

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
2 June 2005 (02.06.2005)

PCT

(10) International Publication Number
WO 2005/050383 A2

(51) International Patent Classification⁷: **G06F**
(21) International Application Number:
PCT/US2004/038280

(22) International Filing Date:
15 November 2004 (15.11.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/519,526 13 November 2003 (13.11.2003) US

(71) Applicant (for all designated States except US): **COMM-
VAULT SYSTEMS, INC.** [US/US]; Two Crescent Place,
Oceanport, NJ 07757-0090 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **KUMAR, Manoj**;
2140 Aldrin Road, Ocean, NJ 07712 (US). **PRASAD,
Arun** [IN/IN]; 2406 E Block, Jothy, Illam., Sahakara,
Nagar, Bellary Road, Bangalore 650092 (IN). **KOT-
TOMTHARAYIL, Rajiv** [IN/IN]; 1508 Garden Drive,
Ocean, NJ 07712 (US).

(74) Agent: **SHANAHAN, Michael**; Brown Raysman Mill-
stein, Felder & Steiner LLP, 900 Third Avenue, New York,
NY 10022 (US).

(81) Designated States (unless otherwise indicated, for every
kind of national protection available): AE, AG, AL, AM,
AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN,
CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI,
GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE,
KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD,
MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,
PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM,
TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM,
ZW.

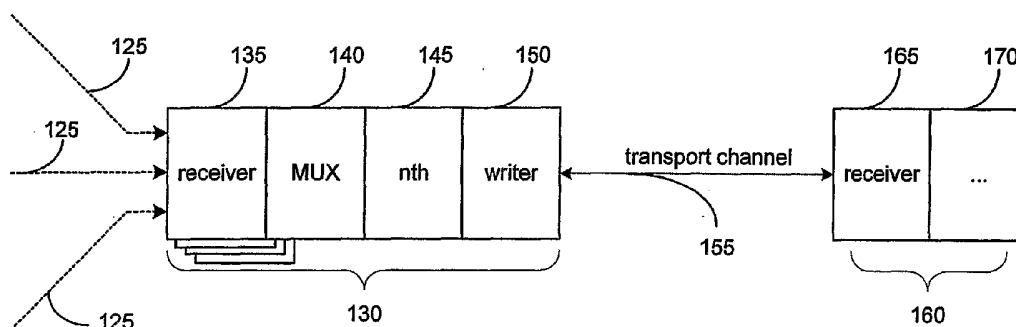
(84) Designated States (unless otherwise indicated, for every
kind of regional protection available): ARIPO (BW, GH,
GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM,
ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM),
European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,
FR, GB, GR, HU, IE, IS, IT, LU, MC, NL, PL, PT, RO, SE,
SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,
GW, ML, MR, NE, SN, TD, TG).

Published:

— without international search report and to be republished
upon receipt of that report

[Continued on next page]

(54) Title: SYSTEM AND METHOD FOR COMBINING DATA STREAMS IN PIPELINED STORAGE OPERATIONS IN A STORAGE NETWORK



(57) Abstract: Described herein are systems and methods for multiplexing pipelined data for backup operations. Various data streams are combined such as by multiplexing by a multiplexing module. The multiplexing module combines the data from the various data streams received by receiver module(s) into a single stream of chunks. The multiplexing module may combine data from multiple archive files into a single chunk. Additional modules perform other operations on the chunks of data to be transported such as encryption, compression, etc. The data chunks are transmitted via a transport channel to a receive pipeline that includes a second receiver module and other modules. The data chunks are then stored in a backup medium. The chunks are later retrieved and separated such as by demultiplexing for restoring to a client or for further storage as auxiliary copies of the separated data streams or archive files.



For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

SYSTEM AND METHOD FOR COMBINING DATA STREAMS IN PIPELINED STORAGE OPERATIONS IN A STORAGE NETWORK

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosures, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

RELATED APPLICATIONS

This application claims the benefit of U.S. provisional application no. 60/519,526 titled SYSTEM AND METHOD FOR PERFORMING PIPELINED STORAGE OPERATIONS IN A STORAGE NETWORK, filed November 13, 2003, which application is incorporated herein by reference in its entirety.

This application is related to the following patents and pending applications, each of which is hereby incorporated herein by reference in its entirety:

- Application Serial No. __/_____, titled SYSTEM AND METHOD FOR PROVIDING ENCRYPTION IN PIPELINED STORAGE OPERATIONS IN A STORAGE NETWORK, filed November 15, 2004, attorney docket number 4982/45;
- U.S. Patent No. 6,418,478, titled PIPELINED HIGH SPEED DATA TRANSFER MECHANISM, issued July 9, 2002, attorney docket number 4982/6;
- Application Serial No. 09/495,751, titled HIGH SPEED TRANSFER MECHANISM, filed February 1, 2000, attorney docket number 4982/7US;

- Application Serial No. 09/610,738, titled MODULAR BACKUP AND RETRIEVAL SYSTEM USED IN CONJUNCTION WITH A STORAGE AREA NETWORK, filed July 6, 2000, attorney docket number 4982/8;
- Application Serial No. 09/744,268, titled LOGICAL VIEW AND ACCESS TO PHYSICAL STORAGE IN MODULAR DATA AND STORAGE MANAGEMENT SYSTEM, filed January 30, 2001, attorney docket number 4982/10;
- Application Serial No. 10/658,095, titled DYNAMIC STORAGE DEVICE POOLING IN A COMPUTER SYSTEM, filed September 9, 2003, attorney docket number 4982/18; and
- Application Serial No. 60/460,234, titled SYSTEM AND METHOD FOR PERFORMING STORAGE OPERATIONS IN A COMPUTER NETWORK, filed April 3, 2003, attorney docket number 4982/35PROV.

BACKGROUND

Backup operations for client data on a storage network are often performed on streams of data which are managed by subclients and sent to a backup drive or media device. Typically, on a given stream, only one sub client can perform a backup at any given time. The concurrency limit for the number of backups that can go to a stream at any given time is one. Indirectly this means that only one backup can be sent to a media or drive at any point.

This limitation has a major drawback. With tape speeds in media increasing and the difference between disk speed and tape speed widening, the tape

throughput is being throttled by the slower disks. This becomes a major issue in a large enterprise where there are many clients with slow, under performing disks with large amounts of data that need to be backed up in a fixed backup window. The only way the backup window can be met is by backing up these clients, each to a different piece of media in different drives. This increases the hardware requirement costs. This also can create a “shoe shining” effect in which the tape is driven back and forth since drive capacity is under-utilized at certain times.

Tape capacity is also growing and data from multiple clients can actually fit on a single piece of media especially if the backup being performed is an incremental backup. Scattering data across many pieces of media is a tape-handling nightmare for backup administrators.

SUMMARY OF THE INVENTION

In accordance with embodiments of the invention, a method is provided for performing a backup operation on a plurality of data streams containing data to be backed up. In one embodiment, the method involves combining the data streams into a single stream of one or more data chunks, including by writing data from more than one of the data streams into at least one data chunk. The combining may be done by multiplexing the data streams. The method further involves transmitting the one or more data chunks over a transport channel to a backup medium and storing the one or more data chunks on the backup medium.

Data from the data streams may be written into a data chunk until the data chunk reaches a predetermined size, or until a configurable time interval has lapsed, or otherwise in accordance with a storage policy as disclosed in some of the pending applications referenced above, and as discussed herein.

During a restore operation or during an operation to create an auxiliary backup copy, the data chunk is retrieved from the backup medium and data from the separate data streams are separated from the data chunk. All data streams written into a data chunk may be separated from each other into separate data stream portions. When the data streams have been multiplexed, separating involves demultiplexing the data streams written into the data chunk. The separated data streams may be restored to a client or further stored as auxiliary copies of the data streams.

In some embodiments, the data streams contain data from a plurality of archive files. Combining the data streams thus may involve writing data from more than one archive files into at least one data chunk, and may further involve writing data from a single archive file into more than one data chunk. In these embodiments, a plurality of tag headers are inserted into the data chunk; each tag header describing data written in the data chunk from a corresponding archive file. Data may be written into a data chunk until the end of an archive file has been reached. When the data chunk is retrieved, from the backup medium, the data from at least one of the archive files is separated from the data chunk, or all the archive files may be separated into separate archive file portions, using the tag headers when necessary to identify and describe the separate archive file portions. The archive file portions may then be restored to a client or may be stored on an auxiliary storage device which may be accessed in turn during a restore operation of a given archive file requested by a client.

In accordance with some embodiments, the invention provides a system for performing a backup operation on a plurality of data streams containing data to be backed up. The system includes one or more receivers for receiving the data streams, a multiplexer for combining the data streams into a combined data stream, a data writer for

writing data from the combined data stream portion of the combined data streams into one or more data chunks, and one or more backup media for storing the one or more data chunks. The system may further include a transport channel for transporting the data chunks from the data writer to the backup media.

In accordance with further aspects of embodiments of the present invention, a data structure is provided for a data chunk stored on a memory device. The data chunk data structure is used by a computer system to backup data and includes a plurality of portions of data from different archive files written into the data chunk from multiplexed data streams containing the archive files and a plurality of tag headers each describing one of the archive file portions written into the data chunk.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 presents a block diagram of a network architecture for a system to perform storage operations on electronic data in a computer network according to an embodiment of the invention.

Fig. 2 presents a flow diagram of a system for multiplexing pipelined data according to an embodiment of the invention.

Fig. 3 presents a flow diagram of a system for multiplexing pipelined data according to an embodiment of the invention.

Fig. 4 presents an exemplary current data format used prior to multiplexing according to embodiments of the invention.

Fig. 5 presents an exemplary media format to support data multiplexing according to embodiments of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention includes methods and systems operating in conjunction with a modular storage system to enable computers on a network to share storage devices on a physical and logical level. An exemplary modular storage system is the GALAXY™ backup and retrieval system and QiNetix™ storage management system available from CommVault Systems of New Jersey. The modular architecture underlying this system is described in the above referenced patent applications, each of which is incorporated herein.

Preferred embodiments of the invention are now described with reference to the drawings. An embodiment of the system of the present invention is shown in Fig. 1. As shown, the system includes a client 85, a data agent 95, an information store 90, a storage manager (or storage management component) 100, a jobs agent 102, a storage manager index 107, one or more media management components (or media agent) 105, one or more media agent indexes 110, and one or more storage devices 115. Although Fig. 1 depicts a system having two media agents 105, there may be one media agent 105, or a plurality of media agents 105 providing communication between the client 85, storage manager 100 and the storage devices 115. In addition, the system can include one or a plurality of storage devices 115.

A client 85 can be any networked client 85 and preferably includes at least one attached information store 90. The information store 90 may be any memory device or local data storage device known in the art, such as a hard drive, CD-ROM drive, tape drive, RAM, or other types of magnetic, optical, digital and/or analog local storage. In some embodiments of the invention, the client 85 includes at least one data agent 95, which is a software module that is generally responsible for performing storage operations on data of a client 85 stored in information store 90 or other memory location.

Storage operations include, but are not limited to, creation, storage, retrieval, migration, deletion, and tracking of primary or production volume data, secondary volume data, primary copies, secondary copies, auxiliary copies, snapshot copies, backup copies, incremental copies, differential copies, synthetic copies, HSM copies, archive copies, Information Lifecycle Management (“ILM”) copies, and other types of copies and versions of electronic data. In some embodiments of the invention, the system provides at least one, and typically a plurality of data agents 95 for each client, each data agent 95 is intended to backup, migrate, and recover data associated with a different application. For example, a client 85 may have different individual data agents 95 designed to handle Microsoft Exchange data, Lotus Notes data, Microsoft Windows file system data, Microsoft Active Directory Objects data, and other types of data known in the art.

The storage manager 100 is generally a software module or application that coordinates and controls the system, for example, the storage manager 100 manages and controls storage operations performed by the system. The storage manager 100 communicates with all components of the system including client 85, data agent 95, media agent 105, and storage devices 115 to initiate and manage storage operations. The storage manager 100 preferably has an index 107, further described herein, for storing data related to storage operations. In general, the storage manager 100 communicates with storage devices 115 via a media agent 105. In some embodiments, the storage manager 100 communicates directly with the storage devices 115.

The system includes one or more media agent 105. The media agent 105 is generally a software module that conducts data, as directed by the storage manager 100, between the client 85 and one or more storage devices 115, for example, a tape library, a hard drive, a magnetic media storage device, an optical media storage device, or other

storage device. The media agent 105 is communicatively coupled with and controls the storage device 115. For example, the media agent 105 might instruct a storage device 115 to perform a storage operation, e.g., archive, migrate, or restore application specific data. The media agent 105 generally communicates with the storage device 115 via a local bus such as a SCSI adaptor.

Each media agent 105 maintains an index cache 110 which stores index data that the system generates during storage operations as further described herein. For example, storage operations for Microsoft Exchange data generate index data. Media management index data includes, for example, information regarding the location of the stored data on a particular media, information regarding the content of the information stored such as file names, sizes, creation dates, formats, application types, and other file-related criteria, information regarding one or more clients associated with the information stored, information regarding one or more storage policies, storage criteria, or storage preferences associated with the information stored, compression information, retention-related information, encryption-related information, stream-related information, and other types of information. Index data thus provides the system with an efficient mechanism for performing storage operations including locating user files for recovery operations and for managing and tracking stored data.

The system generally maintains two copies of the media management index data regarding particular stored data. A first copy is generally stored with the data copied to a storage device 115. Thus, a tape may contain the stored data as well as index information related to the stored data. In the event of a system restore, the index information stored with the stored data can be used to rebuild a media agent index 110 or other index useful in performing storage operations. In addition, the media agent 105 that

controls the storage operation also generally writes an additional copy of the index data to its index cache 110. The data in the media agent index cache 110 is generally stored on faster media, such as magnetic media, and is thus readily available to the system for use in storage operations and other activities without having to be first retrieved from the storage device 115.

The storage manager 100 also maintains an index cache 107. Storage manager index data is used to indicate, track, and associate logical relationships and associations between components of the system, user preferences, management tasks, and other useful data. For example, the storage manager 100 might use its index cache 107 to track logical associations between media agent 105 and storage devices 115. The storage manager 100 may also use its index cache 107 to track the status of storage operations to be performed, storage patterns associated with the system components such as media use, storage growth, network bandwidth, service level agreement (“SLA”) compliance levels, data protection levels, storage policy information, storage criteria associated with user preferences, retention criteria, storage operation preferences, and other storage-related information.

A storage policy is generally a data structure or other information which includes a set of preferences and other storage criteria for performing a storage operation. The preferences and storage criteria may include, but are not limited to: a storage location, relationships between system components, network pathway to utilize, retention policies, data characteristics, compression or encryption requirements, preferred system components to utilize in a storage operation, and other criteria relating to a storage operation. A storage policy may be stored to a storage manager index, to archive media as

metadata for use in restore operations or other storage operations, or to other locations or components of the system.

Index caches 107 and 110 typically reside on their corresponding storage component's hard disk or other fixed storage device. For example, the jobs agent 102 of a storage manager 100 may retrieve storage manager index 107 data regarding a storage policy and storage operation to be performed or scheduled for a particular client 85. The jobs agent 102, either directly or via another system module, communicates with the data agent 95 at the client 85 regarding the storage operation. In some embodiments, the jobs agent 102 also retrieves from the index cache 107 a storage policy associated with the client 85 and uses information from the storage policy to communicate to the data agent 95 one or more media agents 105 associated with performing storage operations for that particular client 85 as well as other information regarding the storage operation to be performed such as retention criteria, encryption criteria, streaming criteria, etc. The data agent 95 then packages or otherwise manipulates the client information stored in the client information store 90 in accordance with the storage policy information and/or according to a user preference, and communicates this client data to the appropriate media agent(s) 100 for processing. The media agent(s) 105 store the data according to storage preferences associated with the storage policy including storing the generated index data with the stored data, as well as storing a copy of the generated index data in the media agent index cache 110.

In some embodiments, components of the system may reside and execute on the same computer. In some embodiments, a client component such as a data agent 95, a media agent 105, or a storage manager 100 coordinates and directs local archiving, migration, and retrieval application functions as further described in Application Serial

Number 09/610,738. These client components can function independently or together with other similar client components.

Data and other information is transported throughout the system via buffers and network pathways including, among others, a high-speed data transfer mechanism, such as the CommVault DataPipe™, as further described in U.S. Patent No. 6,418,478 and Application No. 09/495,751, each of which is hereby incorporated herein by reference in its entirety. Self describing tag headers are disclosed in these applications wherein data is transferred between a flexible grouping of data transport modules each supporting a separate function and leveraging buffers in a shared memory space. Thus, a data transport module receives a chunk of data and decodes how the data should be processed according to information contained in the chunk's header, and in some embodiments, the chunk's trailer. U.S. Patent No. 6,418,478 and Application No. 09/495,751 generally address "logical data" transported via TCP/IP, however, embodiments of the invention herein are also contemplated which are directed to transporting, multiplexing, encrypting, and generally processing block level data as disclosed, for example, in pending Application No. 10/803,542, titled Method And System For Transferring Data In A Storage Operation, attorney docket number 4982/49, which is hereby incorporated herein by reference in its entirety.

As discussed, these applications generally disclose systems and methods of processing logical data. Thus, for example, contiguous blocks of data from a file might be written on a first volume as blocks 1, 2, 3, 4, 5, etc. The operating system of the host associated with the first volume would assist in packaging the data adding additional OS-specific information to the chunks. Thus, when transported and stored on a second volume, the blocks might be written to the second in a non-contiguous order such as

blocks 2, 1, 5, 3, 4. On a restore storage operation, the blocks could (due to the OS-specific information and other information) be restored to the first volume in contiguous order, but there was no control over how the blocks were laid out or written to the second volume. Incremental block level backups of file data was therefore extremely difficult if not impossible in such a system since there was no discernable relationship between how blocks were written on the first volume and how they were written on the second volume.

Thus, in some embodiments, the system supports transport and incremental backups (and other storage operations) of block level data via a TCP/IP (and other transport protocols) over a LAN, WAN, SAN, etc. Additional data is added to the multi-tag header discussed in the applications referenced above which communicates how each block was written on the first volume. Thus, for example, a header might contain a file map of how the blocks were written on the first volume and the map could be used to write the blocks in similar order on the second volume. In other embodiments, each chunk header might contain a pointer or other similar data structure indicating the chunk's position relative to other chunks in the file. Thus, when a file block or other block changed on the first volume, the system could identify and update the corresponding copy of the block located on the second volume and effectively perform an incremental backup or other storage operation.

In the system, for example as in the CommVault Galaxy system, archives are grouped by Storage Policy. Many clients/sub clients can point to the same Storage Policy. Each Storage Policy has a Primary copy and zero or more Secondary copies. Each Copy has one or more streams related to the number of Drives in a Drive Pool.

The system uses a tape media to its maximum capacity and throughput by multiplexing data from several clients onto the same media at the same time. The system

allows for a stream to be reserved more than once by different clients and have multiple data movers write to this same piece of media.

During backup or other storage operations, data from a data agent to a media agent is transferred over a "Data pipeline" as further described herein and in U.S. Patent No. 6,418,478 and Application No. 09/495,751. One or more transport processes or modules, such as the Dsbackup in the CommVault Galaxy system, form the tail end on the Media Agent for the pipeline. For example, in the Galaxy system, the Datamover process running as part of Dsbackup is responsible for writing data to the media. For data multiplexing, many such Data movers belonging to different pipelines have to write to the same piece of media. This can be achieved by splitting the Datamover pipeline process into multiple components including a data receiver, a data writer, and other modules as necessary.

Fig. 2 presents a system for multiplexing pipelined data according to an embodiment of the invention. As shown, Fig. 2 includes various data streams 125 in communication with a transmit data pipeline 130 that includes one or more data receiver modules 135, a multiplexing module 140, additional modules 145, and a data writer module 150. Fig. 2 also includes a transport channel 155 from the transmit pipeline 130 to a receive pipeline 160 that includes a second receiver module 165 and other modules 170.

Backup streams 125 are fed into the transmit pipeline 130. For example, in some embodiments, a backup process, such as the Dsbackup process in the CommVault Galaxy system, packages file data and other data into chunks and communicates the chunks via the backup streams 125. Thus, the transmit pipeline 130 or tail end of the pipeline copies the data received in pipeline buffers from the backup

process via the backup data streams 125. A data receiver 135 processes the data received from each backup stream 125. In some embodiments, there is one data receiver 135 per backup stream 125, thus in the case of multiple backup streams 135, the system might contain multiple data receiver modules 135.

A multiplexing module 140 combines the data received by the receiver module(s) 135 into a single stream of chunks as further described herein. Thus, the multiplexing module 140 may combine data from multiple archive files into a single chunk. Additional modules 145 perform other operations on the chunks of data to be transported such as encryption, compression, etc. as further described herein, in U.S. Patent No. 6,418,478, pending Application No. 09/495,751, and pending Application No. ____/_____, filed November 15, 2004, attorney docket no. 4982/45.

The data writer module 150 communicates the chunks of data from the transmit pipeline 130 over a transport channel 155 to the receive pipeline 160. The transport channel may comprise a buffer, a bus, a fiber optic channel, a LAN, a SAN, a WAN, a wireless communication medium, or other transport methods known in the art. There is generally one data writer 150 per media (not shown) that receives data from multiple data receivers 135 and writes data to the media. The data writer process 150 is generally invoked when the first pipeline is established to use a given media and generally remains running until all the pipelines backing up to this media are finished. The data writer 150 writes the data to media or to the receive pipeline 160 and closes a chunk when the chunk size is reached, the chunk size being a design parameter set to allow only certain size chunks for transmission over the datapipe. In some embodiments, the data writer 150 also updates the Archive Manager tables with the chunk information. A multiplexed chunk thus will contain data from many archive files.

In some embodiments, the transmit pipeline receives data directly from the system's data agents and writes multiplexed data to the media directly without an intervening receive pipeline 160. Thus, in some embodiments, a single pipeline is also contemplated. In embodiments that include both a transmit pipeline 130 and a receive pipeline 160, the receive pipeline 160 processes data received from the transmit pipeline 130 for storage to media, etc. A second data receiver 165 processes data received from the data writer 150 and additional modules 170 which may include encryption, decryption, compression, decompression modules, etc. further process the data before it is written to the storage media by a final data writer module (not shown).

In some embodiments, Data Multiplexing is a property of a Storage Policy. Any storage policy with Data Multiplexing enabled has the ability to start backups for multiple sub clients to run simultaneously to the same media. In some embodiments, a resource manager process on the storage manager allows for multiple volume reservation for media belonging to storage policies with data multiplexing enabled.

During a restore storage operation, the process is essentially reversed. Data is retrieved from the storage media and passed back through the pipeline to the original volume. Thus, during a restore, a data reader module (e.g. – a data receiver directed to also retrieve data from storage) identifies the data by the looking into the tag header of each retrieved chunk. Any offset into the chunk is a relative offset i.e. when restoring data from a given archive file all the data buffers encountered from a different archive file should not be counted into the offset calculation and should be thrown out. Data within each volume block size of data will contain data from different Archive files. The tag header also contains the archive file id. In addition, all the offsets stored are

relative offset within an archive file and does not depend on actual physical location on the tape or other storage media.

A more detailed description of data multiplexing according to embodiments of the invention is now described:

A single backup is made up of one or more archive files. An archive file is made up of the smallest restorable component called the "Chunk". The chunk always belonged to only one archive file. With data multiplexing a chunk interleaves pipeline buffers from different pipelines. A tag header written for each buffer of data will uniquely identify the data to the archive file. The tag header contains the archive file id (serial ID) from the database corresponding to the archive file being backed up.

In some embodiments, for example in the CommVault Galaxy system, one or more modules in the pipeline, such as the DsBackup module, package data or otherwise retrieve data from a primary volume to be backed up and from the pipeline, and sends the data to the DataMover or receive pipeline. DsBackup also initializes indexes and updates the index cache every time it receives a file header from a client. DataMover responsibility is to organize the data received from the dsBackup into chunks, start a new chunk when the size of the chunk reaches the predetermined value, update the archive manager tables information about the chunks and their location on that tape, also handle end of media conditions and media reservations. DataMover uses the MediaFileSystem object, for example I/O system API calls of a media agent or other system component, to write data on to the tape and read data from the tape. MediaFileSystem has a Write buffer and data is written onto the tape when this write buffer is filled with data.

With the new data Multiplexing model of DataMover, the previous DataMover modules and their functionalities undergo changes.

Referring now to Fig. 3, consider clients C1, C2, C3 and C4 are backing up at the same time and their data are getting multiplexed. DsBackup instantiates an object of Data Receiver and initializes the object. Media is mounted as a part of the initialization. One Data Writer object is instantiated for every media group Id.

Considering that all the four clients share the same media group id; only one Data Writer object is instantiated and all the four Data Receiver objects share the same object to write to the media.

Each Data Receiver writes a tag portion immediately by calling the Data Writer's Write () method. Data Writer has an internal buffer, which is the same as the selected block size. When this buffer is full, the buffer is locked and emptied to the media. While this write operation is ongoing to the media, the second buffer will be ready to accept data from the Data Receiver. The thread, which calls the write on Data Writer, will return from the function call when the Media IO is complete. Meanwhile, the second buffer fills. These double buffers are guarded with appropriate semaphores to ensure proper concurrent access.

The Write operation is a blocking call and returns after completing the write. The Data Writer Write API takes in the archive file id as the parameter and once the write is completed, the physical offsets are updated in a list maintained by the Data Writer object accordingly. When the size of the chunk exceeds the pre-determined size, the chunk is automatically closed by writing a file mark, updating the archive manager tables in the storage manager or at the media agent, and also updating the physical offsets in the list or index maintained by the data writer object to track multiplexed storage files. The Data Writer object is responsible for handling the end of media condition and the Data Receiver does not generally require any knowledge about it.

As previously discussed, only the data writer object generally knows the chunk closure, but there are conditions where the close chunk operation could be needed because of a CLOSE ARCHIVE FILE message sent by the client. This means that the system may need to close the chunk though the size of the chunk may not have reached the predetermined size. When a CLOSE ARCHIVE FILE message is received from the client, DsBackup calls into Data Receiver Close that in turn calls the Data Writer Close. This close waits for a pre-determined amount of time for the chunk to get close on its own as the other clients may be still pumping in data to the chunk. If after the pre-determined time the chunk is not closed, the chunk is closed forcefully by writing a file mark and updating the appropriate index cache. The only side effect this could result in is that the chunk may not be as big as the pre-determined size as the close chunk has been force fully done. The pre-determined time for wait can be made configurable or can be made a variable parameter depending on the client type. With this new model there can be a situation that the tag header gets split and is spanned between two data buffers on the tape. This is generally addressed during the restore of data.

The following cases illustrate exemplary backup scenarios and considerations according to embodiments of the invention:

1. Initialization of DataWriter. During the Initialization, the active media for the media group is mounted. This method returns success only if the media is mounted correctly. If the media is already mounted, this method just returns success along with the volume Id of the mounted media. This may be required for logging information for the Data Receiver in some embodiments.

2. **CreateArchiveFile:** In this method, an Archive file header is written on to the media. This uses the special tag header which identifies the data in the tag portion as an archive file header.
3. **WriteToMedia:** This method returns information to the upper layer if the write is successful or not. Method returns information such as, end of the chunk, various media errors, Media is full etc. There is no other way to indicate these conditions other than as a return value in this method.
4. **CloseArchiveFile:** This method closes the archive file by writing an Archive file trailer to the media. This again has a specialized tag header which identifies the data as Archive file trailer. Close Archive file trailer does not return immediately, There is a configurable time interval for which the writing to the current chunk continues. The current chunk will be closed when all the archive files in this chunk gets over or after the above time out interval from the first archive file close request which comes in, whichever is the earliest.

There is generally no need of any call back methods to Data Receiver from Data Writer. All communication from Data Writer to Receiver should be through return values of the functions called in.

Restores of the multiplexed data are often less complicated since restores are generally not multiplexed as the back-ups. But the aim during the restores is to seek to the offsets and restore without looking into the tag headers in all of the data. Data Reader object is instantiated during restore. The parameter for this object remains the same as the current DataMover object. The client opens the required archive file by specifying the archive file id. Then the client sends the seek offset. The Data Reader object queries the archive manager to determine the chunk number that needs to be opened and the volume

that should be mounted to seek to the give offset. Once the media is mounted the media is positioned to the correct file marker so as to open the chunk. Once the chunk header is read and discarded, data is read block by block and the size of the block is the same as the one that was used during the write. Every time a block of data is read all tag headers are examined to determine whether it contains the data of the archive file that we are looking for. This is done by traversing the buffer read in and looking through the tag headers. If it contains any other archive file's data, that tag portion is discarded and the next header is read. If the tag portion contains the data of the archive file that is being searched, then a check is done to see if the tag portion contains the offset that is being searched. If it does not contain the offset, this tag portion is skipped but the physical offset calculations are incremented appropriately. Once the correct block that contains the offset is reached, the data buffer pointer is positioned properly and the success is returned to the caller.

Once the seek is successful, a data reader/retriever module in the pipeline, such as the FsRestoreHead module in the Galaxy system, requests a read with the read size equal to the size of the tag header. The process looks into the tag header to determine the size of the data that has to be read and requests a read with the size of the data. The restore happens in this fashion. The Data reader will have a buffer more than equal the size of one pipe line header as it may need to buffer the data during a read request. The Data Reader also takes care of the case of tag headers that may have spanned between two data blocks.

There is also metadata that is written on to the tape (or other media) during back up to trouble shoot problems and also enable disaster recover programs, such as CommVault's Dr-restore program which retrieves data from backups. During backup every time a chunk is closed, a file marker is written. After this a data block is

constructed containing information, e.g., the list of archive file id's whose data is contained in the recently closed chunk and their physical offsets and the size within this chunk. A file marker follows this data and does not generally make any kind of update to the database. In order to facilitate the disaster recovery tool functionality, we also indicate which of the archive file ids were closed in the current chunk.

The data format on Media changes with Data Interleaving/Multiplexing. An exemplary current data format and related data structures used prior to multiplexing according to embodiments of the invention is shown in Fig. 4, and contains the following fields and properties:

ArchiveFile Header

```
{
    "ARCHIVE_FILE_HEADER_VERSION 2 \n "
    "HeaderSize"
    "FileId"
    "FileSeqNo"
    "StreamId"
    "ArchiveFileGroup "
    "AgroupId "
    "CopyId "
    "AppId"
    "JobId"
    "AppType "
    "BackupLevel"
    "BackupTime"
    "FileType"
    "SubAppType"
    "ClientName "
    "ArchiveFileName "
    "GalaxyVersion "
    "ARCHIVE_FILE_HEADER_END 2"
}
```

ArchiveFile Trailer

```
{
    "ARCHIVE_FILE_TRAILER_VERSION 2 "
    "TrailerSize "
    "FileId "
    "FileSeqNo"
```

```
"StreamId "
"ArchiveFileGroup "
"AgroupId "
"CopyId "
"AppId "
"JobId "
"AppType "
"BackupLevel "
"BackupTime "
"FileType "
"SubAppType "
"ClientName "
"ArchiveFileName "
"GalaxyVersion "
"PhysicalFileSize "
"LogicalFileSize "
"ARCHIVE_FILE_TRAILER_END 2 ",
}
```

ChunkHeader

```
{
    "CHUNK_HEADER_VERSION 2 "
    "HeaderSize"
    "FileId"
    "FileSeqNo"
    "ChunkSeqNo"
    "ChunkVol"
    "ChunkVolFM"
    "TagHeaderOffset"
    "CHUNK_HEADER_END 2"
}
```

ChunkTrailer

```
{
    "CHUNK_TRAILER_VERSION 1"
    "TrailerSize "
    "FileId"
    "FileSeqNo"
    "ChunkSeqNo"
    "ChunkVol"
    "ChunkVolFM "
    "ChunkLogicalSize "
    "ChunkPhysicalSize "
    "CHUNK_TRAILER_END"
}
```


An exemplary media format to support data multiplexing according to embodiments of the invention is shown in Fig. 5:

When data multiplexing is enabled, other elements of the previous system also change in some embodiments as further described below. For example, Auxiliary Copy currently copies data chunk by chunk within an Archive File. The assumption is that the data within a chunk belongs to the same archive file. This is no longer true with data multiplexing. In embodiments where data multiplexing is supported, Auxiliary Copy allows two forms of copy mechanism: Simple Copy (copy whole chunk for all or part of archive files) and De-Multiplexed copy (archive file by archive file; only if source is magnetic).

In a simple copy, Auxiliary Copy creates a list of archive files that needs to be copied and copies then chunk by chunk and volume by volume. Data from different archive files will be copied at the same time to the secondary copy. This is faster, but the resultant copy will have the data still interleaved as the original copy.

In a de-multiplexed copy, Auxiliary Copy will copy data archive file by archive file. The result being that the system may go over the same set of media for each archive file discarding data encountered from a different archive file. This approach is slow and inefficient but the secondary copy has contiguous data for each archive file.

The system uses flags and other signaling mechanisms, for example flag `deMultiplexDataOnCopy` on the `ArchGroupCopy` object, to dictate the choice of copy mechanism. Archive Manager will pass down a list of Archive Files to be copied to the secondary copy, if the copy is setup for a Simple Copy. If the DeMultiplexing is supported on the Copy, `AuxCopyMgr` will pass down a single archive file to be copied.

Auxiliary Copy first creates all the headers for all archive files being copied and then starts the copy. A set of messages will be sent over the pipeline for creating these headers and in turn DSBackup will call DmReceiver create which will add archive file information to the dmreceiverinfo structure maintained in DmWriter. In some embodiments, Auxiliary Copy also supports client based copies, where archive files belonging for a set of clients will be copied. In other embodiments, a synthetic full backup combines archive files backed up from a single sub client and creates a full backup of all the incremental changes since the last full backup. The new archive file being created as part of Synthetic full can be multiplexed with other backups.

Systems and modules described herein may comprise software, firmware, hardware, or any combination(s) of software, firmware, or hardware suitable for the purposes described herein. Software and other modules may reside on servers, workstations, personal computers, computerized tablets, PDAs, and other devices suitable for the purposes described herein. Software and other modules may be accessible via local memory, via a network, via a browser or other application in an ASP context, or via other means suitable for the purposes described herein. Data structures described herein may comprise computer files, variables, programming arrays, programming structures, or any electronic information storage schemes or methods, or any combinations thereof, suitable for the purposes described herein. User interface elements described herein may comprise elements from graphical user interfaces, command line interfaces, and other interfaces suitable for the purposes described herein. Screenshots presented and described herein can be displayed differently as known in the art to input, access, change, manipulate, modify, alter, and work with information.

While the invention has been described and illustrated in connection with preferred embodiments, many variations and modifications as will be evident to those skilled in this art may be made without departing from the spirit and scope of the invention, and the invention is thus not to be limited to the precise details of methodology or construction set forth above as such variations and modification are intended to be included within the scope of the invention.

APPENDIX A

Appendix A describes data structures, software modules, and other elements of the system according to embodiments of the invention, such as in the CommVault Galaxy system.

ArchChunkTable

```

table arch Chunk
(
    id          bigint,          // 64 bit integer for unique chunk identification
    commCellId  integer,
    volumeId    integer,
    fileMarkerNo integer,
    createTime  integer,
    chunkVersion integer,

    // INDEXES:
    primary key (archChunkId),
    foreign key (commCellId) references commCell(id),
    foreign key (volumeId) references MMS2Volume(VolumeId),

);

```

ArchChunkMapping table

```

Table archChunkMapping
(
    archChunkId    bigint,
    archFileId     integer,
    commCellId     integer,
    archCopyId     integer,
    chunkerNumber  integer,
    physicalOffset  bigint,
    logicalOffset  bigint,
    physicalSize    bigint,
    logicalSize     bigint,

    primary key (archChunkId, archFileId, commCellId)
    foreign key (archChunkId, commCellId) references archChunk (id, commCellId),
    foreign key (archFileId, commCellId, archCopyId) references archFileCopy(archFileId,
commCellId, copy),

```

```

);

create table archGroupCopy
(
    id                serial,
    archGroupId       integer,
    commCellId        integer,
    copy              integer,
    name              varchar(MAX_DEFAULT_REC_SIZE,
                              MIN_DEFAULT_REC_SIZE),
    compressionType   integer,
    flags             integer,
    maxMultiplex       integer,
    isActive          integer,
    type              integer,
    startTime          integer,
    waitForIfOffline   integer,
    waitForIfBusy      integer,

    primary key (id, commCellId),
    foreign key (archGroupId) references archGroup(id),
    foreign key (commCellId) references commCell(id)
);

```

- Every Chunk has a Unique 64 bit id used as a counter to track chunks and perform other functions
- Multiple archive file may be part of a single chunk
- The ArchChunkMapping table determines archive files contained in each chunk
- DmReceiver during restores queries archive manager and get the information for the chunk required by providing the archive file id and the physical offsets.
- All data contained in a chunk belongs to the same copy.
- An integer value in the ArchGroupCopy table defines the multiplexing factor, and determines how many clients can backup to this copy at the same time. This factor is applicable for all streams within the copy.
- deMultiplexDataOnCopy flag indicates whether Auxiliary Copy should de-multiplex data when creating a secondary copy. The flag on the secondary copy is what is taken into consideration.

1.1.1 Creation of a Chunk

- DmWriter maintains a list of archive files that were written as part of the chunk
- When the chunk is closed, DmWriter makes a AMDS calls to close a chunk and will pass along a list of archive files that made up the chunk
- Archive Manager creates the necessary entries in the ArchChunkMapping table and the archChunk table entries

1.1.2 API's that changes in Archive Manager and AMDS

(1) `ArchiveManagerDS::getAfileInfo(GetAfileInfoArgs_t * args, GetAfileInfoRet_t * ret)`

Following structures defined in source/include/ArMgr/AmdsArgs.h and CVABasic.h change:

```
typedef struct {
    unsigned long afileNumber;
    unsigned long storagePolicyNumber;
    unsigned long commCellId;
    int needArchChunkId;
} GetAfileInfoArgs_t;
```

```
typedef struct {
    unsigned long afileNumber;
    unsigned long agroupNumber;
    unsigned long commCellId;
    string name;
    string objName1;
    string objName2;
    unsigned long objVersion;
    unsigned long numAppId;
    unsigned long fileType;
    unsigned long backupTime;
    unsigned long seq;
    unsigned long flags;
    longlong_t createJobId;
    unsigned long objType;
    unsigned long backupLevel;
    unsigned long isValid;
    unsigned long streamNum;
    ULONGLONG firstChunkId;
} GetAfileInfoRet_t;
```

(2) `int ArchiveManagerDS::closeChunk(CloseChunkArgsDS_t * args, CloseChunkRetDS_t * ret)`

Following structures defined in source/include/ArMgr/AmdsArgs.h and CVABasic.h change:

```
struct CloseChunkArgsDS_t {
    unsigned long commCellId;
    unsigned long archCopyId;
    ULONGLONG archChunkId;
```

```

        int needNextArchChunkId;
        unsigned long volumeId;
        unsigned long fileMarkerNo;
        unsigned long numberOfAfiles;
        ChunkArchFileMsg_t *afileArray;

        CloseChunkArgsDS_t() : needNextArchChunkId(0) {}
};

typedef struct {
        ULONGLONG newChunkId;
} CloseChunkRetDS_t;

struct ChunkArchFileMsg_t {
        unsigned long archFileId;
        unsigned long chunkNumber;
        u_longlong_t physicalOffset;
        u_longlong_t logicalOffset;
        u_longlong_t physicalSize;
        u_longlong_t logicalSize;
};

typedef struct ArchiveFileId_t
{
        unsigned long commCellId;
        unsigned long agroupNumber;
        unsigned long afileNumber;
} ArchiveFileId_t;

```

1.2 Resource Manager

In previous versions of the system which did not support multiplexing, a single Volume (defined by MMS2Volume table) could be reserved only once – the same held true for drive reservations. This behavior changes to support data multiplexing. A given volume can be reserved multiple times by different jobs for writes. The number of times a volume can be reserved for writes is determined by the an index value, such as the howManyCanMultiplex value set in the ArchGroupCopy table. The same stream also now can be reserved multiple times due to this change.

table archStream

```
(
    commCellId            integer,
    archGroupId            integer,
    archGroupCopyId        integer,
    stream                 integer,
    mediaGroupId            integer,
    isActive               integer,
    PreEmptable            integer,
    lastAfileIdCopied       integer,
    howManyReservations     integer,          // Currently how
                                           // many reservations
    primary key (stream, commCellId, archGroupId, archGroupCopyId),
    foreign key (commCellId) references commCell(id),
    foreign key (archGroupId) references archGroup(id)
    foreign key (mediaGroupId) references MMS2MediaGroup(MediaGroupId)
);;
```

table archStreamReserve

```
(
    jobId                 LONGLONG,
    priority               integer,
    reserveTime            TIME_T,
    interruptJobId          LONGLONG,
);
```

- Resource Manager allows up to “howManyCanMultiplex” number of reservations on a volume, stream and drive when reservation type is “WRITE”
- *Jobs running as part of a copy that support Multiplexing cannot generally be interrupted*
- *These jobs can be suspended or killed*
- The Mark media full option is supported once per media group.
- Streams availability is no longer based on the “inuse” flag but on “howManyReservations” field. If this value equals the “howManyCanMultiplex” value for the ArchGroupCopy then that stream cannot generally be reserved.
- *Resource Manager will reserve a specific drive.*
- The selection of the drive is based on the volumeid and the mediaid set on the drive table. If the requested media is already cache mounted in the drive and then that drive is reserved.
- Resource Manager disallows any reservation of a client that wants to participate in data multiplexing if the client has not been upgraded to support multiplexing.

1.3 Media manager

In previous versions of the system that did not support multiplexing, Media Manager

mounted a particular volume into any available drive and set the drive id in the

reservation tables. This will change as now the reservation will be made for a specific drive. When the mount request is received, Media Manager determines that the drive that is reserved for this job and mounts the media into that drive. If the media is cache mounted in a different drive then the reservation will be switched to that drive if that drive is not reserved.

1.4 DSBBackup

With data multiplexing data of different clients will belong to the same chunk and hence each data block in the chunk has to be identified uniquely so as to perform the restore. This will be achieved by storing the archive file id into the tag header that is associated with every data block. The arch file id will uniquely identify the data block and the database can be used to determine the client to which the data belongs. The structure of an exemplary tag_header is given below.

```
typedef struct
{
    char tag_magic[8];           // version of the tag. => Always use
                                // FILLTAGMAGIC to fill the tag_magic
    ULONGLONG block_offset;      // to be filled by client.
                                // offset within archive
    ULONGLONG block_number;      // filled by client - block no
                                // within afid
    ULONGLONG functional_header_number; // sequence no of fun.header
    unsigned long buf_type;      // type of the buffer -
                                // MSB bit indicates data/functional
                                // header / PL_NAS_CMDS etc.
                                // 0-data 1-functional header
    unsigned long actual_block_size; // size of the data associated with
                                // this header
    unsigned long compressed_block_size; // compressed size of the data
                                // associated with this header
    unsigned long validity_bits;  // flags
    unsigned long buffer_validity; // invalid buffer
    unsigned long data_offset;    // offset of data within the block
    unsigned long next_header_offset; // offset of next header. 0 means none
    unsigned long compression_scheme; // scheme of compression to be applied
}
```

```

// 0 gzip
// 111 (binary) no compression
unsigned char scatter_gather; // wont be used in vldb 98
// There are 3 extra bytes here for alignment
unsigned long compressed_data; // data is compressed y/n ? This filed is
// no longer reliable because of a difference
// on UNIX
unsigned char restore_backup; // data stream of backup==0,restore==1
unsigned char signature_dependency; // 0 not dependent on signature
// 1 dependent on signature

char spare[16 ];
}tag_header_t ;

```

The field validity_bits has been renamed to archive_file_id to store the archive file as shown below.

```

typedef struct
{
    char tag_magic[8]; // version of the tag. => Always use
    // FILLTAGMAGIC to fill the tag_magic
    ULONGLONG block_offset; // to be filled by client.
    // offset within archive
    ULONGLONG block_number; // filled by client - block no
    // within afid
    ULONGLONG functional_header_number; // sequence no of fun.header
    unsigned long buf_type; // type of the buffer -
    // MSB bit indicates data/functional
    // header / PL_NAS_CMDS etc.
    // 0-data 1-functional header
    unsigned long actual_block_size; // size of the data associated with
    // this header
    unsigned long compressed_block_size; // compressed size of the data
    // associated with this header
    unsigned long archive_file_id;
    unsigned long buffer_validity; // invalid buffer
    unsigned long data_offset; // offset of data within the block
    unsigned long next_header_offset; // offset of next header. 0 means none
    unsigned long compression_scheme; // scheme of compression to be applied
    // 0 gzip
    // 111 (binary) no compression
    unsigned char scatter_gather; // wont be used in vldb 98
    // There are 3 extra bytes here for alignment
    unsigned long compressed_data; // data is compressed y/n ? This filed is
    // no longer reliable because of a difference
    // on UNIX
    unsigned char restore_backup; // data stream of backup==0,restore==1
    unsigned char signature_dependency; // 0 not dependent on signature

```

```
                                // 1 dependent on signature
    char spare[16 ];
    }tag_header_t ;
```

The archive file id is filled by the pipe layer of the client during backup. The tag header is written on to the media in the same format without any modification. During restore Data Reader reads the tag header to get find the archive file and in turn determine whether the data associated with that restore is required for the current restore or not.

1.5 DataMover (Windows Implementation)

Datamover is responsible of writing the data transferred over the pipeline to the media. With data multiplexing Datamover gets split into two components, Data Receiver (DmReceiver) and Data Writer (DmWriter). DsBackup invokes an instance of DmReceiver object. The DmReceiver object internally checks for the DmWriter's existence for the requested MediaGroupId. If the DmWriter is not present then a new instance of the DmWriter is created and cached in a DmWriter map. This map is maintained in CVD's context of the media gent and is accessible to all DmReceiver. DmWriter maintains an internal buffer corresponding to the volume block of the data per DmReceiver. The volume block size is determined from the media type being used. Write on the DmReceiver will call DmWriter write. DmWriter will copy the pipeline buffers internally for aligning it to the volume block in the ReceiverInfo structure.

DataMoverBase class is the class that implements the functionality of DataWriter. Since this class will be used for both backup and restore it was given a generic name "DataMoverBase"

In the above classes, Data Receiver is a thin layer which in many cases calls the DataWriter methods directly.

CLAIMS

What is claimed is:

1. A method for performing a backup operation on a plurality of data streams containing data to be backed up, the method comprising:
 - combining the data streams into a single stream of one or more data chunks, including writing data from more than one of the data streams into at least one data chunk;
 - transmitting the one or more data chunks over a transport channel to a backup medium; and
 - storing the one or more data chunks on the backup medium.
2. The method of claim 1, wherein combining the data streams comprises multiplexing the data streams.
3. The method of claim 1, wherein the data streams contain data from a plurality of archive files, and wherein combining the data streams comprises writing data from more than one archive files into at least one data chunk.
4. The method of claim 3, wherein combining the data streams comprises writing data from a single archive file into more than one data chunk.
5. The method of claim 3, comprising inserting a plurality of tag headers into the at least one data chunk, each tag header describing data written in the data chunk from a corresponding archive file.
6. The method of claim 3, wherein combining the data streams comprises writing data into a data chunk until the end of an archive file has been reached.

7. The method of claim 3, comprising retrieving the data chunk from the backup medium and separating data from at least one of the archive files from the data chunk.

8. The method of claim 7, comprising separating the data from the archive files written into the data chunk from each other into separate archive file portions.

9. The method of claim 8, comprising storing the separate archive file portions on an auxiliary storage device.

10. The method of claim 9, comprising accessing the auxiliary storage device during a restore operation of a given archive file.

11. The method of claim 1, wherein combining the data streams comprises writing data into a data chunk until the data chunk reaches a predetermined size.

12. The method of claim 1, wherein combining the data streams comprises writing data into a data chunk until a configurable time interval has lapsed.

13. The method of claim 1, wherein combining the data streams comprises writing data into a data chunk in accordance with a storage policy.

14. The method of claim 1, comprising retrieving the data chunk from the backup medium.

15. The method of claim 14, comprising separating the data from at least one of the data streams from the data chunk.

16. The method of claim 15, comprising separating the data from all data streams written into the data chunk from each other into separate data stream portions.

17. The method of claim 15, wherein combining the data streams comprises multiplexing the data streams and wherein separating comprises demultiplexing the data streams written into the data chunk.

18. A system for performing a backup operation on a plurality of data streams containing data to be backed up, the system comprising:

one or more receivers for receiving the data streams;

a multiplexer for combining the data streams into a combined data stream;

a data writer for writing data from the combined data stream portion of the combined data streams into one or more data chunks; and

one or more backup media for storing the one or more data chunks.

19. The system of claim 18, comprising a transport channel for transporting the data chunks from the data writer to the backup media.

20. A data structure for a data chunk stored on a memory device, the data structure being used by a computer system to backup data, the data structure comprising:

a plurality of portions of data from different archive files written into the data chunk from multiplexed data streams containing the archive files; and

a plurality of tag headers each describing one of the archive file portions written into the data chunk.

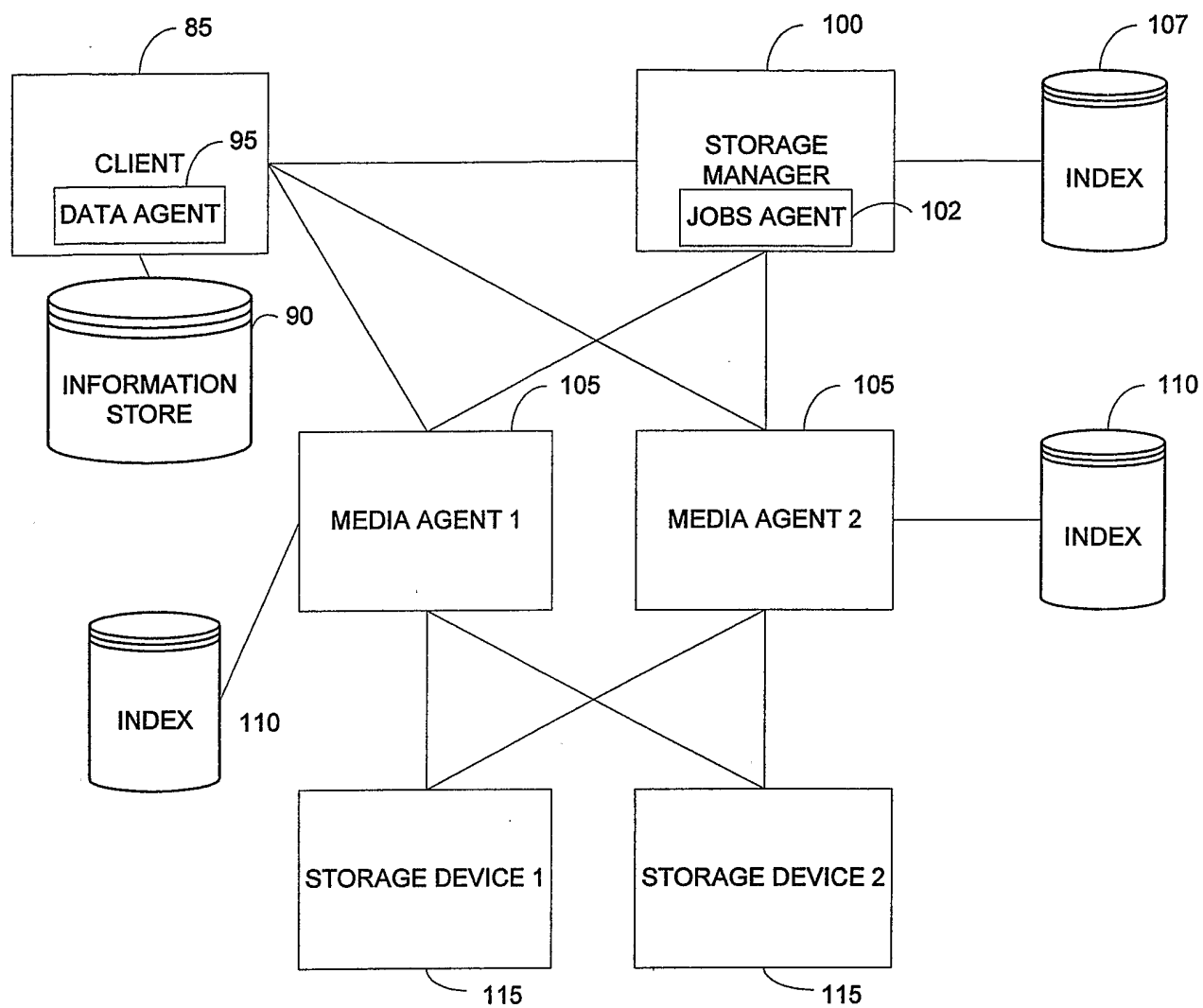


FIG. 1

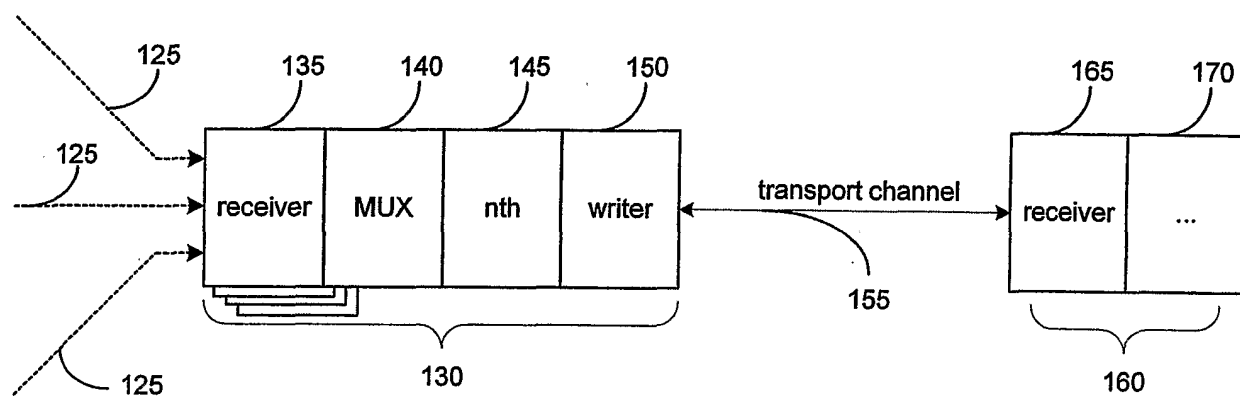


Fig. 2

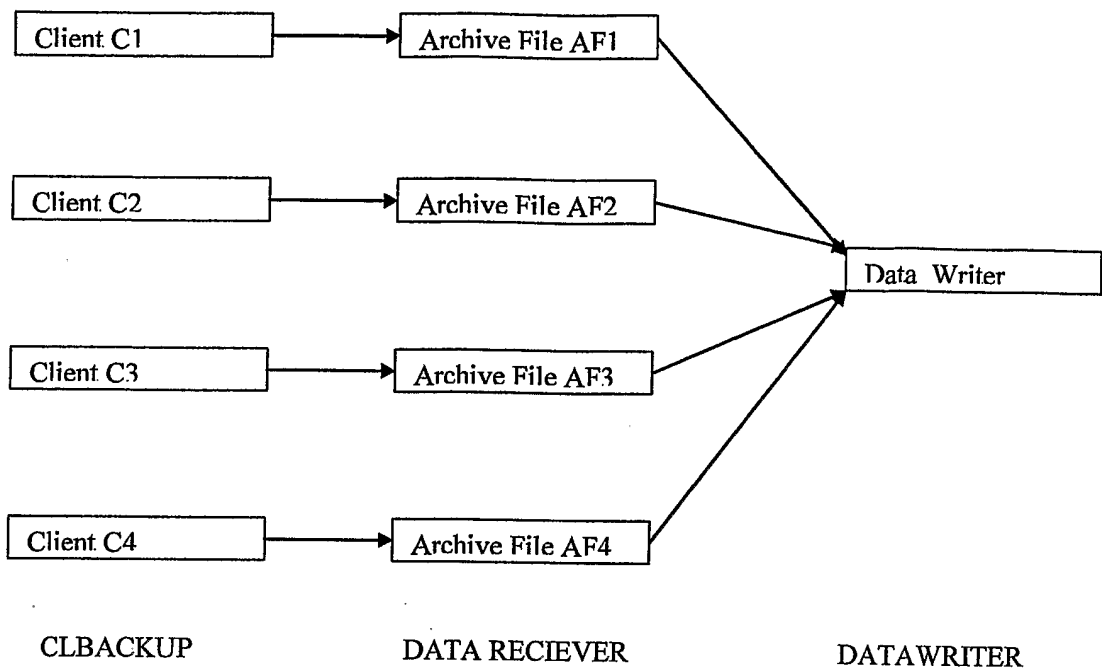


Fig. 3

Archive File Header
FileMarker
Chunk Hedaer
Data For AF1
Data For AF1
Chunk Trailer
FileMarker
ArchiveFiler Trailer
File Marker

Fig. 4

Chunk header
Tag Header for AF1
Data For Af1
Tag Header for AF2
Data for Af2
Tag Header for AF1
Data for Af1
Tag Header for AF2
Data for Af2
File Marker-Chunk
Chunk Map Trailer
FileMarker-meta data

Fig. 5