The Jaquith Archive Server

James W. Mott-Smith

Abstract

Advances in robotic devices and storage media now make it possible to design *near-line* automated storage systems. These systems aim to provide responsive performance to users of tertiary storage devices. The Jaquith system is a prototype archive server that lets network users archive their own files using automated storage. It provides semi-interactive file access to its clients by combining a high-density robotic tape system with disk-based indexing.

Jaquith presents an FTP interface whereby whole files are moved between the client and its storage archive. Each client's archive is separately governed to provide independent namespaces, added security, and parallel operation. A wildcard query mechanism lets users manipulate arbitrary subsets of their files. Two important aspects of the query system are *abstracts*, text tags that can be associated with files, and *versions*, date-stamps that are applied to archived files.

Jaquith throughput is about 135 KB/second when archiving small (10 KB) user files to disk buffers. The use of synchronous disk writes by the server to ensure durability of each user file degrades throughput to 40 KB/second. The performance when writing disk buffers to Exabyte or Metrum tape is severely limited by the time to write a hardware filemark. Consequently, it is important to write several megabytes of data between filemarks for good performance.

1 Introduction

The recent coupling of inexpensive but powerful computers with networked file systems has dramatically increased the amount of information being stored and manipulated digitally. As the quantity of data increases, it becomes more difficult to keep it organized, to locate specific items, to avoid duplication, and to perform data synthesis.

This report describes Jaquith,[1] a prototype tertiary storage manager that combines disk and high-density tape devices to address the "data glut" problem. Jaquith's storage facilities let clients move files from disk to tertiary storage without having to know the location or physical attributes of the storage device. Jaquith is written in **C** and runs on most Unix systems. The code runs as user-level processes which has simplified the port to multiple platforms including SunOS 4.1.1, Ultrix 4.2, and HP-UX 8.05.

The design of a tertiary storage manager is dominated by two issues, the masking of hardware latency and the user model. Latency is the single most difficult aspect of tertiary storage management, and Jaquith has several features specifically designed to address this issue including on-line indexing information, an intelligent buffering scheme, a structured tape layout, and user-supplied file tags, called *abstracts*. Together, these features reduce retrieval time for data stored in a tertiary archive.

Jaquith's user model provides simple but powerful access to archive data. Users move whole files between secondary and tertiary storage with specialized utility programs. A wildcard query facility enables the user to locate and recall specific versions of files, directories, or complete subtrees. An **X**-based graphical user interface makes the query facility easy to use. Jaquith is an archiver so it does not have a delete facility.

This report first motivates the Jaquith design with a discussion of the current generation of hardware and the related latency issues. Latency constraints drive the discussion of the user model and the client interface in Section 3. Section 4 describes the structure and implementation of the server including indexing, buffering, and administration facilities. Section 5 details Jaquith performance. Related work is covered in Section 6. The report finishes with a discussion of Jaquith as a base for future work.

[1]The work described here was supported in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

2 Background

A current problem in networked computer systems is the storage of massive quantities of information, where "massive" may mean anything from many gigabytes to several petabytes, or more. Presently, secondary storage such as magnetic disk is too expensive to hold this data, so tertiary storage is essential. Of the common tertiary media, helical-scan magnetic tape is the medium of choice because of its high density and low cost. For example, an Exabyte drive costs approximately $2000 and a single tape cartridge (a *volume*) costs $6 and holds five gigabytes of data.

The drawback of these tape systems is their low transfer rate and long load delays (the time to mount the volume and locate the beginning of tape). Long term reliability is also an open issue, see [10]. Ampex D-2 tape systems combine high-density storage with very high throughput rates, but carry very high prices that make them more suitable for specialized purposes. Other forms of tape, such as IBM 3480 cartridge tape have transfer rates comparable to disk but hold only 200 MB of data per volume.

The primary goal of the Jaquith project is to overcome these hardware latencies and develop a testbed for automated tape storage systems. The interesting question is: How does one design a robotic tape system with current hardware that provides useful *near-line* storage? The term *near-line* means that it is important to support access rates that are much superior to *off-line* shelf storage, though not quite on par with *on-line* disk storage. A related question is: How do users employ near-line tape storage? Do they view it as extended disk space, temporary swap space when disk space is tight, or as archival storage? The Jaquith testbed was created to help answer these questions. The first implementation of Jaquith stresses simplicity, flexibility and portability. Experience with the testbed will give insight into usage patterns and provide guidance in extending the testbed with advanced aspects of massive storage including indexing techniques, data striping, interactive browsing, and file migration.

The design of Jaquith was influenced heavily by the characteristics of the available hardware. The primary hardware environment for development of the Jaquith system was an Exabyte EXB-120 Cartridge Handling Subsystem. The EXB-120 stores one half terabyte of data in 8mm cassettes arranged in an X-Y plane. Near the end of the development cycle a Metrum robot was acquired. It stores about nine terabytes of data with VHS cartridges placed around two rotating cylinders. Table 1 gives various specifications for both robots and tape readers. See [5, 4, 3, 6] for more information. While the two systems differ in detail, they have the following important

characteristics in common:

- The combined latency to load a tape volume into a reader, seek to the proper location, and read a file is measured in minutes. Unlike disk accesses which are measured in milliseconds and are unnoticed by humans, every single tape access will be visible to the user. Therefore it is important to avoid any extraneous tape accesses.
- The total number of volumes is low and the capacity per volume is high. Therefore data from different sources must be packed onto a single volume. It must be possible to quickly locate a piece of data without reading all of the intervening tape.
- The throughput of a single reader is low enough that the time to transfer a large buffer of data is significant. The latency to transfer data from tape to disk is noticeable by the user.
- The robot arm can load any of the tapes into any of the readers. Jaquith can thus ignore issues of physical partitioning. (In contrast, the Comtex ATL-8 uses a compartmentalized arrangement whereby the archive is divided into carousels, one per drawer. Tapes can only be loaded into drives residing in the same drawer as its carousel.)

	EXB-120 with EXB-8500	Metrum RSS-600 with RSP-2150
Max. number of robot arms	1	1
Max. number of tape readers	4	5
Max. number of tapes	116	600
Tape capacity in GB (type)	5 (8mm)	14.5 (S-VHS)
Throughput per drive (MB/sec)	0.5	1.5
Time to write filemark (sec)	2	8
Time to pick & load tape (secs)	35	16
Time to rewind longest tape (secs)	180	120
Approx. robot price (US$)	90,000	275,000
Approx. reader price (US$)	2,000	35,000
Approx. cartridge price (US$)	6	8

Table 1: Specifications for two robotic tape systems. Important common characteristics: very long user-visible latencies, large capacity, and very low cost per byte.

3 User Model

This section describes the Jaquith user model. The model defines how the user will conceptualize the storage server. The important features are described below: whole-file operations,

versions, wildcard queries with abstracts, and logical archives.

3.1 Overview

To the user, a Jaquith archive is a series of snapshots of the disk file space taken at various points in time. Snapshots are created by "putting" whole disk files to the archive with the `jput` utility and retrieved by "getting" them with the `jget` utility. The utilities never manipulate pieces of a file.

The disk name space is preserved in the archive by the snapshot operation. Files are stored in the archive with their original disk file names and the hierarchical relationship between directories and their files is maintained.

The snapshot operation creates a new version of a file (or directory) in the archive which is distinguished by a time stamp. Previous versions are not replaced, but remain accessible. No data is ever deleted from the archive.

Figure 1 shows a small disk namespace and a corresponding Jaquith archive. The two namespaces are largely identical, but there are multiple versions of several files in the archive. There are also files in the archive that have been deleted from disk, and files on disk that have never been archived.

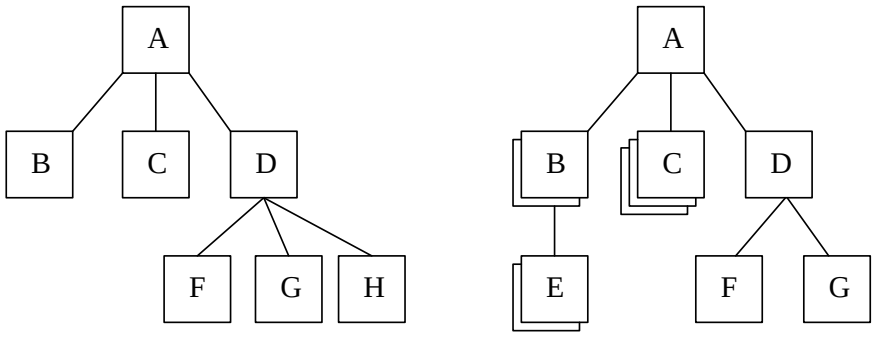

Figure 1: Jaquith user model. On the left is the current filesystem for a small disk. On the right is the corresponding Jaquith archive. The archive has multiple versions of directory `B` and file `C`, and no copy of file `H`. File `E` has been deleted from disk, but old versions are still present in the archive.

3.2 Whole-file Operations

A fundamental decision was made to move whole files between the user and the archiver. The interface to the user is "put" and "get", not "open", "read", "write" as in a typical disk file

system. This model is not only easy to implement and understand, but adapts well to the hardware and software environment.

Moving whole files to tapes keeps tape motion operations to a minimum and increases performance. Complete files are written in contiguous blocks onto the end of the tape, causing no unnecessary tape motion. In contrast, a partial-file model would write parts of files onto the end of the tape in a piecemeal fashion. This would result in fragments of a single file being spread across a large span of tape, or worse, across several tape volumes. Retrieving the data would then require multiple seek (or load) operations to reconstruct the file, and would pay a very high performance price. Table 1 shows the large time penalties for performing tape seek and load operations.

Furthermore, Unix file access patterns are well suited to a get/put scheme. Measurements have shown that Unix files often have sequential access patterns, and are frequently read in their entirety [7, 19]. These patterns suggest that our simple whole-file access model will work well in this environment.

While a whole-file interface is natural at the tape level, it is not the only possibility at the user level. One possibility is to use the whole-file tape transfers as a base on which to build an NFS-like filesystem interface. A migration mechanism would move whole units from tape to disk where they would be modified by **read** and **write** commands. Modified files would be flushed back to tape as whole units when the disk cache became full. Such an NFS interface has two drawbacks: it cannot support multiple file versions and it makes latency unpredictable to the user. Jaquith's whole-file interface provides multiple file versions and consistent performance.

Additionally, an NFS interface with a successful migration policy should do intelligent prefetching, construct file summaries for quick browsing, and be integrated with the kernel. The first two item are research topics beyond Jaquith's scope (see for example [11]), and the last requires system dependencies (mostly in the file descriptor to know whether a file is on disk or not) that we wanted to avoid.

3.3 Wildcard Queries

Two utility programs, `jput` and `jget`, are used to move files between the archive and the disk. `Jput` takes as arguments a list of file names and copies the disk files to the archive. `Jget` takes a list of file names and copies the files from the archive to the disk, restoring their original disk names.

When retrieving information with `jget`, it is possible to specify the files to be retrieved

using several different selection criteria. The criteria are logically ANDed together, so only files that match all the criteria are retrieved from the archive. `Jget` requests are called *wildcard queries* and support four types of selection criteria on the command line. The following discussion describes each of the criteria with example queries.

- File name pattern
- Date specification
- Version numbers
- User-specified text tag (abstract)

The basic retrieval criterion is the file name pattern. Jaquith file name patterns follow the Unix *globbing* model as described in the reference manual pages for **csh**. In brief, globbing is a weak form of pattern matching where a single character in the pattern can represent many characters in the actual file name. For example:

```
jget example1
```

retrieves the file `example1` and

```
jget example?
```

retrieves all files that have an 8 character name beginning with the 7 characters 'example'. Jaquith uses Unix globbing instead of the more powerful regular expression pattern matching because the shell languages have made it the defacto standard for file name matching in the Unix world.

In addition to file name patterns, wildcard queries can give date specifications. Date specifications are used to restrict retrievals to specific date ranges. While the previous example retrieved the latest (most recent) version of `example1` in the entire archive, the following query retrieves the version that was current as of July 12th 1990:

```
jget -asof 12-jul-1990 example1
```

and this query restricts the range even further:

```
jget -range (1-jul,12-jul-1990) example1
```

Older versions of a file are retrieved with *version numbers*, positive integers that specify the first and last versions to be retrieved. Any range of versions can be retrieved with this facility. For example, the oldest version can be retrieved with

```
jget -first 1 -last 1 example1
```

and all versions except the first can be retrieved with

```
jget -first 2 -last -1 example1
```

The '2' indicates the second copy of the file, counting from the front of the list of versions, and

'-1' indicates the first copy counting from the rear of the list. With this notation users can refer to the end of the list without having to know how many copies currently exist in the archive. This is particularly useful for writing retrieval scripts that must run under different archive conditions.

Files are not assigned permanent version numbers since we believe that they would have little meaning to the user. Who can remember what version `example1` had when it was archived last July? Instead, we think that users will browse the archive using the other selection criteria and then request a particular version from the resulting file set. Consequently, a wildcard query is processed in two steps. A set of files is produced ignoring the version numbers, but that matches all the other selection criteria. Then a version number is logically assigned to each file in the set and the version specifiers are applied to produce a final set of files. For example:

```
jget -range (1-jul,12-jul-1990) -first -2 -last -1 example1
```

restricts the date to the specified range, producing a set of candidate files, and then retrieves the last two versions from that set, if they exist.

3.4 File Abstracts

In addition to the file name, date and version selectors already mentioned, `jget` has an additional selection criterion, called an *abstract*. An abstract is a user-specified tag that is attached to each file or directory as it enters the archive. The abstract is free-form human-readable text; it need not have any particular structure (though it is limited to 16 KB) and is not interpreted by the system. Tagging files with meaningful abstracts at archive time makes it much easier to browse the archive at a later time. For example:

```
jput -abs "version 1.0" source.tree
```

archives the user's source tree, tagging each file with the abstract "version 1.0". A more powerful use of abstracts is

```
jput -absfilter file source.tree
```

which invokes the Unix `file` utility to generate a tailored abstract on the fly for each file as it is archived.

With tagged items in the archive, users can retrieve arbitrary subsets of their data using regular expression pattern matching. (See the Unix documentation for `grep` for a complete description of regular expression metacharacters.) Using regular expressions makes it easier to do "approximate searches" in the archive. For example:

```
jget -abs "[Vv]er*1.0" source.tree
```

retrieves the version of the user's source tree that was archived in the example above.

Using abstracts with regular expression searching is preferable to other forms of "content searching". The common alternative is to keep the first N blocks of the file on disk at all times. For many files the front of the file is not a good indicator of the file's contents or version. Also, storing 4 KB per file consumes massive amounts of disk space; abstracts are smaller and consume no space when they're not needed. Abstracts are more flexible than a fixed-block system. For example, a group of previously unrelated files can be assigned a common abstract that effectively ties them together for easy retrieval.

Jaquith has an additional utility, `jls`, with all the features of `jget`, that retrieves only a file's descriptive information (*metadata*), not its contents. The `jls` command mirrors the Unix shell's `ls` command and helps users locate particular files or versions of files before actually restoring them with `jget`.

3.5 Logical Archives

Jaquith supports the partitioning of the physical archive into disjoint domains called *logical archives*. Conceptually, a logical archive is a private storage area within the physical archive with its own name space. Files in one logical archive are completely independent of those in another archive. In fact, the same file may exist in several different logical archives simultaneously. wildcard queries are always directed to a single archive; they do not span archives. Physically, logical archives maintain their own set of volumes within the jukebox. Once a volume has been allocated to a logical archive, it is never retracted or shared.

Partitioning the physical archive into logical archives has several benefits. A logical archive is an administrative unit. All files related to a specific person, group, or project can be grouped together into a a single archive which has its own set of tapes. This makes it possible to remove the group's data from the physical archive when the project ends or the user leaves the organization.

The enforced locality also makes searches through the index faster and reduces the number of tapes mounted and scanned. For example, retrieving all versions of a file which have been archived over a long period of time is likely to be much faster with logical archives than without.

The logical archive is also a valuable protection unit. Jaquith does record protection on a file basis but this is not sufficient. For example, a public logical archive can be made private

by the Archive Administrator without having to worry about the permissions of individual files. As a second, more severe, example consider a situation where nightly system backups are done to one large, public archive. (If the archive is not public the tape system is not interactive because users must still request operator intervention to retrieve their own file.) If a user accidentally leaves a private file unprotected for even a short period of time during which it is archived (with that protection), it becomes publicly readable for all time. Large disk-based systems do not suffer from this problem because the security window can be closed as soon as it is detected. With write-once media the security hole is permanent. Logical archives help solve the problem by letting scripts backup files to specific private archives. Full system dumps may also be done to a secure logical archive as a fall-back.

3.6 The X-based Browser

This section describes an alternative client interface to the Jaquith system called `xjaq` for X-Jaquith. The motivations for a second interface are to expand Jaquith's appeal and to experiment with the Tcl/Tk [17, 18] interface-building tools.

The `jget`, `jput`, `jls` programs described in the previous section, along with the status utility `jstat` comprise Jaquith's command line interface. A command line interface is a powerful tool for advanced users who are interested in creating automated script files, but it is a daunting obstacle to casual users. Casual users will gladly trade some flexibility and power for simplicity and guidance.

`Xjaq` is a browser utility that runs in the `X` environment. It merges most of the functionality of the `jget`, `jput`, `jls` commands into a single program. Whereas the command line interface is based on the concept of a *query*, `xjaq` is based on the concept of a *view*. The query interface marshals all its arguments, scans the archive, and gathers a set of (potentially disparate) files which match the criteria. In contrast, the viewer interface presents the user with a view of the current working directory within the tertiary namespace. From this current location, the user can move up and down through the tree hierarchy, viewing one directory's worth of files at a time. Figure 2 shows the main `xjaq` viewer screen.

The details of any particular view can be customized by the user using buttons, pull-down menus, and other graphical interface elements. For example, the quantity and format of information displayed for each file can be tailored by the user. Also, the set of files displayed in the directory viewer can be restricted by the use of various *filter* parameters, including date, owner, filename

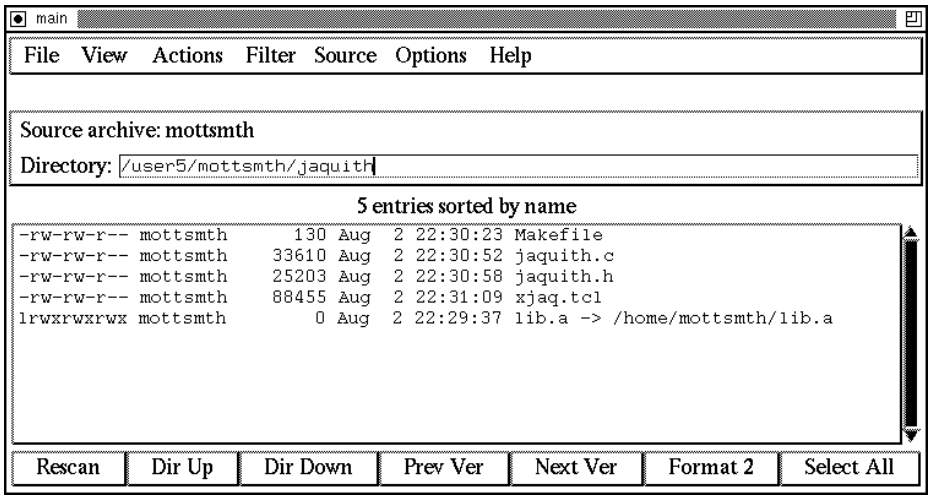

Figure 2: The main xjaq viewer screen. The scrollbox of files is in the center of the window. Along the top is a series of pull-down menus in the Motif style. At the bottom is an optional "speed-bar" of useful functions. The Actions menu provides access to `jget` and `jput` features.

extension, etc. Filter parameters mask certain files from view, thus simplifying the display. Setting the filename filter to ***.tex** would restrict the viewer to TEX documents.

In addition to the main view screen, there are secondary screens for invoking backup (`jput`) and restore (`jget`) functions. Users can browse through tertiary store, select some interesting files with the mouse, and then invoke the restore function with a button click.

Because the directory-at-a-time viewport is sometimes too restrictive `xjaq` also includes a separate *find* screen for large-scale searching. Wildcard queries issued from this screen will generate a list of results in the *find* window, independent of the main viewer.

The browser interface is simpler than the command line interface. While it has slightly restricted functionality, and relies on the command-line utilities to carry out Jaquith operations, it enjoys all the benefits of the **X** environment including display independence, point-and-click operation, and customization.

4 Server Design

The following sections describe the main components of the Jaquith system starting with the process architecture and finishing with the major subsystems. With the exception of the physical tape layout, the design has not been specialized for use with a tape robot. An optical jukebox would work as well or better.

4.1 Overview

The overall Jaquith system structure is shown in Figure 3. From right to left, the three main divisions are: the client programs, the Jaquith server, and the jukebox manager. The three parts communicate using the Unix socket mechanism [1].

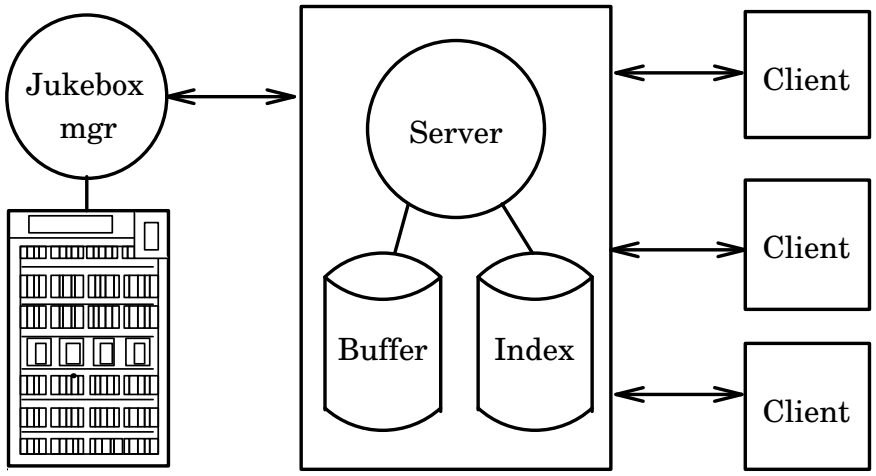

Figure 3: The Jaquith process structure. From right to left: Users run the clients programs `jget`, `jput` at their workstations. The server spawns processes to handle client requests which perform indexing and buffering. The Jukebox manager is responsible for physical device allocation.

Users at workstations on the network run the client programs `jget`, `jput`, and `jls`. These programs request service from the central Jaquith server by connecting to a public socket. In fact there may be several Jaquith servers on the net so the client must choose the appropriate one. Clients are ignorant of all other Jaquith details including the nature of the physical archive, the volume number where their data is stored, and the other users of the archive.

The Jaquith server, in the middle of Figure 3, is responsible for answering client requests by spawning a handler process: each `jput` request generates a *writer* process; each `jget` and `jls` request generates a *reader* process. The handler processes share the local disk for buffering user data and for managing the on-line index. A *reader* process converts a user file name to a buffer number using the index and then extracts the file from the buffer, bringing the buffer from tertiary storage into the buffer pool, if necessary. A *writer* process appends new user data to the current buffer and updates the corresponding index file entry. Independent *cleaner* processes write full buffers to tertiary storage, see Section 4.3.

The Jukebox manager is the back-end of the Jaquith system. Its primary purpose is to control access to the shared Jaquith resources: robot arm, volumes, and volume readers. A Jaquith

process that needs access to tertiary data issues an RPC to the Jukebox manager, which enqueues it. When the specified volume has become available and has been loaded into a volume reader, the Jukebox manager returns control to the caller. In the testbed, RPC queuing is first-come-first-served, but this policy can be changed. Once allocated, a volume and its associated reader cannot be preempted.

A secondary goal of the Jukebox manager is to hide the physical attributes of storage devices from the rest of the Jaquith system; this goal was compromised slightly in the interests of performance. The Jukebox manager completely hides the physical location of the volumes in the cabinet and the activity of the robot arm. However, it exposes the volume reader to the caller so the caller can manipulate the device directly. Allowing the caller to open and write to a device directly improves throughput since the Jukebox manager does not need to set up or manage a data channel between the caller and the device: it is simply a lock manager. However, allowing direct device manipulation means that the Jukebox manager and its callers must be running on the same machine, so that they can share the devices.

The following sections describe the major server components: tape layout, buffering, indexing, and synchronization.

4.2 Tape Layout

Tape layout is critical because it directly affects the performance of the tape drive hardware and because it is the base on which the higher Jaquith software relies. If the high-level software fails or is unavailable, system administrators will be forced to deal with the raw tape format.

A first decision was that the tape layout should adhere to some Unix system standards so that tapes can be exchanged between sites and read with standard Unix tools, when necessary. Consequently, all Jaquith data is written to tape in POSIX `tar` (Tape ARchive) format [2]. The `tar` format has a 512 byte overhead per file but this is balanced by its convenience and portability. More serious is its limit of 255 characters for full file pathnames and its 100 character limit for link pathnames. For the testbed, these values were accepted and warning messages are printed if they are exceeded.

Jaquith uses a standard `tar` format at the low level, but imposes a higher level structure to take best advantage of the hardware features. The features of importance are the sequential access, the long latency for hardware filemarks, and the high-speed search feature.

The tape medium is a sequential access medium and does not support update-in-place

operations. (Some 4mm DAT tapes can be pre-formatted to allow update-in-place operations but performance and capacity are sacrificed, see [27]. This *dataDat* mode has not caught on and neither 8mm nor VHS tape supports it.) The design of the tape layout cannot include rewritable index blocks, or updatable data.

Hardware filemarks are the universal file separator for tape systems. A filemark is a unique bit pattern created by a specialized write command that is distinguishable from all user data. The latency to create a filemark, the space it consumes on tape, and its internal format are properties of the hardware and cannot be altered.

A high-speed search feature is available on both the Exabyte and Metrum readers. The search feature lets the hardware locate a particular filemark without having to read the intervening portion of the tape. Searches can be done in both the forward and reverse directions.

Jaquith takes advantage these features by formatting a tape as a series of large data units separated by hardware filemarks. New data is written in large I/Os to the end of the tape followed by a new filemark. Old data is retrieved from tape by using the high-speed search facility to locate a specific filemark. The data immediately following the filemark is staged back to disk as a single unit. Once on disk, the unit is broken open and the required user file is extracted.

There are two types of data units on a Jaquith tape: *buffers* and *headers*. Every even numbered data unit is a *buffer* that holds user data packed in `tar` format as detailed in the next section. Each intervening unit is a *header* (also in `tar` format) that contains a copy of Jaquith's indexing information for the user files in the adjoining buffer. If the Jaquith disk structure is destroyed the *header* files provide a backup. See Figure 4.

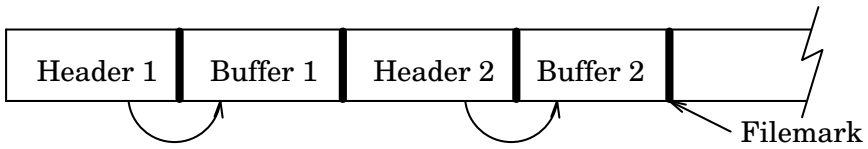

Figure 4: Jaquith tape layout. The tape consists of a series of variable-size units separated by hardware filemarks. Every other unit is data *buffer* which contains user data packed into `tar` format. The intervening units are *headers* that contain the indexing information for all the files in the associated buffer.

Filemarks are the critical aspect of Jaquith's tape layout. They are the only feasible way to navigate quickly on tape, so they are essential. (Some tape hardware supports a block-level search, but this is not universal.) A tape format with many filemarks will position itself to the desired buffer more precisely than one with few filemarks. On the other hand tape marks consume space and hurt write performance. On an EXB-8500 drive a single tape mark consumes 48 KB of

space (more than twice the size of a typical Unix file; see Table 3), so writing data in small units separated by filemarks will significantly reduce volume capacity. Filemarks on older Exabyte 8200 drives are even worse – 2.2 MB each [5].[2] Reducing the number of filemarks also saves write time since writing a single filemark takes about two seconds on the Exabyte 8500 and eight seconds on Metrum RSP-2150.

Interleaving data *buffers* and metadata *headers* is superior to placing all the metadata at the end of the tape. An end-of-tape strategy requires rewriting the metadata after each new buffer is appended to tape. As the amount of data grows, this becomes a significant overhead. With the interleaving strategy each buffer is self-describing since its associated metadata is directly adjacent. A premature end-of-tape indication (due to media failure or software error) will not render the entire tape useless, as would a single end-of-tape index. A single front-of-tape index is not feasible because the tape does not support update-in-place operation. There is no way to grow the index as data buffers are added to the tape.

In normal operation, Jaquith ignores the metadata and seeks directly to the buffer containing the users's file. (The conversion of user file names to buffer numbers is done by the indexes as described in the next section.) In the event that a Jaquith disk index becomes corrupted and needs to be rebuilt, the metadata can be extracted efficiently by using the drive's high-speed search feature to skip from filemark to filemark directly. If individual buffers were formed by writing a file's data and metadata together, the entire tape would have to be read to extract the index. A sequential scan of either an Exabyte or Metrum tape is a multi-hour process.

Each tape is self-contained: There is no "master tape" responsible for the entire archive's indexing information. Such a tape would be a single point of failure for the system. A master tape would also be a bottleneck, limiting parallelism for systems with many tape drives (all needing to do index operations simultaneously). Conversely, a master tape would permanently consume a precious tape reader in systems with only a few readers. Another major benefit of self-contained tapes is the ability to remove an individual tape from the Jaquith system easily. Tapes need to be removed when the physical archive is full, and when a user wants to take her data traveling.

4.3 Buffering

The previous section described how *buffers* are arranged on tertiary storage. This section describes how *buffers* are managed on disk. Figure 5 shows the buffering strategy. First, the

[2] In Exabyte terminology, 48 KB is the size of the rewritable or *long* filemark. A terminating, *short*, filemark is 1K; data cannot be appended to it so it is useless.

mechanisms for creating and managing this strategy are described, and then motivations for the design decisions are presented.

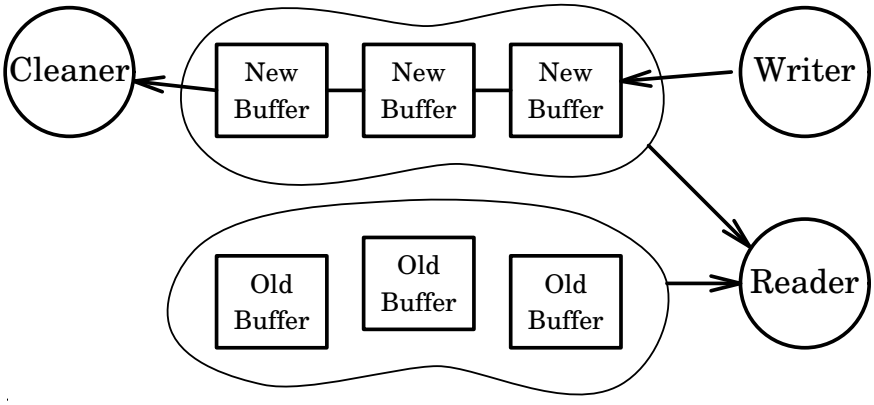

Figure 5: Jaquith buffer handling. Client data is packaged by a *writer* process into buffers and appended to the new-buffer queue. The new-buffer queue is emptied by a *cleaner* process which writes buffers to tape in sequential order. There is also an old-buffer pool managed using least-recently-used replacement. *Reader* processes may satisfy client `jget` requests from either buffer pool.

Every logical archive is an append-only log composed of a sequence of data buffers numbered 0 to N. Buffers are created and filled with user data on the fly; they are not preallocated. Only the last buffer is active and receiving new user data. The active buffer is packed with incoming files until it is "full", that is, until it has passed the target buffer size, typically two megabytes. User files are never split across buffers, so the active buffer may grow beyond its target size.

All user data that is bound for the same logical archive is packed sequentially into the active buffer. This means that data archived by different clients may be put in the same buffer, if the first user did not fill the buffer. As soon as the last of the user's data has been buffered on disk, the `jput` operation is considered complete and a "success" status is returned to the client program.

Buffering is the point at which Jaquith takes responsibility for the user file. Jaquith keeps the state of the active buffer in a log file and updates it only after each user file has been appended and indexed. If a disk or network failure occurs while appending to the buffer, the log file will not be updated to reflect the new buffer status. The next write to the buffer will use the old status, overwriting the partial file and effecting a rollback. The client receives a confirmation on each file so it knows how many files were successfully archived.

Full buffers are subsequently written to tertiary storage and removed from disk by special *cleaner* processes. A cleaner process is spawned when disk consumption reaches a high-water level and terminates when the disk has been cleaned down to a low-water mark. (Users can explicitly

request synchronous writing to tape at a severe performance penalty.) Until they are written out, these buffers reside in the new buffer pool where they can be read by *reader* processes if necessary.

A *buffer*/*header* pair represents the unit of failure at the tape level. If either part of the pair fails to be written, the failure is noted by Jaquith and the pair is retained on disk.[3]

Reader processes satisfy user `jget` requests by looking for the desired buffer in the disk buffer pool. If the buffer is not there it restores it from tape to the pool.

The purposes of the buffer design are threefold:

- To improve throughput by using large I/O operations,
- To minimize robot activity via the cache, and
- To balance response time against disk space use.

Clearly, packing user data into buffers is necessary for good I/O performance. Large I/Os are more efficient than small ones and the cost of preparing a volume (loading and seeking it) is too high to justify many small I/O operations. The overriding factor driving the choice of buffer size is the hardware-dependent filemark, as described in Section 4.2. Large buffers use fewer tape marks reducing space overhead and improving write performance. Buffers on the testbed were set to 2 megabytes for the Exabyte readers. With a 12 KB header file and a 48 KB filemark this yields about $2000/(2000 + 12 + 48 + 48) * 100 \approx 95\%$ volume utilization.

The overhead to move a two megabyte buffer in order to satisfy a small user file request is justified by an assumption of temporal locality. We believe that users will archive and retrieve data on a directory basis. This means that buffers will typically be collections of files from the same disk subtree, and that each buffer will be used to satisfy multiple file requests as the directory is restored. Whether or not this assumption has merit will be proved by the testbed.

Jaquith's LRU buffer cache helps minimize device activity. As Figure 5 shows, the cache includes old buffers previously read from tape and new buffers not yet removed by the cleaner. This cache can be used to satisfy `jget` requests without causing any tape or robot motion. The effectiveness of the cache depends on the amount of disk space available versus the archive reference pattern.

The target buffer size can be set differently for each logical archive. Larger buffers increase latency to the user and use more disk space, but amortize overhead costs over more data.

[3]Data may be lost even though the write to tape succeeded. Both Exabyte and Metrum drives have internal buffers and the SCSI firmware returns a success message when the last data byte is received by the buffer, not after it has been written to tape. Neither drive will run with a buffer size of zero bytes.

Item	Size (bytes)	Item	Size (bytes)
Simple filename	12	Owner name	8
File size	4	Group name	8
Permissions	4	Link name (soft links only)	40
Last access time	8	File list (directories only)	100
Last mod time	8	Buffer number	4
Archive time	8	Offset within buffer	6

Table 2: A Jaquith index record. This table lists the indexing information that Jaquith stores for each archived file or directory. Directory entries contain a list of the files residing in the directory, so they tend to be larger than file entries. All index data is stored in ASCII.

4.4 Indexing

This section describes the structure and content of the indexing information that Jaquith maintains for each user file. The main purpose of the disk index is to let users browse the archive without making any accesses to tape, since each access is noticeable by the user. A backup copy of the disk index is maintained on tape in the *header* data unit, as described in Section 4.1.

For each file archived, Jaquith creates an index record of approximately 70 bytes which is used for responding to lookup queries. Table 2 shows the record items that Jaquith stores for each file. The precise size of the index entry depends on the type of file being archived. Index entries for directories are substantially larger than entries for regular files because they contain a list of the file and subdirectory names.

The price paid for quick lookup performance is disk space. Even at 70 bytes/entry, an Exabyte jukebox can consume a 2 GB disk with indexing information. Using Tables 1, 2 and 3 we can make some assumptions and do a quick calculation about index space. Assuming an average file to be 12 KB, the ratio of files to directories to be 10:1 and index entries to be 70 and 170 bytes for files and directories respectively. Then, the Exabyte has an upper bound of $(116 \cdot 5\text{GB})/12\text{KB} \approx 50$ million files and a total index space consumption of $(50\text{M} \cdot 70\text{B}) + (5\text{M} \cdot 170\text{B}) \approx 4.4$ GB. Actual experience shows that the index space is about three-quarters of one percent of the total bytes stored, including `tar` overhead but not abstracts. This check corroborates our back-of-the-envelope calculation: $560\text{GB} \cdot 0.0075 \approx 4.2$ GB of space.

Our index scheme was designed for the half-terabyte Exabyte jukebox. The larger Metrum jukebox would consume $(600 * 14.5\text{GB} \cdot 0.0075)/2\text{GB} \approx 32$ two-gigabyte disks with indexing information, which is infeasible. Jaquith should either compress the index or store it on tertiary storage and cache the the active index information on disk.

	Source code disk	User disk 1	User disk 2
Total # directories	3961	3321	5400
Total directory space in KB	4393	3951	6892
Total # files	26377	35828	66257
Total file space in KB	863884	394937	838589
File size in KB	28/3/55 (185.0)	11/2/16 (76.2)	12/2/19 (98.9)
Pathname length	50/51/61 (8.3)	35/33/50 (10.2)	52/55/70 (15.2)
Filename length	10/11/16 (3.8)	8/8/15 (4.2)	10/10/16 (4.4)
Files/directory	7/4/19 (12.9)	8/3/21 (37.7)	12/7/22 (26.0)
Non-zero dirs/directory	2/2/6 (3.6)	3/3/7 (5.9)	6/3/16 (10.7)

Table 3: Statistics for three Unix file systems. Notation: X/Y/Z shows average value, 50th percentile, and 90th percentile while standard deviations are in parentheses. Values were gathered by a scan of the static filesystem; statistics weighted by file use may be different.

A simple access method governs the reading and writing of index records. The client's tree structure is replicated on the indexing disk as a subtree of the logical archive's root (`slash`). When a file is put into the logical archive, its pathname is broken into components and the full tree structure is constructed on the indexing disk, leading down to the subdirectory where the file would reside. In place of the file itself, however, is an index file `_jaquith.files` that contains an index record for every file located in that directory. Record entries are appended to this index file each time a file from this directory is archived. Figure 6 shows the structure created for a simple disk name space.

Managing the index file as a log lets *readers* and a *writer* share the index file without any locking. (At worst, a *reader* may get an incomplete last entry, which it discards.)

The use of a standard Unix tree structure, coupled with the fact that all information is stored in ASCII, means that the index can be read and repaired using standard Unix tools (`find`, `grep`, etc.). Since the index is crucial to Jaquith's behavior, and it is the most exposed part of the system, designing it for easy maintenance by people is worthwhile. The index can be placed on a disk array for even more reliability.

The indexer strategy is quite simple; it is based on the same hypothesis that supported the assumption of locality within data buffers, namely that users will tend to use the disk directory as a file grouping mechanism. With this assumption, the indexer will have good performance. It keeps the metainformation for files in the same disk directory close together in the index and it keeps all versions of the same file in the same `_jaquith.files` file. Therefore, the time to answer a client query is *independent* of the query options (date, owner, abstract, etc.), provided that the

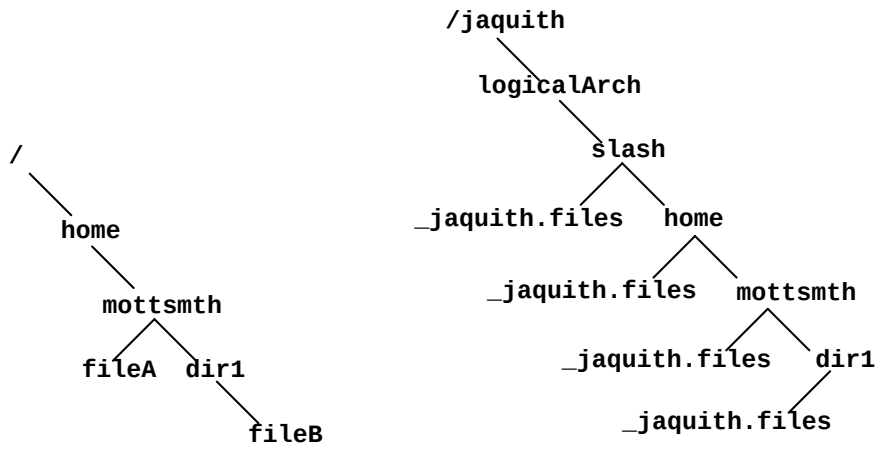

Figure 6: Jaquith indexing structure. On the left is a hypothetical disk name space. On the right is the corresponding index structure. The disk structure is copied beneath the logical archive, and each leaf file is replaced by an entry in the file `_jaquith.files`. For example, `mottsmth/_jaquith.files` contains an entry for both `dir1` and `fileA`.

query specifies a directory. The indexer simply opens and scans one index file to answer any such query. However, the time is *dependent* on the amount of data in the file, since the file has no internal structure and must be read in its entirety. Queries with wildcard characters for the directory name will be slow since Jaquith maintains no auxiliary indices and will have to scan a large part of the archive index. An access method on the *abstract* field might be a valuable addition if it turns out the users often do large non-directory-based queries using abstracts.

One way to add multiple access methods is to use a relational database system (RDBMS) to do all the indexing. Using a full RDBMS with a query language would make it possible to research various indexing techniques (for example [23]), but wouldn't contribute to Jaquith's immediate goals. We considered using the Postgres RDBMS [26, 25] for the initial testbed, but its size and complexity did not seem to warrant its use. For example, Postgres version 4.0 has a storage overhead of 50-60 bytes/tuple [16], which would almost double the size of a standard index entry. Requiring the installation of a full DBMS would make Jaquith much less portable.

4.5 Synchronization

Parallelism is achieved by the use of multiple reader and writer processes. Multiple readers are always allowed and multiple writers on different logical archives are allowed. However, for a given logical archive there can be only one active writer process because there is only a single write point on the archive, namely the end of the log.

The single writer limitation is enforced by the server process, rather than by mutual coordination among writers using locks. While a writer is active, the server enqueues all incoming `jput` requests for the same logical archive. Each of these requests is converted into a writer process when the previous writer completes. Each writer process can perform indexing and buffering without the complexities and overhead of locking since it has no competitors.

Locking is necessary to avoid inconsistencies between writer and cleaner processes (or between multiple cleaners). Jaquith uses lock files for interprocess coordination. These short-term locks add some filesystem overhead but do not significantly limit throughput since they are held just long enough to update Jaquith's buffer-location information. Until Jaquith has updated this information and flushed it to disk, it cannot delete the disk copy of the buffer.

4.6 System Administration

Jaquith system administration is concerned with (1) configuring the system, (2) monitoring activity, and (3) maintaining the physical and logical resources. Items (1) and (2) are controlled by configuration and logging files, and their use is straightforward. Appendix I describes each file and its purpose. This section focuses on item (3) and the three administrative utility programs for managing Jaquith resources:

`jctrl`	— Manages the physical jukebox.
`jcopy`	— Duplicates physical volumes.
`jbuild`	— Rebuilds the on-line index.

Collectively, these utility programs keep Jaquith's world, as described by its files, coordinated with the actual state of the real world. Each utility is run by the Archive Administrator at the server's site (there are no remote administration facilities), and performs the following functions:

`Jctrl` provides manual control over the jukebox when the jukebox manager is not running. Manual control is useful for resetting the system to a known physical state before putting Jaquith on-line. For example, it might be necessary to unload a volume from a reader:

```
jctrl -cmd unload -vol 123 -dev /dev/nrmt3a
```

unloads volume 123 from the specified device and returns it to its slot in the jukebox. Volumes can be completely removed from the jukebox, if the jukebox has an entry/exit port. `Jctrl` is careful to update the Jaquith support files which describe the volume's current status when it removes a volume.

`Jcopy` is used to perform a full volume-to-volume copy operation. A volume-copy

feature is important because tapes have a limited lifetime, determined by usage patterns and storage conditions. Archived data must be rolled forward from old tapes to new tape periodically. A typical invocation:

```
jcopy -v -srcvol 1 -destvol 101
```

acquires two readers from the jukebox manager and copies volume 1 to volume 101. Elapsed time is several hours for a full tape. The source volume is unavailable during this period, but Jaquith remains on-line otherwise. Presently, Jaquith has a *mechanism* for doing the copy, but no *policy* for deciding when to do the copy. It would be valuable to add usage counters to Jaquith's device module. These counters would record the number of tape loads, unloads, seeks, reads and writes on a volume-by-volume basis. The counters would not only guide the Archive Administrator in copying tapes, but would also confirm (or not!) manufacturers' claims about tape reliability and shelf-life.

`Jbuild` rebuilds Jaquith's disk index structure from the *header* units stored on tape. A complete or partial rebuild is possible, as determined by command line parameters. Files can be restored by name, buffer number, or date, as necessary. For example:

```
jbuild -matchpath "/home/mottsmth/src/*" -dev /dev/nrmt3a
```

reads the volume in the specified device and rebuilds the index entries for all files which match the given pathname. Restoring the complete index for a 400 MB file system (approximately 40000 entries in 200 files), takes about 35 minutes.

The Jaquith testbed does not tackle two additional topics which are related to system administration: quotas and security. These are described below for completeness.

The Jaquith testbed provides no quota system. Any client who has been granted write access to some logical archive can fill the physical archive to capacity by consuming all the volumes in the global free-volume pool. Such greediness will use enough CPU and network resources that an Archive Administrator will probably notice the activity, but there is currently no automated way to prevent this. (Jaquith does automatically send mail to the administrator when the number of free volumes drops below a threshold.) Either of two automatic quotas systems seems reasonable: (1) limit the number of files (or bytes) that a user can write to an archive in one day, or (2) preallocate a certain number of volumes to an archive. The first solution resembles some Unix printing quota mechanisms. It is a soft limit used mainly to discourage resource hogging (paper or tape drives). The second solution imposes a hard limit on the archive, but not necessary a specific user.

The testbed's second weak area is its security. The security model is simplistic and the base on which it is built is insecure. The model is weak because it treats login names as unique

identifiers. We recognize that login IDs are not unique and that this is naive for a large network with many (possibly duplicate) user names.

The model is built on the Unix privileged port mechanism, which is known to be insecure. Jaquith client programs are installed with `setuid root` privilege, and then call the server on a privileged port. The server trusts all callers who use a privileged port to be who they claim to be, using their hostname and username to do validation. The MIT Kerberos project has solved the authentication problems in a networked environment [24] and Jaquith should be "kerberized" for better security.

5 Performance

Jaquith's performance has two distinct parts: (1) packaging user files into disk buffers with associated index information and (2) writing full buffers to tertiary storage. Part one is dominated by the cost of synchronous disk writes to guarantee the durability of user data. If the guarantees are disabled then Jaquith's latency is about 1.5 times that of `tar`. Part two is dominated (on both tape systems) by the cost of writing hardware filemarks. Filemarks take several seconds each to create, but act as "search markers" to make retrievals much faster. The read and write performance of the two parts are discussed in more detail below. Further information about the testing environment is in Appendix II.

5.1 Performance of Disk Buffers

Jaquith buffering performance is most easily compared to the `tar` utility since it serves much the same purpose and uses the same data format. To compare `tar` with Jaquith, two sets of source data were prepared: one with a single 10 MB file, and the other with a balanced 10-ary tree of 1000 files, each 10 KB long. `Tar` packaged the 10 megabytes into a single tar file on a local disk. Jaquith packaged the data into two-megabyte buffers with an associated header file.

Table 4 compares `tar` with Jaquith for the two test cases. Six runs were made for each test case, two with `tar` and four with Jaquith. `Tar` was run first in standard mode and then with its output forced through a socket. For this purpose, two very simple TCP socket stubs were written in **C**. The data from `tar` was piped to one stub which copied it across the socket to the receiving stub, which wrote it into a local disk file. The Jaquith runs varied the two parameters *index* and *fsync*. The *index* option tells Jaquith to do an index lookup on each file to determine whether or not

it has been modified since it was last archived (that is, it does an incremental dump). The client waits for a synchronous go-ahead message from the server in this case. *No index* does a full dump so no lookups are done; *Fsync* means that Jaquith should maintain database semantics by forcing all buffer and index changes to disk before updating the log file; *no fsync* means that Jaquith should run without any guarantees of durability (though it still performs an `fsync` when it closes each buffer). For the 1000 file case Jaquith wrote about 124 KB of indexing information.

Buffer write test	1 file of 10 MB	1000 files of 10 KB
`tar`	27.1 (0.15)	51.9 (0.42)
`tar` – with socket	22.1 (3.56)	60.9 (0.84)
Jaquith – no fsync, no index	29.8 (0.28)	72.9 (0.99)
Jaquith – no fsync, index	29.6 (0.41)	84.4 (0.55)
Jaquith – fsync, no index	29.9 (0.09)	219.0 (1.57)
Jaquith – fsync, index	29.8 (0.03)	238.6 (1.45)

Table 4: Buffer write performance. This table shows the time, in seconds, to package 10 MB of data using `tar` and Jaquith. Two scenarios are given: a single file of 10 MB, and 1000 files of 10 KB each. *No index* indicates that Jaquith archived the data without doing any index lookup operations; *index* indicates that Jaquith checked the index before determining that the file needed archiving. *Fsync* indicates that Jaquith performed an `fsync` operation after each file to guarantee data consistency; *no fsync* means that Jaquith did an fsync only when the buffer was closed. Standard deviations are in parentheses.

The `tar` results make two points. First, the network overhead is minimal. This is expected because the client and server were running on the same machine. There is no actual transmission time, only software socket overhead. Second, packaging 10 MB from 1000 files is slower than from a single file: 369 KB/second vs. 193 KB/second. This drop in throughput is due to the extra `stat` and `open` operations.

The Jaquith results show that Jaquith is competitive for large files but its per-file overhead diminishes its performance in the multiple file case. With multiple files, its throughput ranges from 137 KB/second in the best case to 42 KB/second in the worst case.

The overhead in the optimum (no index, no fsync) case is due to general bookkeeping. Bookkeeping involves the management of four I/O streams: the buffer, the header, the log file and the index file. The data is copied to the buffer stream, the index information is copied to the header and index file streams, and then the log is updated to record the transaction. The log record is necessary for roll-back in case the client or network crashes while a file is being processed.

The *index* parameter adds a fixed cost for the synchronous client-server handshake plus overhead time proportional to the size of the index file (`_jaquith.files` file) since it must read

and parse the entire file to do the lookup. The tests in Table 4 were run with a null index so file processing time was a minimum. To assist more realistic situations, Jaquith caches the last index file it has read so that sequential lookups from the same directory do not repay the file I/O or parsing costs.

The huge overhead in the worst cases is due to the cost of synchronous `fsync` operations. The cost of a single `fsync` is roughly 40 ms and Jaquith must do four of them (one for each I/O stream) for each user file in order to guarantee that the file has been archived. This increases the per-file overhead from about 50 ms to 210 ms with a corresponding drop in throughput. A command-line argument lets the Archive Administrator control the frequency of the `fsync` operations.

Jaquith's overhead provides value that `tar` doesn't offer. The indexing provides fast file retrieval and the `fsync`s provide guarantees in the face of crash recovery. Table 5 compares Jaquith's disk read performance with that of `tar`.

	Retrieve one 10 KB file		Retrieve 1000 10 KB files	
Buffer read test	from 10 MB	from 100 MB	from 10 MB	from 100 MB
`tar`	9.1 (0.20)	80.3 (0.42)	67.3 (1.38)	138.2 (0.36)
Jaquith	1.0 (0.06)	1.0 (0.08)	82.5 (0.54)	81.9 (0.42)

Table 5: Buffer read performance. This table shows the time, in seconds, to retrieve 10 KB and 10 MB of data from a group of files using `tar` and Jaquith. In the first case a single 10 KB file is retrieved, in the second case a balanced tree containing 1000 10 KB files is retrieved. Standard deviations are in parentheses.

When retrieving single files, Jaquith's performance is independent of the file's location in the archive; it knows the buffer number and the offset of the file within the buffer. `Tar`'s retrieval time grows with the size of the archive since it scans the entire `tar` package to locate the file. When retrieving whole subtrees of small files, Jaquith achieves a throughput of about 127 KB per second.

5.2 Performance of Tape

The second part of Jaquith's performance is writing disk buffers to tape. Table 6 compares the time to write a two-megabyte file to a prepared tape using a specialized C program and Jaquith.

The Exabyte time can be summarized as 3.5 seconds to copy the data and 2.8 seconds to write the filemark. The corresponding values for the Metrum are 2.1 and 14.6. The times listed in Table 6 are only approximate due to the large fluctuations in the physical tape handling on both tape systems. For example, the 14.6 seconds for the Metrum is twice the nominal time. Repeated tests show that Jaquith pays the nominal 7 seconds for the filemark after the header and 14 seconds for

Tape write test	Exabyte	Metrum
C program – memory	4.2 (0.52)	2.8 (1.81)
C program – FDDI	4.2 (0.44)	3.3 (1.78)
C program – FDDI, fmark	6.4 (0.07)	16.6 (0.14)
Jaquith	8.8 (0.06)	25.0 (0.09)

Table 6: Tape write performance. This table shows the time, in seconds, to write two megabytes of data to a tape drive under different conditions. The vanilla C program writes the data in 32 KB units three ways: *Memory* indicates that the source is a memory buffer, *FDDI* indicates that the source is a remote NFS disk served via FDDI, *fmark* indicates that the program wrote a file mark after the data. The Jaquith time includes the buffer, associated header file ($\approx$ 26 KB), and the two terminating filemarks. Standard deviations are in parentheses.

the one following the buffer. The variations are probably due to the reader's internal cache which makes it hard to predict actual tape head motion and leads to large standard deviations in throughput. Using two-megabyte buffers, Jaquith achieves approximately 235 KB/second throughput.

The large filemark penalty suggests that a larger Jaquith buffer should be used, perhaps 8 megabytes on the Exabyte and 32 megabytes on the Metrum. The increased disk space needed is a minor inconvenience. The only other problem is the increased buffer transfer time, which may be perceived by users who are retrieving single files and must wait for the read to complete.

In addition to the latency of the filemark, there is a huge cost to pick and load the tape, followed by a potentially large seek penalty. These costs are measured in minutes and dominate the actual write time. For good performance, it is important to correctly set the high- and low-water marks which control the cleaner process so that tape loading costs will be amortized over the writing of many dozens of data buffers.

The filemarks which limited write performance now provide fast read performance. Table 7 shows the tape read performance (excluding tape load time). The values include the time for Jaquith to seek past the filemark after the first header. `Tar` does no seeks. All values also include a 25-35 second delay that is characteristic of both tape readers after a rewind operation.

Jaquith uses the filemarks and the high-speed search feature to position the tape directly to the correct buffer. Then it transfers the single buffer to disk and extracts the file. Jaquith's retrieval time is therefore dominated by tape seek time. In contrast, `tar`'s retrieval time is dominated by tape read time since it reads the entire `tar` package from tape to locate the file. Searching is more than an order of magnitude faster than reading for both Exabyte and Metrum systems so Jaquith's advantage increases as the amount of data increases.

Tape read test	Retrieve one 10 KB file from 10 MB on tape		Retrieve one 10 KB file from 100 MB on tape	
	Exabyte	Metrum	Exabyte	Metrum
`tar`	54.0 (1.02)	53.6 (3.34)	253.7 (4.30)	173.5 (0.42)
Jaquith	40.8 (1.14)	39.1 (1.02)	40.6 (1.08)	38.8 (1.12)

Table 7: Tape read performance. This table shows the time, in seconds, to retrieve a single 10 KB file from a group of files using `tar` and Jaquith. In the first case the file is being retrieved from 10 MB of data (1000 files), in the second case it is being extracted from 100 MB of data (10000 files). All the data is located at the beginning of the tape. Standard deviations are in parentheses.

6 Related Work

This section describes several recent projects in automated storage systems, highlighting some of the differences between their goals and those of Jaquith.

6.1 Mass Storage System Reference Model

The Mass Storage System Reference Model [8] is a high-level specification for data storage, movement and access. The specification evolved from the mainframe environment so its main goals are:

Interface — Support for common interfaces: NFS and FTP.
Integration — Merging of second and third level store with automatic file migration.
Modularity — Separation of name service, from control path from data path.
Heterogeneity — Support for multiple network interfaces and robotic devices.

Several large-scale automated systems have been built around the Mass Storage Reference Model. They include the Common File System (originally from Los Alamos National Labs, now sold as DataTree by General Atomics), Unitree (originally from Livermore National Labs now sold by General Atomics), MSS-II at the National Center for Atmospheric Research, and NASA Ames [28, 15, 20, 14, 21].

The supercomputer centers running MSS-style code are primarily concerned with moving huge files though several levels of storage automatically. Therefore they provide file migration where Jaquith does not. They traditionally support both NFS and FTP interfaces since they have the luxury of integrating their code with the operating system. Jaquith runs at user level and provides only an FTP interface.

Additionally, MSS systems strive to integrate tertiary storage invisibly into the secondary filesystem using the standard filesystem interface. Therefore they do not provide added features

such as Jaquith's query mechanism or abstracts.

The MSS model specifies separate modules for the primary functions: name services (*nameserver*), file movement (*bitfile mover*), and storage management (*storage server*). Jaquith takes a more integrated approach, combining several functions into one process. A subroutine library isolates the implementation of a particular function to a single piece of code.

6.2 Systems with Jaquith Features

Beyond the MSS model are several smaller storage products. Many of them share some of Jaquith's features, but none of them has support for all of them: interactive browsing, abstracts, versions, incremental update, and parallel operation.

Automated backup programs, such as Legato Networker, are typical of the class of non-interactive programs. Legato is a software-only product which performs automated backup over a LAN of Unix workstations. It has an assortment of system administrator's criteria for controlling the flow of files to the backup devices. These backup programs do no extra indexing or special tape layout to make restoration fast. Restoring a tape typically means reading a large tar (or dump) file from beginning to end.

The Renaissance package from Epoch Systems is an example of a system offering a more interactive approach, similar to Jaquith. Epoch keeps the first 8 KB of each file on disk as a form of fixed abstract. This uses more space and is less flexible than Jaquith's abstracts. In other respects Renaissance falls somewhere between the MSS systems and the simplistic backup packages. Epoch uses a dedicated server running a version of SunOS with their own modifications, to manage three levels of storage, magnetic disk, optical disk and tape. The Renaissance package has a built-in migration policy. Jaquith does not have a migration policy.

A form of Jaquith's versioning feature is found in the "3D" filesystem from AT&T Bell Labs [22]. Roome's system retrieves old versions of files using data from incremental backups. It is therefore a read-only system. Like Jaquith it trades space for speed by storing complete versions of files, rather than rebuilding them on the fly. The 3D system does not keep a list of files belonging to each directory. Jaquith does keep such a list so it can restore complete, consistent subtrees. Where 3DFS runs with optical disk and presents an NFS interface, Jaquith uses tape and a get/put interface. The 3D filesystem is integrated into the shell. Normal shell syntax is extended with an '@' symbol to provide command-line access to the file versions. Jaquith relies on the shell-like `jls`, which is more complex, but offers more features.

Andrew Hume presented an automated backup system called the “File Motel” in 1987 [12] which was notable for being able to perform incremental dumps. A background process quietly archives files that have been changed without user intervention. Jaquith’s `jput` can also archive files incrementally. Nightly backups can be automated with a `cron` script. Interestingly, while the File Motel’s file backup operation is hidden from the user, file restoration requires human intervention to perform a database lookup. Jaquith hides the database from the user. The File Motel does not have Jaquith’s buffering or query features.

Finally, A special network backup system called Amanda was built at the University of Maryland [9]. Amanda’s goal is performance through parallelism. Amanda is an automated backup program built on top of the Unix `dump` utility that dumps filesystems over the network to multiple disk buffers simultaneously, Jaquith can support multiple concurrent I/O streams provided that they are directed to distinct logical archives. Parallelism can be increased by creating more logical archives, up to the number of readers in the jukebox. Other than parallel operation, Amanda does not offer any of Jaquith’s features. It is non-interactive, has no disk index, and provides no query support.

7 Future Work

This section summarizes the Jaquith project and considers how its fundamental design decisions might lead to future research. Three fruitful areas of further research are: data layout, file migration, and tape striping. Data layout concerns the (re)packing of data on tertiary storage to improve locality and performance. File migration is concerned with the automatic staging of files from tertiary to secondary storage, and Tape Striping involves the spreading of large datasets over several tapes in parallel to improve I/O bandwidth. The impact of these research goals on the Jaquith design is discussed below.

7.1 Data Clustering

There is ample room to explore data repacking in the current Jaquith system. Old tapes can be read, reorganized, and rewritten as desired. In fact, this would make it possible to lift a current Jaquith restriction: Jaquith has no delete feature. By design, all data written to the archive is kept forever. This decision was made to simplify both the indexing code and the tape management code.

Tape management is simpler without deletion because no holes are formed in the tapes by deleted data. This avoids the complexities of file compaction which would be necessary to recover deleted tape space. Once archive tapes become rewritable it is possible to reorganize user files to achieve better clustering and performance, as well as to recover dead-file space.

Indexing is made simple in the current system because no information is ever deleted from the index tree, consequently no in-place modifications are ever made to the disk index files; new data is simply appended to the file. *Reader* and a *writer* processes can share the index file without the overhead of locking.

7.2 File Migration

Jaquith is a potential base for file migration experiments. The client interfaces can be jettisoned and the server can be used purely as a bitfile mover. To be effective the migration policy must be integrated into the Operating System so that it runs without user intervention. This implies the need for kernel modifications, something the testbed avoids. It also implies that there must be an "up-call" mechanism whereby the kernel policy can invoke the Jaquith mechanism at user level.

7.3 Tape Striping

The immediate hardware limitation for Jaquith is latency because the Unix environment uses small files. In a different environment where the average file size is many megabytes or gigabytes (for example satellite data), tape throughput becomes a problem. One solution to the bandwidth problem is to apply disk RAID [13] ideas to tape systems. The main idea is to write a large file out across several tapes in parallel, with parity written to an additional tape for recovery purposes.

Jaquith's current indexing technique needs to be expanded to handle tape striping experiments; it currently assumes that a file does not span a tape. Some additional work must be done to reassemble a buffer from several tapes, and to rebuild a tape from parity in the event of a failure. There remain, of course, the primary problems associated with striping itself: no synchronization among tape drives (unlike disk drives), poor media reliability, and no random data access.

8 Conclusions

Jaquith addresses multiple issues in tertiary storage, many of them successfully, others less so. Jaquith provides robotic storage space to remote network users using a flexible query interface. Its wildcard search capability lets users locate their data easily, and its use of an on-line index with abstracts assists the user in browsing the large namespace. Additionally, Jaquith does intelligent buffering and tape management.

The current implementation also has two notable weaknesses. First, the indexing scheme consumes too much space, over 3/4 of one percent of data space, in our Unix environment. This means that the current generation Metrum jukebox cannot be supported with 1 or 2 current generation disks. Future systems need to compress the index or maintain an on-line cache of a larger index on tertiary storage. Second, Jaquith performance is constrained by the numerous synchronous operations between server and client. The overheads for logging and acknowledging each user file are costly, particularly on a slow network. These costs need to be amortized by batching user files into larger request units and performing the client-server handshake once for each unit.

9 Acknowledgements

My sincere thanks to the many members of the CS department who were, knowingly or not, very fine sounding boards for tertiary storage discussions. I would particularly like to thank Stephen Smoot and Mark Sullivan for focusing me during my periods of indecision and Mike Kupfer for his many encouraging comments over meals of mediocre Chinese food. My appreciation goes to Randy Katz for agreeing to be my second reader, and to Ken Shirriff and Mike Olson for puzzling over early drafts of this report. John Hartman earns my thanks for being a stellar alpha-tester. Finally, my supreme thanks to John Ousterhout for providing levels of wisdom and guidance well beyond what could reasonably be expected from any advisor.

1 Appendix I

Jaquith maintains a number of files describing the state of the physical and logical archive. All of these files reside in the Jaquith root (usually `/arch` but settable with the `-root` flag). In the following description, *archive* refers to the name of the containing logical archive.

`freevols`	The global pool of available volumes. These will be consumed as needed by the cleaner process.
`devconfig`	The list of device names to be controlled by the jukebox manager.
`volconfig`	The list of volumes and their slot locations in the jukebox.
`tbuf.lru`	The global least-recently-used buffer list. Buffers at the top of the list are prime candidates for removal if disk space is tight.
`rebuild.`*pid*	Temporary work file created by `jbuild`.
`thdr.`*pid.num*	Emergency work file created by `jbuild` when it can't parse the thdr file it got from tape.
archive`/config`	Jukebox server info and target buffer size
archive`/tbuf`	Current buffer number. *i.e.* the current buffer is `tbuf.`*num*.
archive`/tbuf.`*num*	Tape buffer files. These files contain user file data in POSIX tar format.
archive`/thdr`	Tape header files. These files contain the indexing information in case the disk index structure is destroyed.
archive/meta.*num*	Meta information about `tbuf.`*num* maintained while buffer is being built, in case of crash.
archive`/log`	The list of operations which have been requested on this logical archive.
archive`/filemap`	The list of volumes assigned to this archive along with the starting buffer number on the volume.
archive`/auth`	The authorization information for the logical archive. The file consists of single-line entries in no particular order. Each entry has the form *username groupname hostname permission*. The first 3 items are globbing patterns, and *permission* is `R`, `W`, or `O` for read, read-and-write or ownership respectively. Ownership means that the specified user or group has 'root' access to this logical archive. Client requests are validated by first looking for an entry with a corresponding user name. If none is found then an entry with a matching group name is used, else permission is denied.
archive`/volinfo`	Physical volume information for this logical archive. Entries are: current volume id, next location on volume where buffer is to be written, remaining space in KB on volume, last buffer number written, current buffer number.
archive`/tbufinfo`	Current buffer information for this logical archive. Entries are: buffer size in bytes, buffer header size in bytes, number of user files in buffer.

2 Appendix II

Testing configuration and parameters.

Host	DS5000/200 with 32 MB of memory, 1 SCSI bus, 1 ethernet adapter, and 1 FDDI adapter.
Local disks	one RZ55, one RZ57.
Exabyte hardware	EXB-120 with microcode version 2.24. EXB-8500 with microcode version 0446
Metrum hardware	RSS-600 with RSP-2150 with microcode version 3.02/3.00.
Operating system	Ultrix 4.2.
I/O operation size	32768 bytes.
Jaquith buffer unit size	2 megabytes.

The source data for the Jaquith and `tar` tests was placed on a separate NFS filesystem served by FDDI. The destination filesystem was a local RZ57 disk. This arrangement minimized contention on the local SCSI bus and ensured that access to the data source was not the limiting factor in the tests. Below are the times for tar-ing a ten megabyte file to a local disk.

Source and destination sharing same disk	44 secs
Source and destination on separated disks on same SCSI chain	32 secs
Source data served by FDDI over NFS, destination on local disk	27 secs

Bibliography

[1] *UNIX Programmer's Supplementary Documents, vol. 1, 4.3BSD*. Computer Systems Research Group, Berkeley, CA, June 1987.

[2] *IEEE Standard Portable Operating System Interface for Computer Environments*. Institute of Electrical and Electronics Engineers, 345 E. 47th St., New York, NY, 1988.

[3] *METRUM Information Storage RSS-600 Technical Manual*. METRUM Information Storage, Denver, CO, Nov 1990.

[4] *Exabyte-120 Cartridge Hbandling Subsystem User's Manual*. Exabyte Corporation, Boulder, CO, 1991.

[5] *Exabyte-8500 8mm Cartridge Tape Subsystem User's Manual*. Exabyte Corporation, Boulder, CO, 1991.

[6] *METRUM Information Storage RSP-2150 Operation Guide*. METRUM Information Storage, Denver, CO, July 1991.

[7] Mary G. Baker et al. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 198–212, Oct 1991.

[8] Sam Coleman and Steve Miller (eds.). Mass storage system reference model: Version 4. IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

[9] James da Silva, Ólafur Guŏmondsson, and Daniel Mossé. Performance of a Parallel Network Backup Manager. In *1992 Summer Usenix Conference Proceedings*, pages 217–225, 1992.

[10] Ann Chervenak Drapeau. U.C. Berkeley Technical report. To appear.

[11] Joel Fine et al. Abstracts: A Latency-Hiding Technique for High-Capacity Mass-Storage Systems. Technical Report UCB/CSD 99/11 June 1992, University of California, Berkeley, June 1992.

[12] Andrew Hume. The File Motel. In *1988 Summer Usenix Conference Proceedings*, pages 61–72, 1988.

[13] Randy H. Katz, Garth A. Gibson, and David A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1858, December 1989.

[14] Fred McClain. DataTree and UniTree: Software for file and storage management. In *Digest of Papers*, pages 126–128. Tenth IEEE Symposium on Mass Storage Systems, May 1990.

[15] Marc Nelson, David L. Kitts, John H. Merrill, and Gene Harano. The NCAR mass storage system. In *Digest of Papers*. Eighth IEEE Symposium on Mass Storage Systems, November 1987.

[16] Michael Olson. Personal communication.

[17] J. K. Ousterhout. Tcl: An Embeddable Command Language. In *1990 Winter Usenix Conference Proceedings*, pages 133–146. USENIX, Jan 1990.

[18] J. K. Ousterhout. An X11 toolkit based on the Tcl language. In *1991 Winter Usenix Conference Proceedings*. USENIX, Jan 1991.

[19] J. K. Ousterhout et al. A Trace-Driven analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 15–24, Dec 1985.

[20] Anthony L. Peterson. E-Systems Modular Automated Storage System (EMASS) software functionality. In *Digest of Papers*, pages 73–76. Eleventh IEEE Symposium on Mass Storage Systems, October 1991.

[21] Dennis F. Reed and Gary A. Mueller. Automated cartidge system library server. In *Digest of Papers*, pages 105–110. Tenth IEEE Symposium on Mass Storage Systems, 1990.

[22] W. D. Roome. 3DFS: A time-oriented file server. In *1992 Winter Usenix Conference Proceedings*, pages 405–418. Usenix, Jan 1992.

[23] Stuart Sechrest. Atrribute-Based Naming of Files. Technical Report CSE-TR-78-91, University of Michigan, January 1991.

[24] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *1988 Winter Usenix Conference Proceedings*, pages 191–202. USENIX, Jan 1988.

[25] Michael Stonebraker et al. The Implementation of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, Mar 1990.

[26] Michael Stonebraker and Larry Rowe. The Design of POSTGRES. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, 1986.

[27] Eng Tan and Bert Vermeulen. Digital audio tape for data storage. *IEEE Spectrum*, October 1989.

[28] David Tweten. Hiding mass storage under UNIX: NASA's MSS-II architecture. In *Digest of Papers*, pages 140–145. Tenth IEEE Symposium on Mass Storage Systems, May 1990.