the current log. This would allow all of the log files except that last one to be rewritten by a background garbage collector without having the collector attempting to rewrite the file being used to log user actions. The format of the "root file" was designed with this in mind: while we currently only use entries for the current log and the expunge log, we could use the same format to describe a sequence of log files.

The hardest part in doing background operations of this sort is to minimize the chances of conflict between the background process (which will have its own transaction) and the normal processing. To do this for expunging, we split the "copy and update" step of the current expunge algorithm into separate copy and update steps; the destruction of the messages in the Deleted message set is still performed synchronously, but the cost of this operation is determined by the size of the Deleted message set rather than the size of the entire database. The copy step parses a log segment (one or more log files) and produces a compressed version of it—copying needs to read the database but does not write into it. And the reading of the database does not need to be done under the copy transaction, since a "fuzzy dump" is adequate; we only need to ensure that all existing messages get copied. Thus, this copying can use the "normal processing" transaction when reading the database; using the same transaction as the normal processing guarantees that no transaction conflicts can occur with the user's activities.

The update step is then performed. The compacted log is again parsed and each time a message entry is found in the log, the database is updated to point to the new location. Here the updates need to be done atomically, so it is possible that there may be conflict with normal processing. However, as we discuss below, these transaction conflicts can be effectively hidden from the user. Finally, the length of the old log file is set to zero and the root file is rewritten to make the old log file the new "expunge log." (This avoids the cost of a file creation when a log compression is begun.)

Managing large objects also involves another, less obvious, problem: if the objects are very large, even simple operations can no longer be performed in a single transaction. For example, the GetNewMail operation of WalnutOps copies newly retrieved messages from the new mail log to the current log; even if the new mail log contains only a single message, it may not be practical to perform the copy in one transaction (the cost of redoing the copy if the transaction aborts may be too high to be practical). The major technique used in Walnut to ensure that the system can be used to manage large messages (as well as large numbers of messages) is to break up any potentially long-running operation into a sequence of constant-sized pieces such that the transaction can be committed after execution of one (or more) piece.

Breaking long transactions into small pieces is an idea that is widely used in high-performance systems such as airline reservation systems (for instance, [Gifford84]), but it came as a surprise to us that an electronic mail system would have the same requirements! One corollary of breaking all operations into constant-sized chunks is that Walnut must do its own crash recovery. A crash may occur during a long-running operation that has partially completed; since one or more transactions may have committed during the operation, it is necessary to complete the operation before further processing can be performed. So when a long running operation (like copying the new mail log) is begun, it is logged and recorded in the database; the progress of the operation is also recorded as transactions are committed that complete portions of it. (The progress need not be recorded each time a transaction commits, but only frequently enough so that substantial amounts of work are not lost.) When a restart is performed following a crash, Walnut ensures that any in-progress operation is completed before control is returned to the higher levels of the system that handle user requests.

5.2 Caching and Transactions

The serializability guarantee provided by any transaction system is a weak one: if the transaction commits, then it must have been serializable with other activity in the system. The user would like to believe that serializability is sufficient to prevent aborts; in particular, if the user is manipulating a private database, he generally never expects to see an aborted transaction. But there may be many conditions under which the transaction system may abort a transaction for reasons other than interference with another transaction:

1. The server may recover from a crash by aborting all uncommitted transactions,
2. Since Alpine pages do not correspond nicely to Cypress entities and relationships, Alpine may abort a transaction that is logically independent of other activity, but happens to manipulate the same physical page as another transaction.
3. The server may time out an idle transaction by aborting it.
4. The current Alpine backup system aborts all transactions on a file before copying it.

While the page-level transactions of Alpine make it somewhat more likely that a spurious transaction abort will occur, in any system there will be some cases in which a transaction will be aborted even though there was no interference. In a system such as Walnut, where most of the time users are browsing essentially private databases, it is important to handle aborted transactions to minimize the possibility that spurious aborts will be reflected back to the user.

The other problem that transactions cause for systems like Walnut involves cache management. The collection of messages, message sets, and message set buttons that are displayed is a large cache of data extracted from the database; when a transaction is aborted (for whatever reason), this cache must be revalidated (at least) or recomputed (at worst). Since Walnut cannot control how large this cache is (we can't limit the number of message sets that the user has displayed), we must ensure that the cache revalidation is extremely efficient. It doesn't do to recompute the contents of the screen when an aborted transaction occurs.

Walnut uses version stamps on each message set and on the set of message sets (the buttons that appear in the control window) to ensure consistency between the display and the database. The user interface code maintains a set of "expected version stamps;" these are checked for consistency with the database when operations are performed that involve a particular message set (such as moving a message from one message set to another) or the set of message sets (such as adding a new message set to the database). In this way, we ensure that the effect of operations as displayed on the screen are consistent with the current contents of the database. However, changes to the database that invalidate the contents of the display are not noticed until some operation is performed that causes a version stamp to be checked—since most databases are private, it is generally the case that this is not a problem.

An aborted transaction in Walnut is just taken as evidence of *possible* interference; interference is *certain* only when the expected and actual version stamps disagree. If execution of a WalnutOps operation causes a transaction abort and the version stamps agree after opening a new transaction, then the operation is simply retried—if it succeeds, then the transaction abort must have been spurious. It is possible that this retry logic on two machines will produce a "recovery deadlock," where each machine attempts to recover, again causing aborted transactions; however, we avoid this by giving up if the second transaction is aborted—if this happens, the chances that the interference is real are very high. This scheme works quite well; it is also used in the Whiteboards application [Donahue85b], where public databases are more common.

It is also important to note that the version stamps used in Walnut cover many different pieces of many different relations; there is no simple mapping from the underlying structure of the database to the collection of version stamps maintained. This suggests that a database system that provided logical locking at the level of an individual relation or tuple (a Cypress "relationship") might be as difficult to deal with as the page-level locking of Alpine. For instance, it is important that the system be able to change message pointers without holding logical locks. An attempt to read a message from the database might fail because another transaction is changing the message pointers in the database, but this is not a transaction abort that should be reflected back to the user. Message pointers are not part of the user's view of the database—they are simply an artifact of the implementation that should be hidden if at all possible.

Using the version stamps to ensure cache consistency means that we can also close the transaction during long idle periods; we do not need to hold a transaction open to ensure the consistency of the cached data being displayed. The program that implements the WalnutOps interface contains a simple watch-dog process that checks for long idle periods (currently five minutes) and performs a shutdown of the system if it is idle. This shutdown is completely transparent to the WalnutOps clients; when the next operation is executed, the system performs a normal startup before proceeding. For an application like Walnut that is idle most of the time, this is particularly important; it means that the utilization of the resources of the server is determined solely by the active Walnut users, rather than by the (much larger) number of users that currently may have a portion of a Walnut database displayed on their screen.

5.3 Integrating Applications

We currently have three fairly well-developed database applications running in the Cedar environment. The first is Walnut. Whiteboards provides an electronic equivalent of the whiteboards and corkboards that we have in our offices [Donahue85b]. Finally, Finger manages a database of information on machines and users in our laboratory. We also have developed a number of applications programs that run on top of Walnut: the two most important are the Walnut query processor mentioned above and a program that automatically sorts incoming messages into message sets according to user-given classification specifications. Additionally, Finger uses WalnutRegistry to record when a user reads his mail and several programs use WalnutWindow operations to display data stored in a Walnut database. This experience has taught us some important lessons about how databases affect applications development.

One of the hopes of the design of Cypress was that storing things in a database would make the integration of applications easier. The Cypress report [Cattell83] argues that the introduction of a common data model should simplify sharing of data among multiple applications. The general belief of the relational database community seems to be that the simplicity of the relational model and the elegance of relational operators as a common query language should make it very easy for many programs to share a common database. Unfortunately, our experiences with integrating the applications that we have developed, in particular, layering other programs on top of Walnut, argue that this is not quite true.

The belief that a common database query language can integrate a collection of applications seems founded on the premise that the structure of the database is semantically significant. The "universal relation" approach of Ullman [Ullman82] is one attempt to hide some of the underlying structure of a relational database by allowing the user to specify relational queries as if there were a single "universal" relation in the database. As Ullman notes, this presents an attractive user interface by removing some of the necessity to deal with the underlying logical structure, which may have been chosen for reasons other than ease of understandability. Working on Walnut has shown us two other reasons why the logical structure of the database is not a reasonable basis for a query processor; basically the database may contain both too little and too much information for standard relational queries to be of much use.

First, the logical structure of the Cypress databases built by Walnut does not contain one of the most important parts of an electronic mail database: the messages themselves. These are stored in separate log files. And there are good reasons not to store the messages in the database: space reclamation is easier and the system is far more robust in the face of failures of the database software. But, if one were to apply a relational query language directly to a database built by Walnut, the connection with the log would be missing—which would make the query results rather uninteresting!

Second, as we described above, a Walnut database also contains information about the progress of long-running operations. The database is a natural place to store such information; however, it is of no utility to the normal Walnut user and should be carefully hidden. In fact, if the database currently holds information about an operation in progress, then the database state may be inconsistent and the entire database should be protected until the operation completes.

What we have done in Walnut (and our other database applications) is to define a programmer's interfaces that give a characterization of the database as an abstract datatype. The WalnutOps interface hides the details of managing a separate log and database, but presents the client with a view of a database containing messages and with no special handling of long-running operations. WalnutOps *completely* defines the structure of a Walnut database (comprising both a Cypress database and a collection of log files) as far as client programs are concerned. The use of a programmer's interface, rather than a query language that directly manipulates the structure of the database, has several important benefits:

1. The operations exported by the interface are all defined to preserve the invariants that Walnut depends on, rather than using lower-level operations and run-time checks to ensure

consistency. In some cases this means that we can use the type-checking of the Cedar programming language to do invariant-checking, with the obvious performance gains of static over dynamic checks. More importantly, some of the invariants upon which Walnut depends are very expensive to check. It is far more efficient to prove that:

a. the WalnutOps operations preserve the invariant "every message belongs to one or more message sets; if a message belongs to no other message sets, then it belongs to the Deleted message set," and that

b. only WalnutOps operations are used to manipulate the database

than it is to check the database for the truth of the invariant. (It is possible in Cedar to subvert the WalnutOps interface, but it is extremely unlikely that this would be done—it's very hard to do inadvertently and Cedar programmers believe in the importance of interfaces strongly enough to ask for interface changes rather than trying to access unsafe lower-level procedures.)

2. The interface can hide the details of transaction management; when writing cooperating applications this is an important benefit. Except in cases where separate programs must commit changes independently, it is better to share a common transaction than to have to worry about inadvertent conflict caused by using separate transactions. WalnutOps hides transactions inside the implementation—interference (*i.e.*, the notion of why a transaction might abort) for WalnutOps clients is reflected in the mismatch between expected and actual version stamps. (Separate transactions are used when background new mail processing is being performed, but there is no chance for conflict since new mail is written to a separate log file.) For applications that require a sequence of WalnutOps operations to be performed before another user request is serviced, the WalnutWindow interface provides a procedure that allows "Walnut user level transactions."

3. Providing a programmer's interface also makes it relatively easy to implement "triggered procedures" safely and efficiently; we have done this for Walnut in the WalnutRegistry interface. As we described above, WalnutRegistry allows client programs to register procedures to be called when events of interest occur; when a WalnutOps operation is performed that causes an event of interest to occur, the set of procedures registered for the event are invoked. Because WalnutOps is the programmer's interface to the database and each WalnutOps operation preserves the database invariants, there is no way a procedure registered with WalnutRegistry can destroy the invariants of a Walnut database. Also, we can use the process creation mechanism of Cedar to ensure that the procedures called when an event occurs cannot hold up normal processing, either because they take a long time to execute or (even worse) go into an infinite loop. The implementation of WalnutRegistry uses a rather complicated process structure to guarantee that the procedures registered by WalnutRegistry will be called in the order in which the events for which they were registered occurred (this might not be true if a new process were simply forked for each procedure

call); however, because the invocation of a procedure registered with the interface may be delayed, no guarantees are made about the contents of the database at the time of the call.

4. The final benefit of building a programmer's interface as a collection of procedures is that it makes it easier to change the structure of the underlying database without changing the applications that access the database – for precisely the same reasons that one uses a database schema rather than coding access paths into programs. We have taken advantage of this several times in the development of Walnut; the low-level components of the system have been reworked many times without having to change the upper-level pieces. The WalnutOps interface took a long time to design, but has served as a very important fixed point in the development of the system.

One open question is how to connect several applications written in this fashion. One of the useful properties of building query processors at the database schema level is the uniformity of structure that using schemas provides – there is much less uniformity in a collection of programmer's interfaces designed by several individuals. How to build query processors that span applications like Walnut with more traditional database applications (where the schema may reflect the semantics of the database) is an interesting question for future research.

5.4 Background Processing

One of the major successes of Walnut has been its use of background processing to retrieve new mail. The implementation of background new mail, though, turned out to be harder than we anticipated; our experience identified two problems that need to be considered when designing such systems: isolation and robustness. We discuss each of these below.

5.4.1 Isolation

Background processes need to be carefully isolated from normal processing; the foreground and background activities should require very limited synchronization and have no chance for transaction conflicts. The new mail process of Walnut synchronizes with the normal processing at three points:

1. When the new mail log is opened. A call is made to WalnutOps to open the log and the WalnutOps transaction is used to read the root file to determine which file to open. The file is opened, however, under a new transaction; while new mail is written to the log, it is impossible for any conflicts between this transaction and the WalnutOps transaction to occur.
2. After the mail from a server has been completely read. A call is made to WalnutOps to record the information about the current length of the new mail log. Again, the WalnutOps transaction is used to record the information in the database; the new mail process uses the completion of the call to WalnutOps as the indication that the information has been successfully written. This is also a cheap operation, so the time necessary to synchronize

the two processes is very small (normal processing is not held up for long while this synchronization occurs).

3. When the servers are all emptied: A call is made to WalnutOps to "return" the new mail log. This is done because the implementation of WalnutOps also maintains a copy of the open file "handle" to the new mail log; if the user requests new mail while a background retrieval is underway, we can abort the retrieval (by aborting the transaction being used to write on the log) and then returning whatever mail happens to exist of the new mail log from previous retrievals. Our users come first; we prefer to give them what we have rather than make them wait for (possibly) more mail. "Returning" the new mail log causes the log file to be closed and the open file handle to be discarded.

The ability to stop reading mail at any time by aborting the new mail transaction points out the second concern we had when designing background retrieval—that it be extremely robust in the face of failures.

5.4.2 Robustness

One of the responsibilities of implementing a background processing algorithm is to make the user unaware of its existence, except that things sometimes go much faster than he might expect. In particular, the implementor must assume that the user cannot be relied upon to avoid doing nasty things (like rebooting his machine) during sensitive periods. The most difficult part of getting this right for Walnut was to ensure that the mail was flushed from Grapevine only when it was certain that the database recorded its existence on the new mail log. Since this could not be done under a transaction (there is no means for allowing Grapevine and Alpine to use a common transaction), a protocol was designed to ensure that the necessary operations would happen in the right order and the chance of unnecessary duplication of messages (messages appearing both on Grapevine and in the new mail log) was acceptably low and would be properly handled. Having a "transaction server" that could be used by all network resources to agree on transaction commits or aborts would have made the robustness of this part of Walnut much easier to guarantee.

6. Conclusions

Walnut is definitely an atypical database application; yet systems like Walnut will become increasingly important as databases become a more integral component of our computing environments. One conclusion to come out of this work is that there is less difference between database applications and database systems than seems obvious at first glance. Walnut uses many of the same techniques that appear in Alpine or Cypress: logging, caching, version stamps, and crash recovery are obvious examples. This is an example of Lampson's dictum: *Use a good idea again* [Lampson84, pg. 18]. One natural reaction to this is to claim that the database system must be poorly designed for so many things to have to be duplicated. However, much of Walnut's complexity

XEROX

Walnut: Storing Electronic Mail in a Database

James Donahue and Willie-Sue Orr

CSL-85-9