US009575990B2

(12) **United States Patent**
Thomsen et al.

(10) **Patent No.: US 9,575,990 B2**
(45) **Date of Patent: *Feb. 21, 2017**

(54) **PARTITIONING DATA WITHIN A DISTRIBUTED DATA STORAGE SYSTEM USING VIRTUAL FILE LINKS**

(75) Inventors: **Dirk Thomsen**, Heidelberg (DE); **Ivan Schreter**, Malsch (DE)

(73) Assignee: **SAP SE**, Walldorf (DE)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 72 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **13/324,826**

(22) Filed: **Dec. 13, 2011**

(65) **Prior Publication Data**

US 2013/0117528 A1 May 9, 2013

Related U.S. Application Data

(63) Continuation-in-part of application No. 13/290,835, filed on Nov. 7, 2011.

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F 17/30233*** (2013.01)

(58) **Field of Classification Search**
CPC G06F 17/30194; G06F 17/30233; G06F 17/30079; G06F 17/30315
USPC 707/610
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

2010/0106934	A1*	4/2010	Calder et al.	711/173
2010/0235335	A1*	9/2010	Heman et al.	707/703
2011/0072059	A1*	3/2011	Guarraci	707/823
2011/0099266	A1*	4/2011	Calder	G06F 8/65 709/224

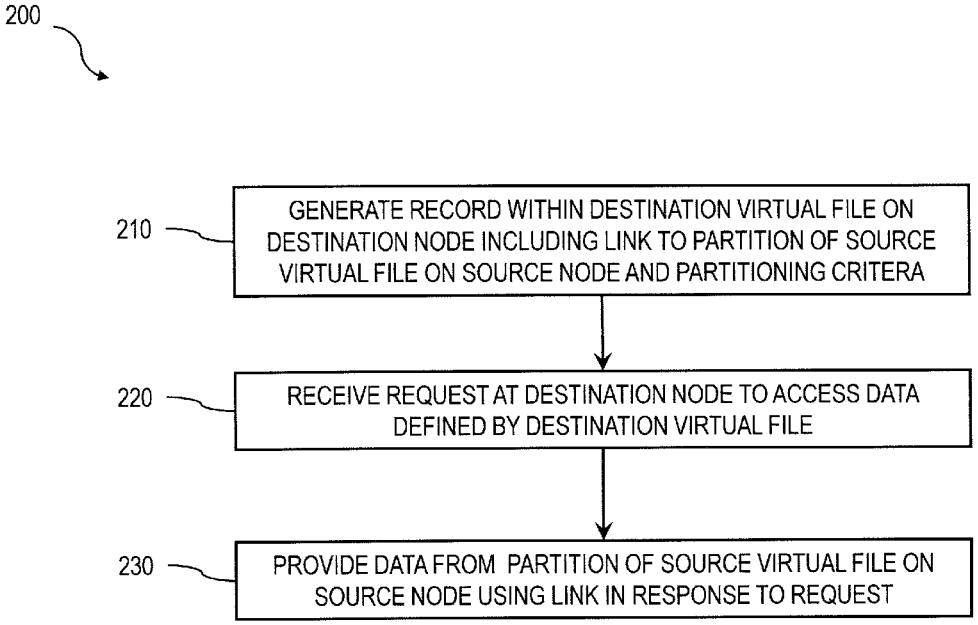
* cited by examiner

Primary Examiner — Kris Andersen
(74) *Attorney, Agent, or Firm* — Mintz Levin Cohn Ferris Glovsky and Popeo, P.C.

(57) **ABSTRACT**

A record within a destination virtual file is generated on a destination node of a distributed data storage system. The record comprises (i) a link directed to a partition of a source virtual file stored on a source node and (ii) partition criteria characterizing the partition. The source virtual file is mapped to a chain of linked pages stored in a page buffer of the distributed data storage system and the partitioning criteria is used by at least one of the source node and the destination node to identify data associated with the partition. A request is later received at the destination node to access data defined by the destination virtual file. Data is provided, in response to the request, from the partition of the source virtual file stored on the source node using the link and the partitioning criteria. Related apparatus, systems, techniques and articles are also described.

15 Claims, 4 Drawing Sheets



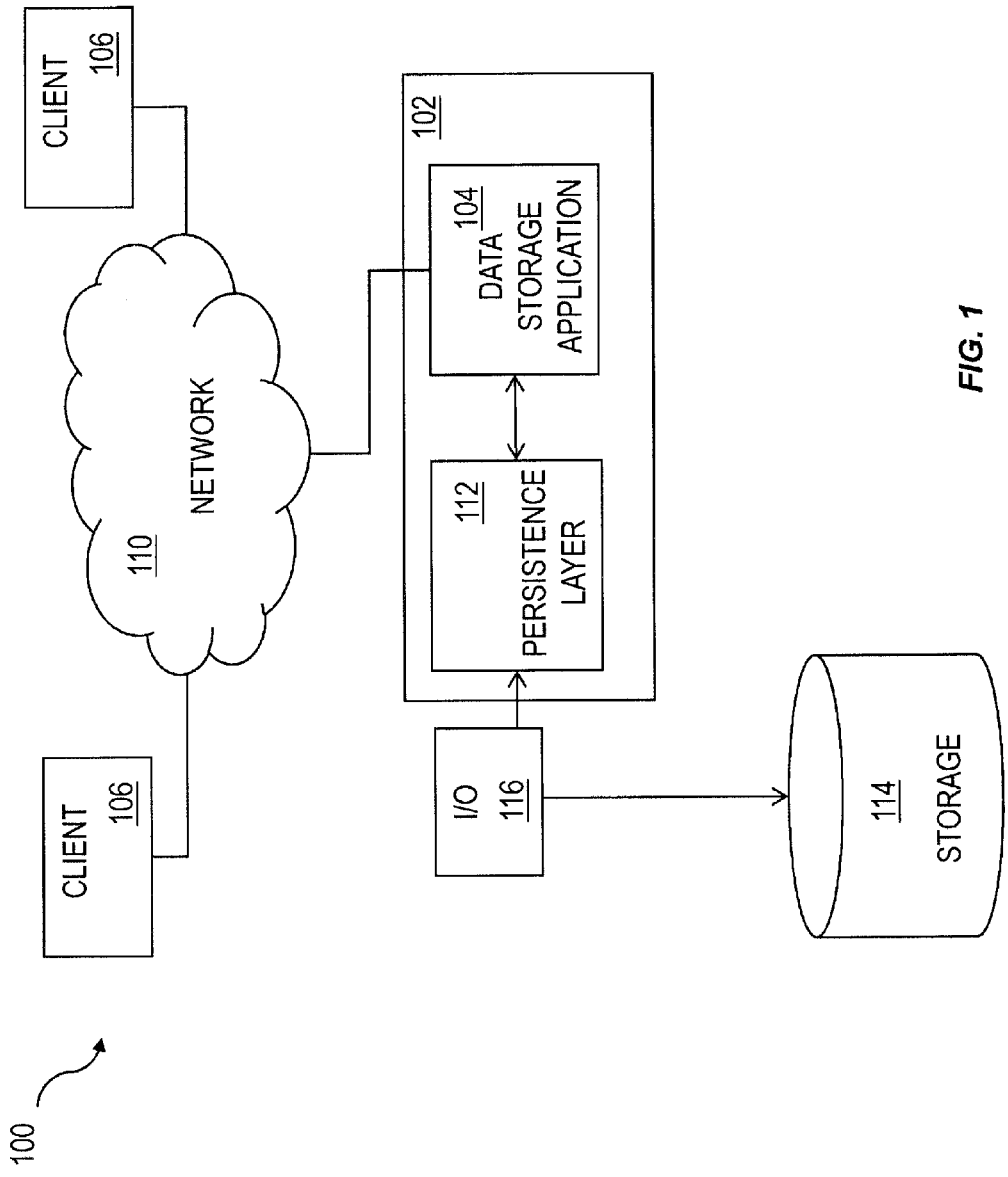

CLIENT
106
CLIENT
106
NETWORK
110
102
DATA STORAGE APPLICATION
104
PERSISTENCE LAYER
112
I/O
116
STORAGE
114
100

FIG. 1

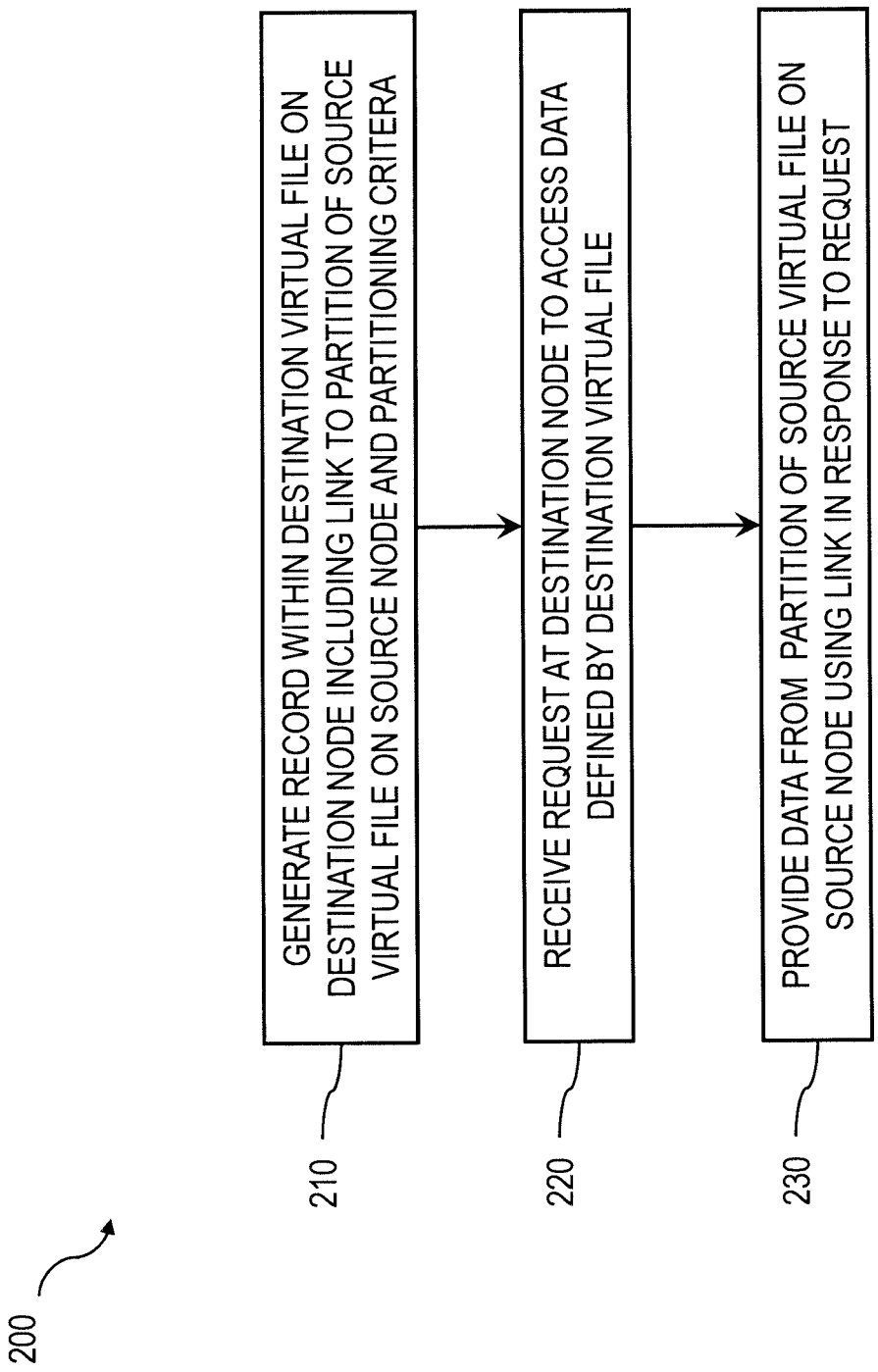

200
210
GENERATE RECORD WITHIN DESTINATION VIRTUAL FILE ON DESTINATION NODE INCLUDING LINK TO PARTITION OF SOURCE VIRTUAL FILE ON SOURCE NODE AND PARTITIONING CRITERA
220
RECEIVE REQUEST AT DESTINATION NODE TO ACCESS DATA DEFINED BY DESTINATION VIRTUAL FILE
230
PROVIDE DATA FROM PARTITION OF SOURCE VIRTUAL FILE ON SOURCE NODE USING LINK IN RESPONSE TO REQUEST

FIG. 2

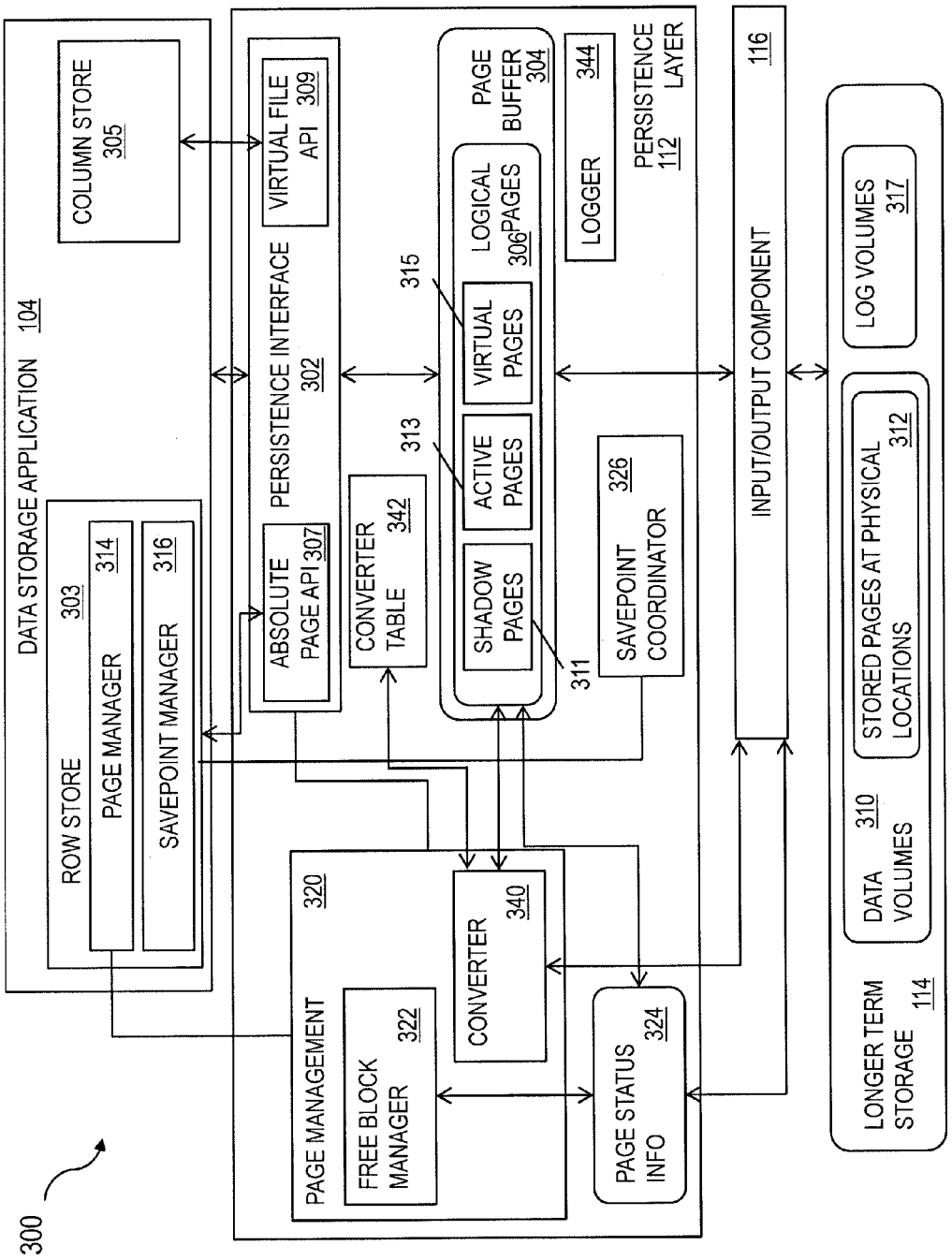

300
DATA STORAGE APPLICATION 104
COLUMN STORE 305
ROW STORE 303
PAGE MANAGER 314
SAVEPOINT MANAGER 316
PERSISTENCE INTERFACE 302
VIRTUAL FILE API 309
ABSOLUTE PAGE API 307
CONVERTER TABLE 342
315
313
311
PAGE BUFFER 304
LOGICAL PAGES 306
VIRTUAL PAGES
ACTIVE PAGES
SHADOW PAGES
LOGGER 344
PERSISTENCE LAYER 112
SAVEPOINT COORDINATOR 326
INPUT/OUTPUT COMPONENT 116
PAGE MANAGEMENT 320
FREE BLOCK MANAGER 322
CONVERTER 340
PAGE STATUS INFO 324
LONGER TERM STORAGE 114
DATA VOLUMES 310
STORED PAGES AT PHYSICAL LOCATIONS 312
LOG VOLUMES 317

FIG. 3

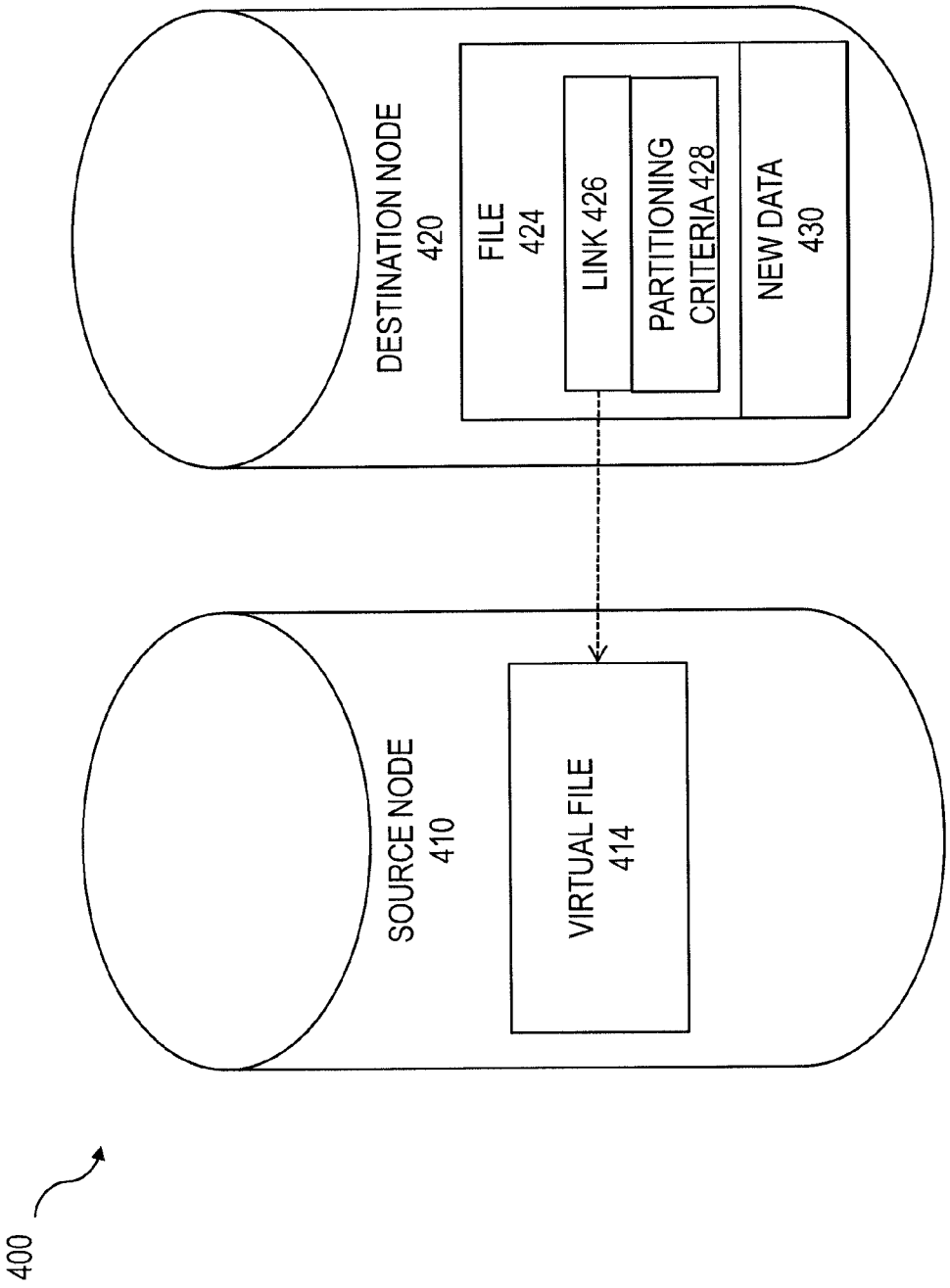

400
SOURCE NODE
410
VIRTUAL FILE
414
DESTINATION NODE
420
FILE
424
LINK 426
PARTITIONING CRITERIA 428
NEW DATA
430

FIG. 4

PARTITIONING DATA WITHIN A DISTRIBUTED DATA STORAGE SYSTEM USING VIRTUAL FILE LINKS

RELATED APPLICATION

This application is a continuation-in-part of U.S. patent application Ser. No. 13/290,835 filed on Nov. 7, 2011, the contents of which are hereby fully incorporated by reference.

TECHNICAL FIELD

The subject matter described herein relates to techniques for partitioning data and enabling access to partitioned data within a distributed data storage system using virtual file links.

BACKGROUND

In a distributed data storage system, data containers sometimes become so big that they must be partitioned/split over several nodes due to memory limitations and/or performance issues. Partitioning a data container means splitting a data container into several parts as well as combining smaller data containers into a single data container and moving associated data to another node. With some conventional database systems, this requires not only moving the data but also moving one or more logs tracking changes to such data container. These operations can be processor intensive and can also consume additional storage.

SUMMARY

In one aspect, a record is generated within a destination virtual file on a destination node of a distributed data storage system. The record comprises (i) a link directed to a source virtual file stored on a source node and (ii) partition criteria characterizing a partition of the source virtual file. The source virtual file is mapped to a chain of linked pages stored in a page buffer of the distributed data storage system. The partitioning criteria is used by at least one of the source node and the destination node to identify data associated with the partition. Thereafter, a request is received at the destination node to access data defined by the destination virtual file. In response to the request, data is provided from partition of the source virtual file stored on the source node using the link and the partitioning criteria.

Data generated at the node subsequent to the generation of the record can be appended to the record. The appended data can form a delta log at the destination node. The destination virtual file and the delta log can be overwritten during a columnar table merge operation with a new version of the destination virtual file comprising data from the partition of the source virtual file and the delta log. The link in the record can be overwritten during the columnar table merge operation. The new version of the destination virtual file can be persisted to secondary data storage. The new version of the destination virtual file can be persisted after a savepoint on the destination node. The partition of the source virtual file can be dropper after the new version of the destination virtual file is persisted to the secondary data storage. Dropping the partition of the source virtual file can include ceasing log operations relative to a portion of the source virtual file corresponding to the partition and/or deleting a portion of the source virtual file corresponding to the partition.

Articles of manufacture are also described that comprise computer executable instructions permanently stored on non-transitory computer readable media, which, when executed by a computer, causes the computer to perform operations herein. Similarly, computer systems are also described that may include a processor and a memory coupled to the processor. The memory may temporarily or permanently store one or more programs that cause the processor to perform one or more of the operations described herein. In addition, operations specified by methods can be implemented by one or more data processors either within a single computing system or distributed among two or more computing systems.

The subject matter described herein provides many advantages. For example, the current techniques allow for the partitioning of data among nodes within a distributed data storage system (i.e., a database system comprising a plurality of nodes, etc.) with little performance impact. Using links to virtual files as described herein obviates the need, during recovery from a log backup, for moving a portion of a virtual file from one node to the other which in turn requires writing a redo log on a destination node for all moved data or explicit expensive synchronization of recovery on several nodes. Stated differently, the use of a link to a virtual file requires a single operation (reading the linked virtual file) as opposed to physically moving data which requires at least three operations (writing data from the virtual file to the destination node, writing a log from the virtual file to the destination node, and reading data written to the destination node).

The details of one or more variations of the subject matter described herein are set forth in the accompanying drawings and the description below. Other features and advantages of the subject matter described herein will be apparent from the description and drawings, and from the claims.

DESCRIPTION OF DRAWINGS

FIG. **1** is a diagram illustrating a system including a data storage application;

FIG. **2** is a process flow diagram illustrating partitioning data within a distributed data storage system using virtual file links;

FIG. **3** is a diagram illustrating details of the system of FIG. **1**; and

FIG. **4** is a diagram illustrating partitioning of data on a source node to a destination node within a distributed data storage system.

Like reference symbols in the various drawings indicate like elements.

DETAILED DESCRIPTION

FIG. **1** shows an example of a system **100** in which a computing system **102**, which can include one or more programmable processors that can be collocated, linked over one or more networks, etc., executes one or more modules, software components, or the like of a data storage application **104**. The data storage application **104** can include one or more of a database, an enterprise resource program, a distributed storage system (e.g. NETAPP Filer available from NETAPP of Sunnyvale, Calif.), or the like.

The one or more modules, software components, or the like can be accessible to local users of the computing system **102** as well as to remote users accessing the computing system **102** from one or more client machines **106** over a network connection **110**. One or more user interface screens

produced by the one or more first modules can be displayed to a user, either via a local display or via a display associated with one of the client machines **106**. Data units of the data storage application **104** can be transiently stored in a persistence layer **112** (e.g. a page buffer or other type of temporary persistency layer), which can write the data, in the form of storage pages, to one or more storages **114**, for example via an input/output component **116**. The one or more storages **114** can include one or more physical storage media or devices (e.g. hard disk drives, persistent flash memory, random access memory, optical media, magnetic media, and the like) configured for writing data for longer term storage. It should be noted that the storage **114** and the input/output component **116** can be included in the computing system **102** despite their being shown as external to the computing system **102** in FIG. **1**.

Data retained at the longer term storage **114** can be organized in pages, each of which has allocated to it a defined amount of storage space. In some implementations, the amount of storage space allocated to each page can be constant and fixed. However, other implementations in which the amount of storage space allocated to each page can vary are also within the scope of the current subject matter.

FIG. **2** is a process flow diagram **200** in which, at **210**, a record is generated on a destination node of a distributed data storage system. The record can include (i) a link directed to a source virtual file stored on a source node and (ii) partition criteria characterizing a partition of the source virtual file. The source virtual file can be mapped to a chain of linked pages stored in a page buffer of the distributed data storage system, the partitioning criteria being used by at least one of the source node and the destination node to identify data associated with the partition. Subsequently, at **220**, a request is received at the destination node to access data defined by the destination virtual file. In response to the request, at **230**, data is provided from the partition of the source virtual file stored on the source node using the link and the partitioning criteria.

FIG. **3** shows a software architecture **300** consistent with one or more features of the current subject matter. A data storage application **104**, which can be implemented in one or more of hardware and software, can include one or more of a database application, a network-attached storage system, or the like. According to at least some implementations of the current subject matter, such a data storage application **104** can include or otherwise interface with a persistence layer **112** or other type of memory buffer, for example via a persistence interface **302**. A page buffer **304** within the persistence layer **112** can store one or more logical pages **306**, and optionally can include shadow pages **311**, active pages **313**, data pages of virtual files **315** and the like. The logical pages **306** retained in the persistence layer **112** can be written to a storage (e.g. a longer term storage, etc.) **114** via an input/output component **116**, which can be a software module, a sub-system implemented in one or more of software and hardware, or the like. The storage **114** can include one or more data volumes **310** where stored pages **312** are allocated at physical memory blocks.

In some implementations, the data storage application **104** can include a row store **303** and a column store **305**. The row store **303** can comprise or be otherwise in communication with a page manager **314** and/or a savepoint manager **316**. The page manager **314** can communicate with a page management module **320** at the persistence layer **112** that can include a free block manager **322** that monitors page status information **324**, for example the status of physical pages within the storage **114** and logical pages in the persistence layer **112** (and optionally in the page buffer **304**). The savepoint manager **316** can communicate with a savepoint coordinator **326** at the persistence layer **204** to handle savepoints, which are used to create a consistent persistent state of the database for restart after a possible crash. The row store **303** can access the persistence interface **302** via an absolute page API **307**. The column store **305** which can store columns in contiguous memory can access the persistence interface **302** via a virtual file API **309**.

In some implementations of a data storage application **104**, the page management module of the persistence layer **112** can implement shadow paging. The free block manager **322** within the page management module **320** can maintain the status of physical pages. The page buffer **304** can included a fixed page status buffer that operates as discussed herein. A converter component **340**, which can be part of or in communication with the page management module **320**, can be responsible for mapping between logical and physical pages written to the storage **114**. The converter **340** can maintain the current mapping of logical pages to the corresponding physical pages in a converter table **342**. The converter **340** can maintain a current mapping of logical pages **306** to the corresponding physical pages in one or more converter tables **342**. When a logical page **306** is read from storage **114**, the storage page to be loaded can be looked up from the one or more converter tables **342** using the converter **340**. When a logical page is written to storage **114** the first time after a savepoint, a new free physical page is assigned to the logical page. The free block manager **322** marks the new physical page as "used" and the new mapping is stored in the one or more converter tables **342**.

The persistence layer **112** can ensure that changes made in the data storage application **104** are durable and that the data storage application **104** can be restored to a most recent committed state after a restart. Writing data to the storage **114** need not be synchronized with the end of the writing transaction. As such, uncommitted changes can be written to disk and committed changes may not yet be written to disk when a writing transaction is finished. After a system crash, changes made by transactions that were not finished can be rolled back. Changes occurring by already committed transactions should not be lost in this process. A logger component **344** can also be included to store the changes made to the data of the data storage application in a linear log. The logger component **344** can be used during recovery to replay operations since a last savepoint to ensure that all operations are applied to the data and that transactions with a logged "commit" record are committed before rolling back still-open transactions at the end of a recovery process.

With some data storage applications, writing data to a disk is not necessarily synchronized with the end of the writing transaction. Situations can occur in which uncommitted changes are written to disk and while, at the same time, committed changes are not yet written to disk when the writing transaction is finished. After a system crash, changes made by transactions that were not finished must be rolled back and changes by committed transaction must not be lost.

To ensure that committed changes are not lost, redo log information can be written by the logger component **344** whenever a change is made. This information can be written to disk at latest when the transaction ends. The log entries can be persisted in separate log volumes **317** while normal data is written to data volumes **310**. With a redo log, committed changes can be restored even if the corresponding data pages were not written to disk. For undoing

uncommitted changes, the persistence layer **112** can use a combination of undo log entries (from one or more logs) and shadow paging.

The persistence interface **302** can handle read and write requests of stores (e.g., in-memory stores, etc.). The persistence interface **302** can also provide write methods for writing data both with logging and without logging. If the logged write operations are used, the persistence interface **302** invokes the logger **344**. In addition, the logger **344** provides an interface that allows stores (e.g., in-memory stores, etc.) to directly add log entries into a log queue. The logger interface also provides methods to request that log entries in the in-memory log queue are flushed to disk.

Log entries contain a log sequence number, the type of the log entry and the identifier of the transaction. Depending on the operation type additional information is logged by the logger **344**. For an entry of type "update", for example, this would be the identification of the affected record and the after image of the modified data.

When the data application **104** is restarted, the log entries need to be processed. To speed up this process the redo log is not always processed from the beginning. Instead, as stated above, savepoints can be periodically performed that write all changes to disk that were made (e.g., in memory, etc.) since the last savepoint. When starting up the system, only the logs created after the last savepoint need to be processed. After the next backup operation the old log entries before the savepoint position can be removed.

When the logger **344** is invoked for writing log entries, it does not immediately write to disk. Instead it can put the log entries into a log queue in memory. The entries in the log queue can be written to disk at the latest when the corresponding transaction is finished (committed or aborted). To guarantee that the committed changes are not lost, the commit operation is not successfully finished before the corresponding log entries are flushed to disk. Writing log queue entries to disk can also be triggered by other events, for example when log queue pages are full or when a savepoint is performed.

The column store **305** can persist its tables to virtual files provided by the persistence layer **112** via the virtual file API **307**. Internally the persistence layer **112** can map a virtual file to a chain of linked pages **315** stored in the page buffer **304**. Data belonging to one columnar table can be stored in multiple virtual files: one virtual file per column for a main storage and one virtual file for a delta log. In addition, one virtual file can optionally be stored per column for the main storage of the history part of the table, and/or one virtual file can optionally be stored per table for the delta of the history part of the table. The persistence layer **112** can maintain a directory that stores for each virtual file the start page and additional information such as the size and the type of the virtual file.

As stated above, virtual files can be used to store main and delta parts of columnar tables. These files can be read on the first access of the corresponding table into memory. With some implementations, while read accesses happen only on the in-memory representation of data, updates, appends, overwrites and truncates can also be written to the virtual file on disk. After partitioning/moving of a virtual file from a source node to a destination node, the virtual file can be read into memory on first access on the destination node. To support recovery from log backup, moving a virtual file from one node to the other (if the techniques described below are not incorporated) can either require writing a redo log on the destination node for all partitioned/moved data or explicit expensive synchronization of recovery on several nodes, which is in both cases too big performance penalty.

The content of a main storage can only change when a delta merge operation is performed. Therefore the main virtual files can only be written when a merge is done. Note that this does not mean that main data is written to disk during a merge operation: when the column store **305** writes to a virtual file, the data can be written into the page buffer **304** of the persistence layer **112**. It is the responsibility of the persistence layer **112** to determine when the data in the virtual file is actually flushed to disk (e.g., during page replacement or at latest when the next savepoint is written, etc.).

A delta merge operation is unique to the column store **305** and is not synchronized with the savepoints of the persistence layer **112**. Delta merge is primarily an optimization of in-memory structures performed on the granularity of a single table. The savepoint, on the other hand, works on the whole database and its purpose is to persist changes to disk.

All changes executed on column store **305** data go into delta storages in the data volumes **310**. The delta storages can exist only in memory as opposed to be written to disk. However, the column store **305** can, via the logger **344**, write a persisted delta log that contains logical redo log entries for all operations executed on the delta storages. Logical log, in this context, means that the operation and its parameters are logged but no physical images are stored. When a delta merge operation is executed, the changes in the delta storage can be merged into the main storage and the delta log virtual file can be truncated.

Despite of the name "delta log", the delta log virtual files are not really logs from the persistence layer **112** point of view. For the persistence layer **112** they are just data. The actual redo log and undo entries can be written a log volume **317** in the persistence layer **112**. The virtual files used for delta logs can be configured as logged. Whenever column store **305** writes to the delta log virtual file, the persistence layer interface **302** invokes the logger **344** and an undo manager to write redo log entries and undo information. This ensures that the delta log virtual files can be restored after a restart—just like any other data. After the delta log virtual files are restored they are ready to be processed by column store **305** to rebuild the in-memory delta storages from the logical delta log entries.

During a delta merge operation the main files for the affected table(s) can be rewritten and the delta log file can be truncated. For all these operations no log is written by the persistence layer **112**. This is possible, because all operations executed on the tables were already logged when the delta files were written as part of the original change operation. The merge operation does not change, create or delete any information in the database. It is just a reorganization of the way existing information is stored. To prevent that logs are written for a merge operation, the virtual main files are configured as not logged and a special not logged operation is used for delta log truncation.

During restart, the persistence layer **112** can restore the main virtual files from the last savepoint. The delta log virtual files can be restored from the last savepoint and from the redo log. When the persistence layer **112** has finished its part, the main storage of the columns can be loaded from the virtual files into column-store memory. This involves memory copy operations between data cache in the page buffer **304** of the persistence layer **112** and the contiguous memory areas in column store **305**. The column store **305** can then execute the logical redo entries from delta log virtual files and rebuild the in-memory delta storages.

As mentioned above, there is metadata that allows to define for each columnar table whether it is to be loaded during system startup. If a table is configured for loading on demand, the restore sequence for that table is executed on first access.

In some situations, it can be necessary to partition data within a virtual file among two or more nodes. With reference to the diagram **400** of FIG. **4**, a virtual file **414** on a source node **410** within a distributed data storage system can be partitioned by creating a new file **424** (e.g., an empty object, etc.) on a destination node **420** containing a special record **426** with (i) a link to the original virtual file **414** on the source node **410**; and (ii) partitioning criteria **428** that characterizes a corresponding partition on the original virtual file **414**. This operation can be logged by the logger **344**. The original virtual file **414** can be kept unchanged on the source node. When the virtual file is read on the destination node, the link **426** is encountered which with the partitioning criteria **428**, in turn, results in the data from the partition of the original virtual file **410** being accessed on the source node **420**. New data **430** can be appended after the link record **426** on the destination node **420**, thus always all data of the file can be read. As used in this context, nodes can refer to servers having their own persistency.

When performing columnar table merge operation, virtual files containing compressed data of a corresponding table can be overwritten completely with the new version and the virtual file for deltas will be truncated. These operations can automatically overwrite the link **426** to the source node **410**. During a link cleanup operation after commit, the original files can be scheduled for removal. After a savepoint on the destination node **420** has persisted the new versions of the files to secondary storage, old files on the source node **410** can be dropped (non-logged). At this time, files can be completely moved to the destination node **420**, without unnecessary performance penalty by logging whole contents on the destination node **420**.

During recovery from log, link record **426** can be recovered as well as all new data **430** appended to the file. The portion of the old virtual file **414** corresponding to the partition on the source node **410** can be kept because dropping of the virtual file **414** on the source node **410** need not be logged. If the portion of the virtual file **414** corresponding to the partition is deleted during recovery, the deletion can schedule dropping the portion of the original file **414** corresponding to the partition on the source node **410** after recovery ends. Otherwise, there would be a file **424** with the link **426** to old data on the source node **420** corresponding to the partition (or even a chain of links, if the virtual file **414** has been moved several times).

The subject matter described above can be extended to enable a virtual file to be repartitioned from n partitions to m partitions and/or to join n partitions to a single partition. With such variations, instead of writing a single link on each destination node **420**, several links can be written on the destination node **420** which refer to all original n partitions where to read the source data (which can be from a plurality of different nodes). Such an arrangement can be further optimized if a new partition of the source node **410** only a subset of the data of some of the original partitions (e.g., previous partition criteria was partition key module **2** while the new criteria is partition key modulo **4**, etc.).

Aspects of the subject matter described herein can be embodied in systems, apparatus, methods, and/or articles depending on the desired configuration. In particular, various implementations of the subject matter described herein can be realized in digital electronic circuitry, integrated circuitry, specially designed application specific integrated circuits (ASICs), computer hardware, firmware, software, and/or combinations thereof. These various implementations can include implementation in one or more computer programs that are executable and/or interpretable on a programmable system including at least one programmable processor, which can be special or general purpose, coupled to receive data and instructions from, and to transmit data and instructions to, a storage system, at least one input device, and at least one output device.

These computer programs, which can also be referred to programs, software, software applications, applications, components, or code, include machine instructions for a programmable processor, and can be implemented in a high-level procedural and/or object-oriented programming language, and/or in assembly/machine language. As used herein, the term "machine-readable medium" refers to any computer program product, apparatus and/or device, such as for example magnetic discs, optical disks, memory, and Programmable Logic Devices (PLDs), used to provide machine instructions and/or data to a programmable processor, including a machine-readable medium that receives machine instructions as a machine-readable signal. The term "machine-readable signal" refers to any signal used to provide machine instructions and/or data to a programmable processor. The machine-readable medium can store such machine instructions non-transitorily, such as for example as would a non-transient solid state memory or a magnetic hard drive or any equivalent storage medium. The machine-readable medium can alternatively or additionally store such machine instructions in a transient manner, such as for example as would a processor cache or other random access memory associated with one or more physical processor cores.

The subject matter described herein can be implemented in a computing system that includes a back-end component, such as for example one or more data servers, or that includes a middleware component, such as for example one or more application servers, or that includes a front-end component, such as for example one or more client computers having a graphical user interface or a Web browser through which a user can interact with an implementation of the subject matter described herein, or any combination of such back-end, middleware, or front-end components. A client and server are generally, but not exclusively, remote from each other and typically interact through a communication network, although the components of the system can be interconnected by any form or medium of digital data communication. Examples of communication networks include, but are not limited to, a local area network ("LAN"), a wide area network ("WAN"), and the Internet. The relationship of client and server arises by virtue of computer programs running on the respective computers and having a client-server relationship to each other.

The implementations set forth in the foregoing description do not represent all implementations consistent with the subject matter described herein. Instead, they are merely some examples consistent with aspects related to the described subject matter. Although a few variations have been described in detail herein, other modifications or additions are possible. In particular, further features and/or variations can be provided in addition to those set forth herein. For example, the implementations described above can be directed to various combinations and sub-combinations of the disclosed features and/or combinations and sub-combinations of one or more features further to those disclosed herein. In addition, the logic flows depicted in the

accompanying figures and/or described herein do not necessarily require the particular order shown, or sequential order, to achieve desirable results. The scope of the following claims may include other implementations or embodiments.

What is claimed is:

1. A non-transitory computer program product storing instructions that, when executed by at least one programmable processor forming part of at least one computing system, cause the at least one programmable processor to perform operations comprising:

partitioning data in a file, stored on a source node of a distributed data storage system comprising a plurality of nodes including the source node and a destination node, into a source virtual file corresponding to one or more source partitions on the source node and a destination virtual file corresponding to one or more destination partitions on the destination node;

generating, on the destination node, a destination record within the destination virtual file, the destination record comprising (i) a link directed to the source virtual file stored on the source node and (ii) partition criteria characterizing the one or more source partitions corresponding to the source virtual file, the source virtual file being mapped to a chain of linked pages stored in a page buffer of the distributed data storage system, the page buffer being a temporary persistency layer;

appending, subsequent to the generation of the destination record, data generated at the destination node to the destination record, the appended data forming a delta log at the destination node;

receiving, at the destination node from a requesting device, a request to access data defined by the destination virtual file;

determining, at the destination node using the destination virtual file, that the data, requested by the requesting device, is stored in the one or more source partitions corresponding to the source virtual file;

providing the data, requested by the requesting device, from the one or more source partitions corresponding to the source virtual file stored on the source node to the requesting device through the destination node using the link and the partition criteria;

initiating a columnar table merge operation;

overwriting, as part of the columnar table merge operation, (i) the link, in the destination record, directed to the source virtual file stored on the source node, and (ii) the destination virtual file and the delta log with a new version of the destination virtual file comprising data from the one or more source partitions corresponding to the source virtual file and the delta log;

persisting, as part of the columnar table merge operation, the destination virtual file to a secondary storage; and

dropping, from a logging schedule and as part of the columnar table merge operation, the source virtual file stored on the source node.

2. The computer program product as in claim **1**, wherein the new version of the destination virtual file is persisted after a savepoint on the destination node.

3. The computer program product as in claim **2**, wherein dropping the source virtual file comprises ceasing log operations relative to a portion of the source virtual file corresponding to the one or more source partitions subject to the columnar table merge operation.

4. The computer program product as in claim **3**, wherein dropping the source virtual file comprises deleting the portion of the source virtual file corresponding to the one or more source partitions subject to the columnar table merge operation.

5. The computer program product as in claim **1**, wherein the operations further comprise:

partitioning the source virtual file into a different number of partitions; and

writing, in the destination record, links to each of the different number of partitions.

6. The computer program product as in claim **5**, wherein the different number of partitions refer to source data on the source node and on a node other than the source node and the destination node.

7. The computer program product as in claim **1**, wherein each virtual file is flushed to physical storage during page replacement or when a next savepoint is written.

8. The computer program product as in claim **1**, further comprising a partition map, wherein the partition map includes a mapping of the one or more partitions corresponding with the source virtual file stored on the source node to the destination node.

9. A method comprising:

partitioning data in a file stored in a source node, of a distributed data storage system comprising a plurality of nodes including a source node and a destination node, into a source virtual file corresponding to one or more source partitions on the source node and a destination virtual file corresponding to one or more destination partitions on the destination node;

generating, on the destination node, a destination record within the destination virtual file, the destination record comprising (i) a link directed to the source virtual file stored on the source node and (ii) partition criteria characterizing the one or more source partitions that correspond with the source virtual file, the source virtual file being mapped to a chain of linked pages stored in a page buffer of the distributed data storage system, the page buffer being a temporary persistency layer;

appending, subsequent to the generation of the destination record, data generated at the destination node to the destination record, the appended data forming a delta log at the destination node;

receiving, at the destination node from a requesting device, a request to access data defined by the destination virtual file;

determining, at the destination node using the destination virtual file, that the data, requested by the requesting device, is stored in the one or more source partitions corresponding to the source virtual file;

providing the data, requested by the requesting device, from the one or more source partitions corresponding to the source virtual file stored on the source node to the requesting device through the destination node using the link and the partitioning criteria;

initiating a columnar table merge operation;

overwriting, as part of the columnar table merge operation, (i) the link, in the destination record, directed to the source virtual file stored on the source node, and (ii) the destination virtual file and the delta log with a new version of the destination virtual file comprising data from the one or more source partitions corresponding to the source virtual file and the delta log;

persisting, as part of the columnar table merge operation, the destination virtual file to a secondary storage; and,

dropping, from a logging schedule and as part of the columnar table merge operation, the source virtual file stored on the source node.

10. The method as in claim **9**, wherein the new version of the destination virtual file is persisted after a savepoint on the destination node.

11. The method as in claim **10**, wherein dropping the source virtual file comprises ceasing log operations relative to a portion of the source virtual file corresponding to the one or more source partitions subject to the columnar table merge operation.

12. The method as in claim **11**, wherein dropping the source virtual file comprises deleting the portion of the source virtual file corresponding to the one or more source partitions subject to the columnar table merge.

13. The method as in claim **9**, wherein each virtual file is flushed to physical storage during page replacement or when a next savepoint is written.

14. A system comprising:

at least one data processor; and

memory coupled to the at least one data processor, the memory storing instructions, which when executed, cause the at least one data processor to perform operations comprising:

partitioning data in a file stored on a source node, of a distributed data storage system comprising a plurality of nodes including a source node and a destination node, into a source virtual file corresponding to one or more source portions on the source node and a destination virtual file corresponding to one or more destination portions on the destination node;

generating, on the destination node, a destination record within the destination virtual file, the destination record comprising (i) a link directed to the source virtual file stored on at least two source nodes and (ii) partition criteria characterizing the one or more source partitions corresponding with the source virtual file on one of the source nodes, the source virtual file being mapped to a chain of linked pages stored in a page buffer of the distributed data storage system, the page buffer being a temporary persistency layer;

appending, subsequent to the generation of the destination record, data generated at the destination node to the destination record, the appended data forming a delta log at the destination node;

receiving, at the destination node from a requesting device, a request to access data defined by the destination virtual file;

determining, at the destination node using the destination virtual file, that the data, requested by the requesting device, is stored in the one or more source partitions corresponding to the source virtual file;

providing the data, requested by the requesting device, from the one or more source partitions corresponding to the source virtual file stored on the corresponding source node to the requesting device through the destination node using the link and the partitioning criteria;

initiating a columnar table merge operation;

overwriting, as part of the columnar table merge operation, (i) the link, in the destination record, directed to the source virtual file stored on the source node, and (ii) the destination virtual file and truncating the delta log with a new version of the destination virtual file comprising data from the one or more source partitions corresponding with the source virtual file and the delta log;

persisting, as part of the columnar table merge operation, the destination virtual file to a secondary storage; and,

dropping, from a logging schedule and as part of the columnar table merge operation, the source virtual file stored on the source node.

15. The system as in claim **14**, wherein each virtual file is flushed to physical storage during page replacement or when a next savepoint is written.

* * * * *