(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
16 September 2010 (16.09.2010)

PCT

(10) International Publication Number
WO 2010/104902 A2

(51) International Patent Classification:
G06F 17/30 (2006.01)

(21) International Application Number:
PCT/US2010/026746

(22) International Filing Date:
10 March 2010 (10.03.2010)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
12/402,024 11 March 2009 (11.03.2009) US

(71) Applicant *(for all designated States except US)*: **ORACLE INTERNATIONAL CORPORATION** [US/US]; 500 Oracle Parkway, Redwood Shores, California 94065 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **ORELAND, Jonas** [SE/SE]; 500 Oracle Parkway, S-94065 Stockholm (SE). **CLEMENT, Frazer** [GB/GB]; 500 Oracle Parkway, Redwood Shores, Surrey 94065 (GB). **ULIN, Tomas** [SE/SE]; 500 Oracle Parkway, S-94065 Stockholm (SE).

(74) Agent: **LEMBKE, Kent, A.**; MARSH FISCHMANN & BREYFOGLE LLP, 8055 E. Tufts Avenue, Suite 450, Denver, Colorado 80237 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, 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, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

— *without international search report and to be republished upon receipt of that report (Rule 48.2(g))*

(54) **Title**: COMPOSITE HASH AND LIST PARTITIONING OF DATABASE TABLES

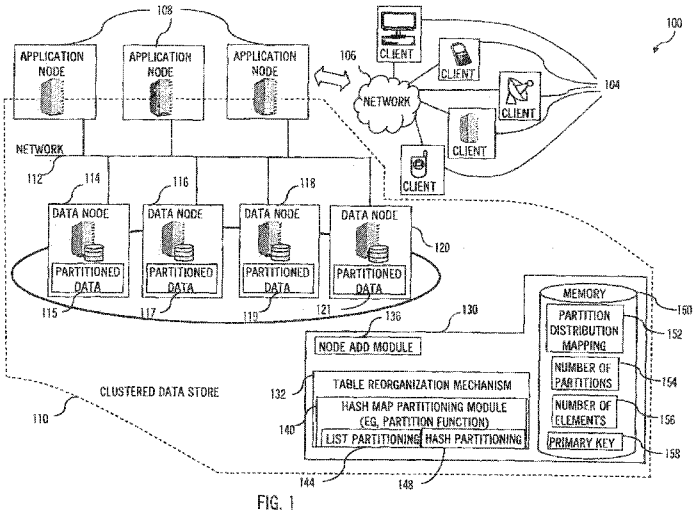

FIG. 1

(57) **Abstract**: A method for partitioning during an online node add. The method includes providing a data storage cluster (110) with first and second nodes (114, 116), and storing a table (204) of data in the data storage cluster (110) with a first partition (222) storing a set of rows or data elements in the first node and a second partition (320) storing a set of rows or data elements in the second node. The method includes adding a third node (118) to the cluster and adding a third partition to the table using a partitioning mechanism (140, 234) to create a distribution mapping (152) for data elements in the first, second, and third partitions. The distribution mapping (152) provides substantially uniform distribution of the data elements over the first, second, and third partitions by the partitioning mechanism (140, 234) using modulo hash partitioning as a function of data elements or by combining hash and list partitioning (144, 148) such that data is retained on the original partitions.

WO 2010/104902 A2

COMPOSITE HASH AND LIST PARTITIONING OF DATABASE TABLES

BACKGROUND OF THE INVENTION

1. Field of the Invention.

[0001] The present invention relates, in general, to methods and systems for managing data storage in tables and databases, and, more particularly, to methods and systems for providing improved partitioning of a table of data to support adding a node to a data storage cluster.

2. Relevant Background.

[0002] In the data storage or information technology industry, relational database management systems (RDMS) provide important support for a wide variety of commercial applications, and there is growing demand for methods and devices for effectively and efficiently storing large quantities of data in a manner that allows it to be quickly retrieved and reliably stored. In databases, information is typically stored in rows of data fields storing pieces of information (e.g., one field may store a person's name, another field may store the person's address, and so on within a row) with one or more of the fields providing a key (e.g., a primary key may be included that uniquely identifies each row of data in a database or table). For example, clustered, high-availability databases are required by telecommunication companies and many other service providers such as financial institutions, Web-based retailers and service providers, and the like. The rate of increasing size of databases causes many challenges within the data storage industry including how to add additional storage devices (such as disk drives, tape drives, optical drives, servers, and the like) to provide more nodes for storing larger and larger tables, e.g., to handle rapidly increasing volumes of data stored in rows of a table may require modification of a database cluster of nodes to handle the added quantity of information.

[0003] Every RDMS developer eventually encounters a situation in which a table stores a huge amount of historical data, but users typically only retrieve small, distinct portions at any particular time. For example, a financial institution may track millions of records related to trades of stocks spanning years, but a user may only need to retrieve trade data for a small time

period such as a particular month. To improve query performance as well as storing growing volumes of data in a table, a DBMS developer often splits a large table into separate tables with the same structure (e.g., same fields/columns). A table typically is partitioned horizontally with each member or separate table having the same number of columns/fields as the original table, and each column has the same attributes (such as data type, size, and the like) as the corresponding column in the original table. An ongoing challenge for the DBMS developer is how to partition tables of data in a manner that provides a better utilization of hardware and allows for quick reproduction and/or reorganization of data.

[0004] Particularly, it would be desirable to provide mechanisms for deciding how to partition tables in relational database management systems implementing shared-nothing architectures to provide database management. With the shared-nothing approach, each processor has its own memory as well as local disks or data storages. Except for the communication network, no other resources are shared among the processors. For example, MySQL's NDB Cluster storage engine is a distributed, in-memory, shared-nothing storage engine with synchronous replication to other sites and automatic horizontal data partitioning across the nodes that store the distributed data (e.g., buckets or partitions of a table). With this storage engine, any given row of data of a table is eligible to be stored in any of the partitions on the clustered nodes, and, typically, RDMS uses synchronous replication. The table's definition specifies which rows map to which partitions based on a partitioning function, and, as a result, it is important to choose or design the partitioning function to achieve effective reorganization of a table of data such when a node is being added to a cluster. Partitioning is a significant design problem for other storage engine products as inefficient partitioning can quickly result in undesirable distributions of data (e.g., with some nodes storing larger portions of a table) and inefficient use of memory during reorganization processes.

[0005] There remains a need for a table reorganization mechanism used by the storage engine for adding partitions to a table that remain online and available. Design requirements for such a table reorganization mechanism may be that online reads, updates, and scans (i.e., transactions) should not be blocked. Additionally, it may be desirable for the reorganization to be completed without requiring duplication of an entire table, but, instead, only the rows that are moved to a new partition (e.g., in an added node or in an existing node) need to exist in two places in memory. For example, when the table reorganization module is used in

combination with an add node operation, this means that no extra memory is required on the original older nodes.

[0006] Many storage engines support a number of differing mechanisms for partitioning tables including by data ranges (e.g., partition a table into 12 partitions coinciding with the months of the year), by use of linear hashing, by modulo hashing, and the like. Linear hashing only needs to split a bucket or partition when adding a new bucket or partition, but linear hashing introduces a skew in data distribution among the data buckets or partitions. An advantage in partitioning by linear hashing is that the adding, dropping, merging, and splitting of partitions is made much faster, which can be beneficial when dealing with tables containing extremely large amounts of data. But, as mentioned above, a disadvantage is that data is less likely to be evenly distributed between partitions as compared with the distribution using modulo hashing in partitioning. Modulo hashing has no skew of data, but, unfortunately, all rows of a table need to be moved when the number of buckets or partitions is changed.

[0007] To understand partitioning using linear hashing and modulo hashing, it may be useful to consider a simple example. Consider the following data (`id`, `name`), where the primary key is `id` that may make up the rows of a table with two fields or columns: (1,"Tomas"); (2,"Kent"); (3,"Jonas"); (4,"Frazer"); (5,"John"); (6,"Eliza"); (7,""); (8,""); (9,""); (10,""); (11,""); and (12,""). When this set of data is stored by a storage engine (such as within a MySQL Cluster with the ndb storage engine), the data may be separated onto different nodes. For example, the storage engine may use a modulo hashing distribution for storing the data in two buckets on two nodes, and the data may look like: Node 1 - (1,"Tomas"); (3,"Jonas"); (5,"John"); (7,""); (9,""); and (11,"") and Node 2 - (2,"Kent"); (4,"Frazer"); (6,"Eliza"); (8,""); (10,""); and (12,""). For a three node configuration, the data may be distributed as: Node 1 - (1,"Tomas"); (4,"Frazer"); (7,""); and (10,""); Node 2 - (2,"Kent"); (5,"John"); (8,""); and (11,""); and Node 3 - (3,"Jonas"); (6,"Eliza"); (9,""); and (12,""). This example has been simplified, and, in practice, modulo hashing involved taking the modulo of the hash of the primary key (rather that of the primary key itself as it may appear here). It can be seen that the data is evenly distributed in both cases with each partition having 6 rows in the first example and each partition having 4 rows of data in the second example. However, the data resides very differently on three nodes compared to two nodes with only four data entries being on the same node in the 2 cases (i.e., (1,"Tomas"); (2,"Kent"); (7,""); and (8,"") are on the same nodes

in each partitioning implementation). Even distribution is desirable but reshuffling of data or all rows when adding nodes or partitions can make online addition of nodes or reorganization of partitioned tables painful, e.g., if 1 gigabyte of data is stored in a partitioned table, adding a node using modulo hashing may require 1 gigabyte of data to be moved.

[0008] Alternatively, the storage engine may use a linear hash partitioning for storing this same data in two buckets on two nodes, and the data may look like: Node 1 - (1,"Tomas"); (3,"Jonas"); (5,"John"); (7,""); (9,""); and (11,"") and Node 2 - (2,"Kent"); (4,"Frazer"); (6,"Eliza"); (8,""); (10,""); and (12,""). For three nodes, the partitions or buckets may be: Node 1 - (1,"Tomas"); (5,"John"); and (9,""); Node 2 - (2,"Kent"); (6,"Eliza"); and (10,""); and Node 3 - (3,"Jonas"); (4,"Frazer"); (7,""); (8,""); (11,""); and (12,""). In another implementation of linear hashing, one of the original nodes may have the same entries as it had before or originally, while the other original node may have half of its original entries with the other half being on the new node. The 3-node partitioning example shows the desirable property that Nodes 1 and 2 do not have any new data as compared to the 2-node configuration. It is, however, evident that the distribution of data is very uneven and skewed when we have 3 nodes. This is apparent with Node 3 having twice as many data entries or rows as Node 1 and Node 2.

[0009] Since conventional methods of partitioning including use of linear hashing and modulo hashing have significant drawbacks, there remains a need for a new partitioning mechanism or module for use by storage engines in shared-nothing and other RDMS data storage architectures or systems.

SUMMARY OF THE INVENTION

[0010] Briefly, a partitioning method is provided to address at least some of the above problems by combining the preferred aspects of modulo hash partitioning with that of linear hashing. The new technique may be termed HashMap partitioning. Linear hashing as may be used in partitioning tables with database storage engines (such as MySQL NDB Cluster storage engine) allows for minimal copying during a table reorganization (such as to add a node to a clustered data store). However, linear hashing does not give an even data distribution unless the new number of nodes/buckets is some multiple of two times the

original number. Use of modulo hashing distributions provides even distribution over the nodes or partitions on such nodes, but it often requires a great deal of data shuffling as the data on the original nodes is not retained. The HashMap partitioning will generate a distribution mapping provides a combining of hash and list partitioning to provide the best of both the modulo hashing and linear hashing techniques, for a new partitioning that facilitates performing an online altering or modifying of a partitioning of a table. In addition, HashMap partitioning will use minimal data copy and a minimal need for extra memory or space in the original data storage nodes/devices during the altering or reorganization process (e.g., during an online node add). All of this is done while maintaining an even distribution of data in the partitions of the storage nodes. For example, when a storage engine uses the HashMap partitioning tool or mechanism together with adding database nodes in a shared-nothing cluster like a MySQL cluster, no additional memory on the existing nodes is needed while the repartitioning of the table is performed to incorporate the new nodes and copy/move data from old to new partitions/nodes.

[0011] More particularly, a computer-based method is provided for reorganizing a database such as providing partitioning upon addition of a node to a data storage cluster. The method includes providing a data storage cluster with first and second nodes, and storing a table of data in the data storage cluster with a first partition storing a set of rows or data elements in the first node and a second partition storing a set of rows or data elements in the second node. The method also includes modifying the data storage cluster to include a third node for storing data from the table. A storage engine may be included in the storage cluster to manage the nodes and data storage, and the method may include adding a third partition to the table including using a partitioning mechanism to create a distribution mapping for data elements in the first, second, and third partitions. The method then includes copying a portion of the rows of the table from both the first and second nodes to the third partition of the third node based on the distribution mapping. Then, the copied rows are deleted from the first and second nodes. Prior to the deletion of copied rows, the method may include switching over distribution to use the first, second, and third partitions for data transactions according to the new mapping, and also waiting for scans using prior distribution of the first and second partitions to finish.

[0012] In the method, the distribution mapping does not require additional memory space for the first and second nodes as the copying is only from the first and second nodes (the old

nodes) and not to or onto the first and second nodes (e.g., these nodes retain data and only lose data that is moved to the new node/partition). In some embodiments, the distribution mapping provides substantially uniform distribution of the data elements (or rows) over the first, second, and third partitions. This may be achieved by having the partitioning mechanism use modulo hash partitioning as a function of the data elements (and not as a direct function of the number of partitions in the table). The partitioning mechanism may be thought of as combining hash and list partitioning such that uniform distribution is achieved but data is also retained on the original or old partitions rather than requiring significant data copying/shuffling.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] Fig. 1 illustrates a functional block diagram of a computer system or network including a clustered data store with a plurality of data nodes used to store partitioned data (e.g., databases or tables that are horizontally partitioned or the like) and a storage engine adapted for partitioning a table of data, e.g., as part of a node add process;

[0014] Fig. 2 illustrates a functional block or schematic diagram of computer system or network with a data or storage node used to store an application data set or table in a single partition but showing addition of a second node or node group to the storage cluster;

[0015] Fig. 3 illustrates the computer system or network of Fig. 2 after a distribution mapping has been generated by a partitioning tool run by the storage engine showing the copying of data to the newly added node (e.g., to create a second partition or bucket for the application data);

[0016] Fig. 4 illustrates the computer system or network of Figs. 2 and 3 after data copying with data distribution switched to reflect the use of two partitions;

[0017] Fig. 5 illustrates the computer system or network of Figs. 2-4 after previously copied rows or data entries have been deleted from the original older node; and

[0018] Fig. 6 is a flow chart of an exemplary table reorganization that may be performed using HashMap partitioning, e.g., as part of or concurrent with a node add process.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0019] Briefly, embodiments of the present invention are directed to methods and systems for providing enhanced partitioning of data, such as a database table, that is distributed over a number of nodes, and the partitioning techniques described are particularly useful when a data storage node is being added (e.g., the system is being expanded in size to support storage of growing amounts of data). For example, clustering is a way of scaling by distributing load over many servers, and clustered computer systems or networks are used to allow several to many hosts to appear as a single server of data. The following description provides specific clustering examples that use MySQL's NDB Cluster storage engine, which is a distributed, in-memory, shared-nothing storage engine with synchronous and automatic data partitioning across the nodes of the clustered data store or system, but it will be understood that the partitioning techniques described are useful in nearly any data storage system in which data is partitioned and it is desirable to alter the partitions over time. When a storage engine or similar component manages data that is distributed over different nodes, the methods and systems described herein provide a partitioning module or mechanism that allows the storage engine to decide how to reorganize the distributed data into useful partitions (e.g., to properly horizontally partition a table of data).

[0020] The partitioning problem arises because there is a need to be able to grow or expand a data storage system, and, if possible, to allow the growth to occur online. As discussed above, the use of modulo hashing to partition a table (or hash partitioning) is useful in that it provides an even distribution of data, but when one or more nodes are added to a clustered or other data store, all the data has to be reshuffled or moved to reflect a new partitioning, which may result in gigabytes of data having to be copied and reorganized. The partitioning method proposed here involves combining hash and list partitioning to provide a new HashMap partitioning that may be used online such as by a storage engine or other database management mechanism/tool to partition data more efficiently. The HashMap partitioning method (and associated software module/mechanism) provides benefits of both the modulo hashing technique in that data is evenly distributed and linear hashing techniques in that large amounts of data does not have to be copied or moved. As will become clear, HashMap partitioning provides better utilization of hardware including memory and facilitates quicker reproduction and reorganization of data such as during a node add process.

[0021] Figure 1 illustrates an exemplary partitioned data storage network or system 100 useful for implementing aspects of the invention. The data storage system 100 includes a number of clients 104 that communicate with a number of application nodes 108 via one or more digital communications networks 106 (e.g., local area networks, wide area networks, the Internet, and so on with wired or wireless connections and/or data transfer). The application nodes 108 store and access data in a clustered data store 110 and communicate via network 112 with a plurality of data nodes 114, 116, 118, 120 storing partitioned data 115, 117, 119, 121 (e.g., horizontal partitions of one or more data tables). The clustered data store 110 is managed, at least in part, by storage engine 130 (e.g., a MySQL NDB Cluster storage engine or the like), and the storage engine 130 includes a table reorganization mechanism 132 that utilizes a HashMap partitioning module (e.g., with a partition function) 140 to determine how to partition data 115, 117, 119, 121.

[0022] The partitioning of data in the data store 110 on nodes 114, 116, 118, 120 may be modified, for example, when a node add module 136 is utilized by the storage engine 130 to add a node(s) to the data store 110. Typically, the storage engine 130 supports both list partitioning 144 and modulo hashing or hash partitioning 148, and the HashMap partitioning module 140 may be thought of as utilizing a combination of these partitioning techniques to achieve desirable online partitioning (e.g., such as partitioning data when the nodes 118, 120 were added to the data store 110 with minimal data copy and with even data distribution in the four nodes 114, 116, 118, 120). As part of managing the partitioning or reorganization of data, the table reorganization mechanism 132 may access memory 150 that stores a partition distribution mapping 152 indicating which portions of a table (e.g., rows or data entries) belong in each partition or node, a number of partitions 154 to be used (typically one partition per node to be used to store a data table), a number of data elements 156 (e.g., data entries/rows), and a primary key 158 (e.g., a unique identifier for a data entry) to be used in the partitioning process. In a typical implementations the partitioning module 140 may not need to know the data element number 156 but may instead access the following information from memory 150: the current number of map entries, how the entries map to existing partitions, and how many partitions to redistribute to.

[0023] The clients 104 may take a variety of hardware forms such as personal, laptop, notebook, and other computers or computing devices, cellular phones, personal data assistants,

servers, and so on, and the clients 104 may utilize a wide variety of interfacing software applications including, but not limited to, JDBC, ODBC, NDB API, LDAP, webservices, and the like. The clients 104 may transmit database queries or other data requests over the network(s) 106 to the application nodes 108. The application nodes 108 may also be implemented in or using a variety of computers or computing devices such as servers using one or more of NDB API, MySQL Server, OpenDS, openldap, SailFin, GlassFish, FreeRADIUS, and other for interfacing with the clients 104 and/or with the data store 110, and the nodes 108 may run applications such as web services, search engines, and so on.

[0024] The data nodes 114, 116, 118, 120 generally will comprise servers/hosts and data storage devices such as disks, disk arrays, tape-based storage devices, optical data storage devices, and the like. The storage engine 130 may run on one or more of the data nodes 114, 116, 118, 120 or may run on another device that may be adapted with one or more processors managing operation of input/output devices and memory and running software modules or programs that may be provided via computer-readable medium adapted to cause a computer or the system 100 or data store 110 to perform the functions described herein. In this discussion, computer and network devices and data store devices are described in relation to their function rather than as being limited to particular electronic devices and computer architectures. To practice the invention, the computer devices and network devices may be any devices useful for providing the described functions, including well-known data processing and communication devices and systems such as desktop computers, and even personal digital assistants, personal, laptop, and notebook computers with processing, memory, and input/output components, and server devices configured to maintain and then transmit digital data over a communications network. Data, including device qualification data, device simulation data, event messages/files simulating device operations in response to device qualification data, and transmissions to, from, and within systems is typically communicated in digital format following standard communication and transfer protocols, such as TCP/IP, HTTP, HTTPS and the like, but this is not intended as a limitation of the invention.

[0025] A useful approach to explaining the operation of the system 100 may be to continue with the relatively simple example began in the background section and partitioning performed with the storage engine 130 using the HashMap partitioning module 140. In this example, it is assumed that two of the nodes 114, 116 are initially being used to store a table of

data that is horizontally partitioned into two partitions 115, 117. The partition of data 115, 117 on nodes 114, 116 may take the form: Node 114 - (1,"Tomas"); (3,"Jonas"); (5,"John"); (7,""); (9,""); and (11,""); and Node 116 - (2,"Kent"); (4,"Frazer); (6,"Eliza"); (8,""); (10,""); and (12,"").

[0026] According to one embodiment, the storage engine 130 may call the table reorganization mechanism 132 to reorganize the partitioned data such as to add data node 118 via the add node module 136. In this regard, the HashMap partitioning mechanism 140 is called, and it retrieves the primary key 158 (e.g., the identifiers 1, 2, 3, and so on), the number elements 156 in the table (e.g., twelve in this example), and number of partitions desired 154 (e.g., 3 partitions after reorganization). The HashMap partitioning mechanism 140 then generates a partition distribution mapping 152 that may be used by the table reorganization mechanism 132 in reorganizing the partitioned data 115, 117 from two partitions/nodes to partitioned data 115, 117, 119 with three partitions/nodes. The partitioning may take the ideal form of: Node 114 - (1,"Tomas"); (5,"John"); (9,""); and (11,""); Node 116 - (2,"Kent"); (6,"Eliza"); (10,""); and (12,""); and Node 118 - (3,"Jonas"); (4,"Frazer); (7,""); and (8,""). In this partitioning or reorganization of the data in the table we see that the number of elements is twelve and the number of partitions is three, and the HashMap partitioning module 140 is able to create a partition distribution mapping 152 indicating in which data partition each entry or row should be stored. It is seen that an even distribution of data (here 4 data entries per node but this example can, of course, be applied to tables with thousands to millions or more data entries across many nodes). Significantly, this example also shows that the mapping 152 provided by partitioning module 140 is such that node 114 and node 116 keep the same data in the two and three node partitioning configurations of the data table (with the exception of the data from each node 114, 116 that was moved to node 118).

[0027] To further understand the benefit of having the data stay in the same place/node as in the latter example, it may be useful to describe the general algorithm implemented by the table reorganization mechanism 132 when performing online reorganization of data from two to three nodes. An initial step may be to setup triggers to capture changes of the data in the table affected by the reorganization (e.g., a table with partitions on one or more of the nodes 114, 116, 118, 120 of data store 110). The trigger(s) is preferably adapted to make sure that the same change happens to the data that is copied during a copying step. Then, a copying step

may be performed by the reorganization mechanism 132 to copy data from the old partitioning to the new partitioning (e.g., in the above 3-node example, data is copied from node 114 and node 116 to node 118 to create the third partition in node 118). The old partitioning of two nodes is changed to the new partitioning of three nodes by the storage engine 130 and/or the reorganization mechanism 132. Next, the old partitioning may be deleted (e.g., the data entries no longer needed on the first two nodes, in the above example, may be deleted).

[0028] As a result of this partitioning technique, the retention of data at original or existing nodes provides large benefits in performing the step or process of copying data to the new partition or node. It should be understood that no new memory is needed to keep the copy of the new data partitioning in the existing nodes. To understand why this is so, it may be helpful to look at a smaller data partitioning example in which conventional modulo hashing is utilized to form a new partitioning or distribution mapping. In this example, the old partitioning of data may call for two partitions 115, 117 on nodes 114, 116 appearing as: Node 114 – (1,"Tomas"); (3,"Jonas"); and (5,"John"); and Node 116 - (2,"Kent"); (4,"Frazer"); and (6,"Eliza"), for a table with 6 entries or rows and a primary key of the integers 1 to 6. Using modulo hashing to add a node and provide three partitions of data 115, 117, 119 on nodes 114, 116, 118 may provide the following new partitioning: Node 114 - (1,"Tomas") and *(4,"Frazer"); Node 116 - (2,"Kent") and *(5,"John"); and Node 118 - *(3,"Jonas") and *(6,"Eliza"). Note, the actual data needed to be copied to the new partitioning is marked with an "*". The other, unmarked data already resides on the node in the old partitioning, and does not need to be copied, and the example shows a need for extra memory in performing modulo hash-based partitioning.

[0029] In contrast, using the HashMap partitioning module 132 provides a more ideal setup of new partitioning or reorganization of the table of data distributed in the clustered data store 110. Specifically, the old or original partitioning is again: Node 114 – (1,"Tomas"); (3,"Jonas"); and (5,"John"); and Node 116 - (2,"Kent"); (4,"Frazer"); and (6,"Eliza"). To add a new node (node 118) to the cluster used to store the table of data, the storage engine 130 calls the table reorganization mechanism 132, which, in turn, uses the HashMap partitioning module 140 to produce distribution mapping 152 providing the following new partitioning: Node 114 - (1,"Tomas") and (5,"John"); Node 116 - (2,"Kent") and (6,"Eliza"); and Node 118 - *(3,"Jonas") and *(4,"Frazer"). In this reorganization process, it can be seen that new

memory use on the original nodes 114, 116 is not needed. Also significant is the fact that much less copying of data is required because only the data stored in the new partition on node 118 (i.e., the added or new node) needs to be copied from the original or old nodes 114, 116 (or from partitioned data 115, 117).

[0030] Hence, an aspect of the invention may be considered how the HashMap partitioning module 140 is configured to create a distribution mapping 152 that has the properties like the latter example. Another aspect of the invention may then be the usage of the new table partitioning in the context of online reorganizing data in a database (e.g., a database arranged as a table that is stored in a distributed manner in clustered data store 110). The following provides one embodiment of the HashMap partitioning module 140 or its partition function. In the description below, P(PK) is used as the function that the module 140 or mechanism 130 uses to decide which partition (e.g., node in the examples) data or data entries should reside in, with "PK" being the primary key of the table. In the above exemplary implementation of P(PK), PK is `id` and the P(PK) evaluates to 1, 2, or 3, corresponding to the nodes 114, 116, 118 of system 100.

[0031] A new partitioning-algorithm has been introduced, labeled HashMap partitioning here, which may be considered modulo hash plus list partitioning. A storage engine 130 may support or implement modulo hashing or hash partitioning 148, and the modulo hashing function may be described with the partition function: P(PK) = md5(PK) % #PARTITIONS (wherein md5 refers to commonly used hash function sometimes labeled Message-Digest algorithm 5). In contrast, the storage engine 130 of embodiments of the invention may build upon such hash partitioning 148 by providing a HashMap partitioning 140 described with the partition function: P(PK) = map[md5(PK) % elements in(map)]. With this partitioning type, the partition function P(X) (or function implemented by module 140) is not a direct function of the number of partitions. As a result, adding partitions to a table does not affect the partition function, and the real reorganization is performed when switching the HashMap or partition distribution mapping 152 for a table. It will also be understood that use of HashMap partitioning with module 140 also easily supports nodes with different sizes, which is useful in some storage systems 100 when such nodes are also supported by the particular storage engine 130.

[0032] In practice, the evenness of the HashMap partitioning may depend on the relationship between the number of map entries and the number of nodes. For example, if the number of nodes is a factor of the number of map entries then data is balanced. If not, then there may be an imbalance. The resulting imbalance as a percentage is given by the equation 100/CEILING((#MAP_ENTRIES/#NODES),1). In some embodiments, a selection of 240 is made as a default number of map entries as part of the design of the system 100 in part because the number 240 has factors including 1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 16, 20, 24, 30, 40, and 48.

[0033] Figures 2-5 provide another working example of operations of a data storage system that is used to provide online adding of a node (or a node group of two or more nodes providing data backup or replication) and in which partitioning of data is performed using HashMap partitioning rather than conventional modulo hashing. Figure 2 schematically illustrates a data storage system 200 used to support an application 208 in storing/accessing an application data stet or table 204. The data table 204 is shown to be a relatively simple database with four columns including fields for an identifier that may be used as the primary key, a first name, a last name, and a country, and the table 204 is shown to only include four data elements or entries to simplify explanation of partitioning of data. The application 208 may communicate as shown at 208 with data storage or clustered data storage that includes a first node group 220, such as to transmit the data of table 204, to query the stored table, or modify the table.

[0034] The node group 220 is shown to include two data storage devices/systems (e.g., a server and disk array or the like), and each node in the group 220 (labeled Node 1) is shown to store a copy of the table 204 in a data partition or bucket 222, with entries/rows 224, 225, 226, 227. A storage engine 230 is provided to manage operation of the node group 220 including partitioning of data, with Figure 1 showing the system 200 using only one node group 220 and one partition 222 on that node (i.e., Node 1) to store all of the data in table 204. In performing the partitioning, the engine 230 uses the partitioning tool 234, which is adapted to implement HashMap partitioning as described above as a combination of hash and list partitioning to create a partition distribution map that defines which partitions or buckets data is placed in system 200. The storage engine 230 may be operated/commanded by a system manager (or in response to communications 210 from application 208) to begin an add node (or add group

node) online process. As shown in Figure 2, a new or second node group 240 is added to the data cluster of system 200 and made available to the storage engine 230.

[0035] In Figure 3, the storage system 200 is shown after the partitioning tool 234 has been used to generate a distribution mapping. As discussed above, the partitioning is performed in a manner that provides both for more uniform distribution among the nodes or node groups 220, 240 but also limits the need for additional memory by retaining entries on the original partitions/nodes (except that which is moved to the new partition). As shown, the partition distribution mapping calls for two rows or data entries to be stored in each partition 222, 320 after online add node operations are complete (e.g., after partitioning or reorganizing data). The entries 224, 226 with primary keys 1 and 3 are retained on the first node 220 or partition 222 and the entries 225, 227 with primary keys 2 and 4 are mapped to the new partition 320 in the added node 240. Figure 3 also illustrates copying of these data entries 324, 328 into partition 320 as shown at step 310. As shown, no extra space or memory is needed on the existing nodes, here just the nodes of node group 220 but in a more typical examples a plurality of nodes would not be forced to provide space to support the new partitioning by having data copied to them from other existing nodes as part of a large data reshuffling.

[0036] Figure 4 shows the storage system 200 after copying of data from the original node 220 to the new or added node 240 is complete. Also, the storage system 200 shows that distribution 410 is being switched to reflect the use of two data partitions or buckets 222 320 for the data from (or accessed by) application 208. As shown, the node 222 still contains all original data entries 224, 225, 226, 227 after the copying and even as or after the switching of the distribution to the new partition distribution mapping. In part, this is because embodiments of the partitioning techniques described herein may call for the storage system 200 (or storage engine 234) to complete the running scans (or wait for running scans to complete) prior to taking a next step in the reorganization. For example, the next step may involve deleting all data or entries that were previously copied to the new partition 320 from the original partition 222. Figure 5 illustrates the storage system 200 after completion of the deleting of rows 225, 227, which were copied to partition 320 as rows/entries 324, 328, from the old or original partition 222. In Figure 5, the reorganization of the storage system 200 has been completed and partitioning was performed as part of adding node group 240 in an online add node. The process was completed in an efficient manner without requiring additional

space in the memory of the original node 222 and provided uniform distribution of the data entries, with these two aspects being provided by the partitioning tool utilizing HashMap partitioning in which generating a partition distribution mapping was performed using modulo hashing as a function of the number of entries or elements in the table or map rather than as a function of the number of desired partitions.

[0037] Figure 6 illustrates exemplary steps or processes that may be performed during a table or data reorganization within a data storage system or network, such as by steps carried out by a storage engine managing partitioning and storage to nodes within a clustered data store to provide data in databases to applications. The method 600 begins at 610 with a determination to reorganize a table of data, and this may often occur when it is determined that additional nodes should be used to store the table data such as when the size of data set exceeds a particular size to facilitate searching or otherwise utilizing the data entries in the table. The method 600 continues at 620 with using HashMap partitioning as discussed above to add one or more new partitions to a table. For example, an existing table may be horizontally partitioned into 8 partitions with each partition stored on a different node, and it may be determined that the data should be further partitioned to 10, 12 or more partitions on a similar (or different) number of nodes. In step 620, the HashMap partitioning mechanism or partition function is used to generate a new partition distribution mapping of the data (e.g., modulo hashing performed as a function of the number of data elements or in combination with list partitioning) that provides for uniform distribution of the data over the nodes and, typically, retains data upon original nodes during copying/reproduction steps (to avoid requiring additional space on the original or old nodes).

[0038] The method 600 continues at 630 with creating reorganization triggers. These triggers are typically adapted to transfer data used to maintain and organize the table data from the old partitions to the new partitions. For example, data manipulation language (dml) statements (which are statements used to store, retrieve, modify, and erase data from a database) associated with the affected data entries may be transferred from the original or old table partitions to the new or added partitions (or associated nodes) such that an update on a row that will/should move from the old to the new partition is applied to both copies (e.g., the copy still in the original/old partition as well as the copy newly created in the new partition). At step 636, the rows or data entries identified by the new distribution mapping (or HashMap

partitioning) for moving to the new or added partition(s) are copied from the one or more old or original partitions to the new or added partition. In step 640, the commit record is written or a step is taken to commit transactions (e.g., transaction statement such as SQL statements that have changed data and have been entered by the storage engine (e.g., by NDB Cluster storage engine) but have not yet been saved).

[0039] Method 600 continues at 650 with switching over to the new data distribution such that new partitions are used for transactions/scans, and, in some embodiments, step 650 may involve simultaneously altering trigger-based change propagation to be from the new partitions back to the old partitions. At step 654, the method 600 involves waiting for scans using the old distribution to be finished before continuing with the process 600. At 660, the method 600 includes enabling replication triggers on the new partitions. At 670, replication triggers for the moved rows are disabled on old partitions on epoch boundaries. In step 680, reorganization triggers are dropped, and then at 686 the method includes scan/deleting the moved rows from the old partitions. The method 600 ends at 690 such as with cleanup of miscellaneous storage system resources.

[0040] Errors may occur during table reorganization, and error handling may be provided by a storage engine to require that anything that goes wrong before the commit step 640 is performed results in reorganization being rolled back. Note, a node failure is typically not considered something the goes wrong for error handling. A complete cluster crash before the commit step 640 may make the cluster or storage engine perform a rollback on a subsequent cluster start. A complete cluster crash after the commit step 640 (but before completion at 690) may make the cluster or storage engine perform rollforward on a subsequent cluster start.

[0041] Although the invention has been described and illustrated with a certain degree of particularity, it is understood that the present disclosure has been made only by way of example, and that numerous changes in the combination and arrangement of parts can be resorted to by those skilled in the art without departing from the spirit and scope of the invention, as hereinafter claimed.

WE CLAIM:

1. A method for reorganizing a table of a database, comprising:

providing a data storage cluster including a first node and a second node;

storing the table in the data storage cluster with a first partition comprising a set of rows in the first node and a second partition comprising a set of rows in the second node;

modifying the data storage cluster to include a third node for storing data from the table;

with a storage engine managing the data storage cluster, adding a third partition to the table including using a partitioning mechanism to create a distribution mapping for data elements in the first, second, and third partitions;

copying a portion of the rows of the table from both the first and second nodes to the third partition of the third node based on the distribution mapping; and

deleting the copied rows of the table from the first and second nodes, wherein the distribution mapping only calls for copying from the first and second nodes and not to the first and second nodes.

2. The method of claim 1, wherein after the copying and the deleting, data from the table is substantially uniformly distributed over the first, second, and third partitions.

3. The method of claim 1, wherein the partitioning mechanism uses hash partitioning as a function of the data elements in the table.

4. The method of claim 1, wherein the partitioning mechanism comprises a partition function combining hash and list partitioning, the hash partitioning being performed without regard to a number of the partitions in the reorganized table.

5. The method of claim 1, further, prior to the deleting, comprising switching over distribution to use the first, second, and third partitions for data transactions according to distribution mapping and further comprising waiting for scans using prior distribution of the first and second partitions to finish before performing the deleting.

6. The method of claim 1, further comprising, prior to the copying, creating reorganization triggers that transfer data manipulation language statements to the third partition that correspond to the portion of the rows copied to the third partition.

7. A data storage system for storing rows of data in a table, comprising:

a server running a storage engine including a partitioning module;

a set of data nodes managed by the storage engine; and

a table of data horizontally partitioned with a partition with a set of rows in each of the data nodes, wherein the storage engine operates the partitioning module to generate a distribution mapping defining a new partitioning of the data table when a new data node is added to the set of data nodes, the distribution mapping providing substantially uniform distribution of rows of the data across partitions in the new partitioning and retaining a subset of the rows of the data in each of the original ones of the data nodes.

8. The system of claim 7, wherein the partitioning module comprises a partition function that combines hash and list partitioning to create the distribution mapping.

9. The system of claim 7, wherein the partitioning module includes modulo hash partitioning as a function of elements in the data table.

10. The system of claim 7, wherein the storage engine copies a portion of the rows of the data from each of the partitions on the original data nodes to a partition on the new data node according to the distribution mapping without copying any of the rows of the data to the original ones of the data nodes.

11. The system of claim 7, wherein the partitioning module implements a partition function defined by P(PK) = map[md5(PK) % elements in(map)], wherein PK is a primary key of the data table.

12. A computer program product including computer useable medium with computer readable code embodied on the computer useable medium, the computer readable code comprising:

computer readable program code devices configured to cause a computer to effect adding a node to a data storage cluster, wherein the data storage cluster stores a table of data in a horizontally partitioned manner over a number of nodes according to a first distribution mapping;

computer readable program code devices configured to cause the computer to effect to create a second distribution mapping defining partitioning of the table of data including an additional partition in the added node; and

computer readable program code devices configured to cause the computer to effect copying one or more rows associated with the table of data from the number of nodes to the added node, the copied rows being defined by the second distribution mapping and the copying excluding copying between the number of nodes, whereby data is retained on the number of nodes.

13. The product of claim 12, further comprising computer readable program code devices configured to cause the computer to effect deleting the copied rows from the number of nodes, whereby data is retained on the number of nodes except data in the copied rows.

14. The product of claim 12, wherein the second distribution mapping defines a substantially uniform distribution of data from the table over the nodes of the data storage cluster.

15. The product of claim 14, wherein the second distribution mapping defining comprises modulo hashing the data in the table as a function of elements in the table.

16. The product of claim 12, wherein the second distribution mapping defining comprises performing HashMap partitioning of the data table.

17. The product of claim 16, wherein the HashMap partitioning comprises a combination of hash and list partitioning.

18. The product of claim 12, further comprising computer readable program code devices configured to cause the computer to effect deleting the copied rows from the number of nodes after switching distribution to the second distribution mapping and waiting for scans to the number of nodes based on the first distribution mapping to complete.

19. The product of claim 12, further comprising computer readable program code devices configured to cause the computer to effect creating reorganization triggers transferring dml from partitions based on the first distribution mapping to partitions based on the second distribution mapping and, after the copying, committing to transactions to the table of data stored in the data storage cluster.

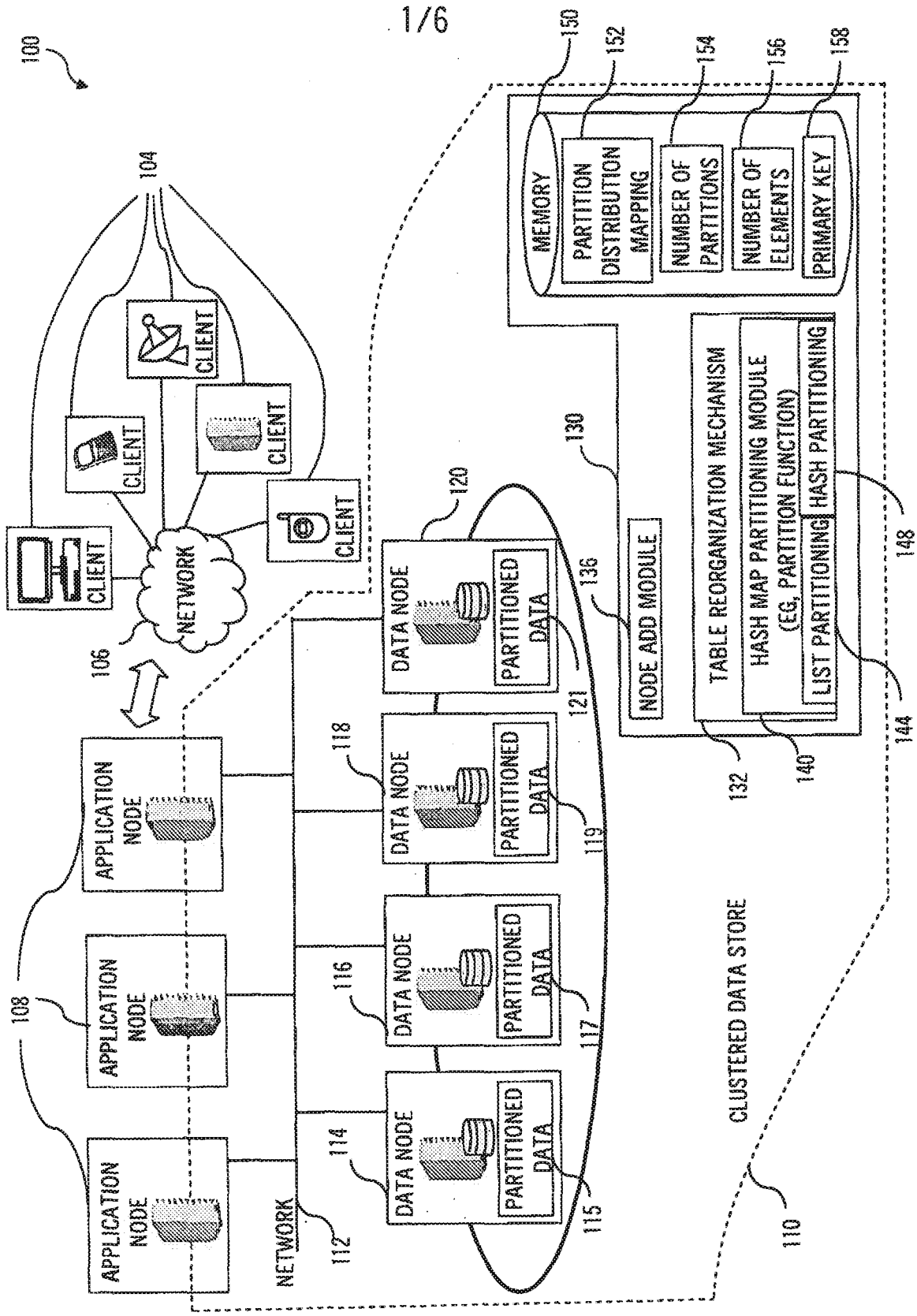
100
104
CLIENT
CLIENT
CLIENT
CLIENT
CLIENT
NETWORK
106
108
APPLICATION NODE
APPLICATION NODE
APPLICATION NODE
NETWORK
112
114
DATA NODE
PARTITIONED DATA
115
116
DATA NODE
PARTITIONED DATA
117
118
DATA NODE
PARTITIONED DATA
119
120
DATA NODE
PARTITIONED DATA
121
CLUSTERED DATA STORE
110
130
NODE ADD MODULE
136
MEMORY
150
PARTITION DISTRIBUTION MAPPING
152
NUMBER OF PARTITIONS
154
NUMBER OF ELEMENTS
156
PRIMARY KEY
158
TABLE REORGANIZATION MECHANISM
132
HASH MAP PARTITIONING MODULE (EG, PARTITION FUNCTION)
140
LIST PARTITIONING
HASH PARTITIONING
144
148

FIG. 1

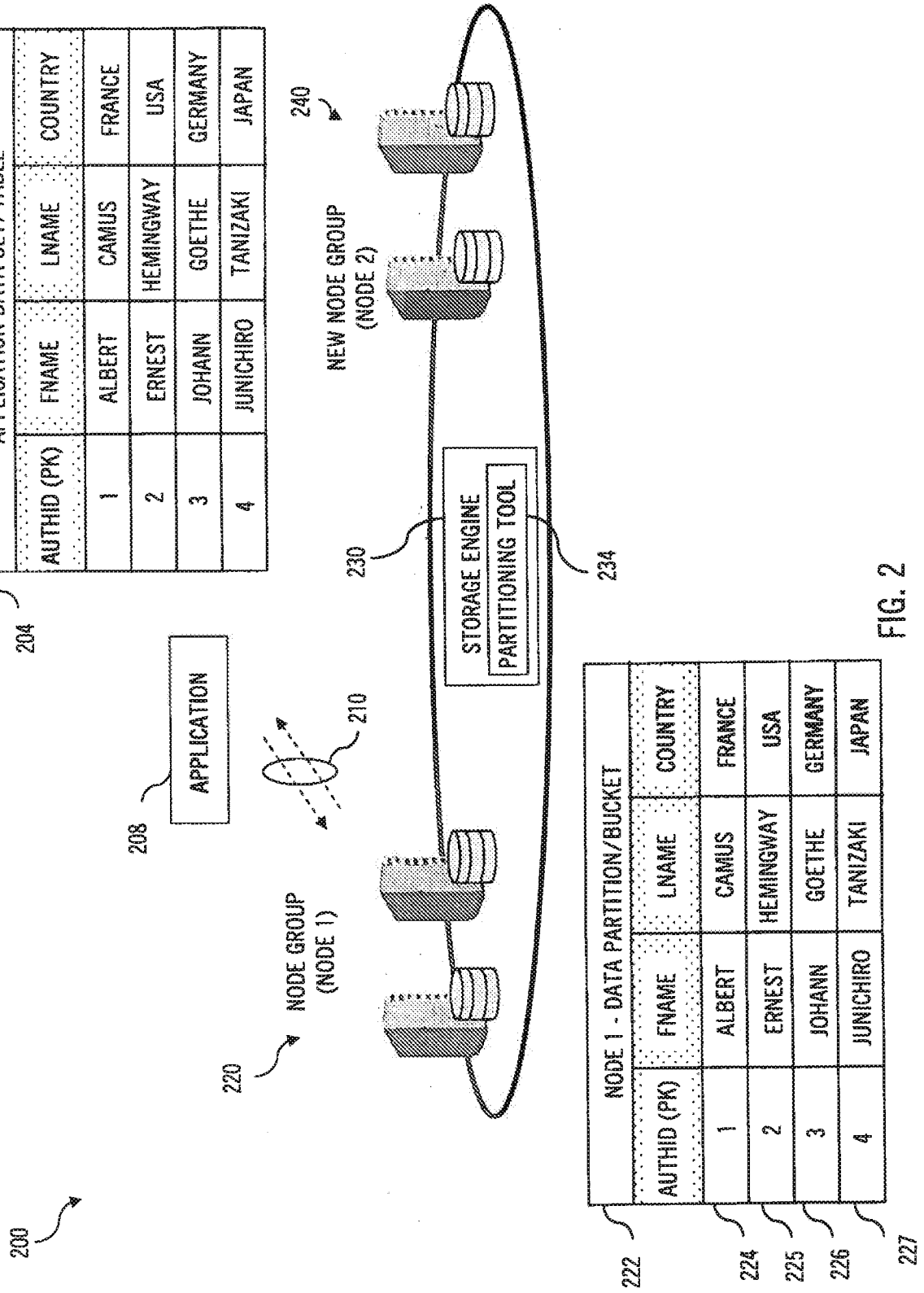
200
204
APPLICATION DATA SET/TABLE
AUTHID (PK)
FNAME
LNAME
COUNTRY
1
ALBERT
CAMUS
FRANCE
2
ERNEST
HEMINGWAY
USA
3
JOHANN
GOETHE
GERMANY
4
JUNICHIRO
TANIZAKI
JAPAN
208
APPLICATION
210
220
NODE GROUP
(NODE 1)
230
STORAGE ENGINE
PARTITIONING TOOL
234
240
NEW NODE GROUP
(NODE 2)
222
NODE 1 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
1
ALBERT
CAMUS
FRANCE
2
ERNEST
HEMINGWAY
USA
3
JOHANN
GOETHE
GERMANY
4
JUNICHIRO
TANIZAKI
JAPAN
224
225
226
227

FIG. 2

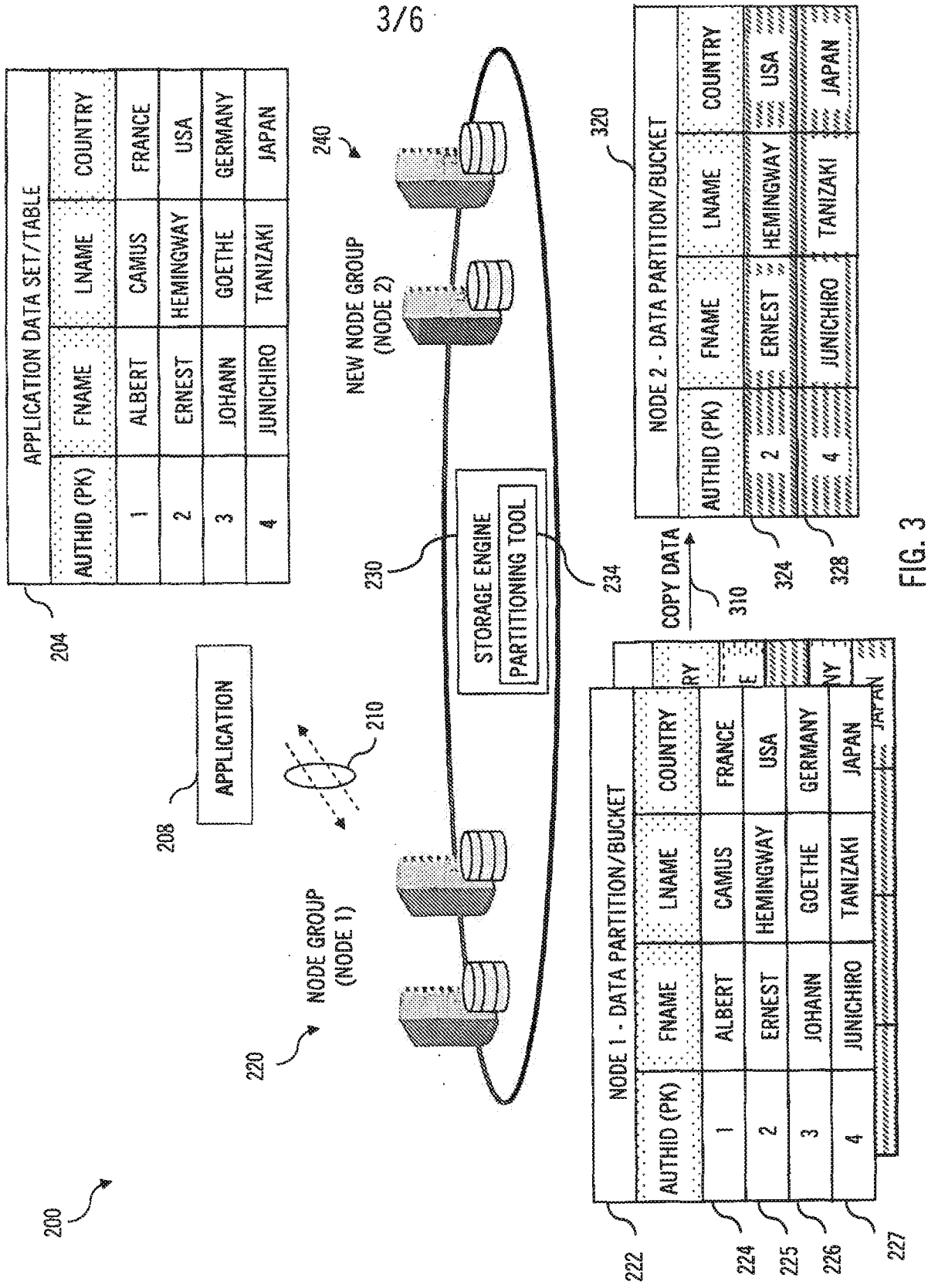
200
204
APPLICATION DATA SET/TABLE
AUTHID (PK)
FNAME
LNAME
COUNTRY
1 ALBERT CAMUS FRANCE
2 ERNEST HEMINGWAY USA
3 JOHANN GOETHE GERMANY
4 JUNICHIRO TANIZAKI JAPAN
208
APPLICATION
210
220
NODE GROUP (NODE 1)
230
STORAGE ENGINE
PARTITIONING TOOL
234
240
NEW NODE GROUP (NODE 2)
COPY DATA
310
320
NODE 2 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
2 ERNEST HEMINGWAY USA
4 JUNICHIRO TANIZAKI JAPAN
324
328
222
NODE 1 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
1 ALBERT CAMUS FRANCE
2 ERNEST HEMINGWAY USA
3 JOHANN GOETHE GERMANY
4 JUNICHIRO TANIZAKI JAPAN
224
225
226
227

FIG. 3

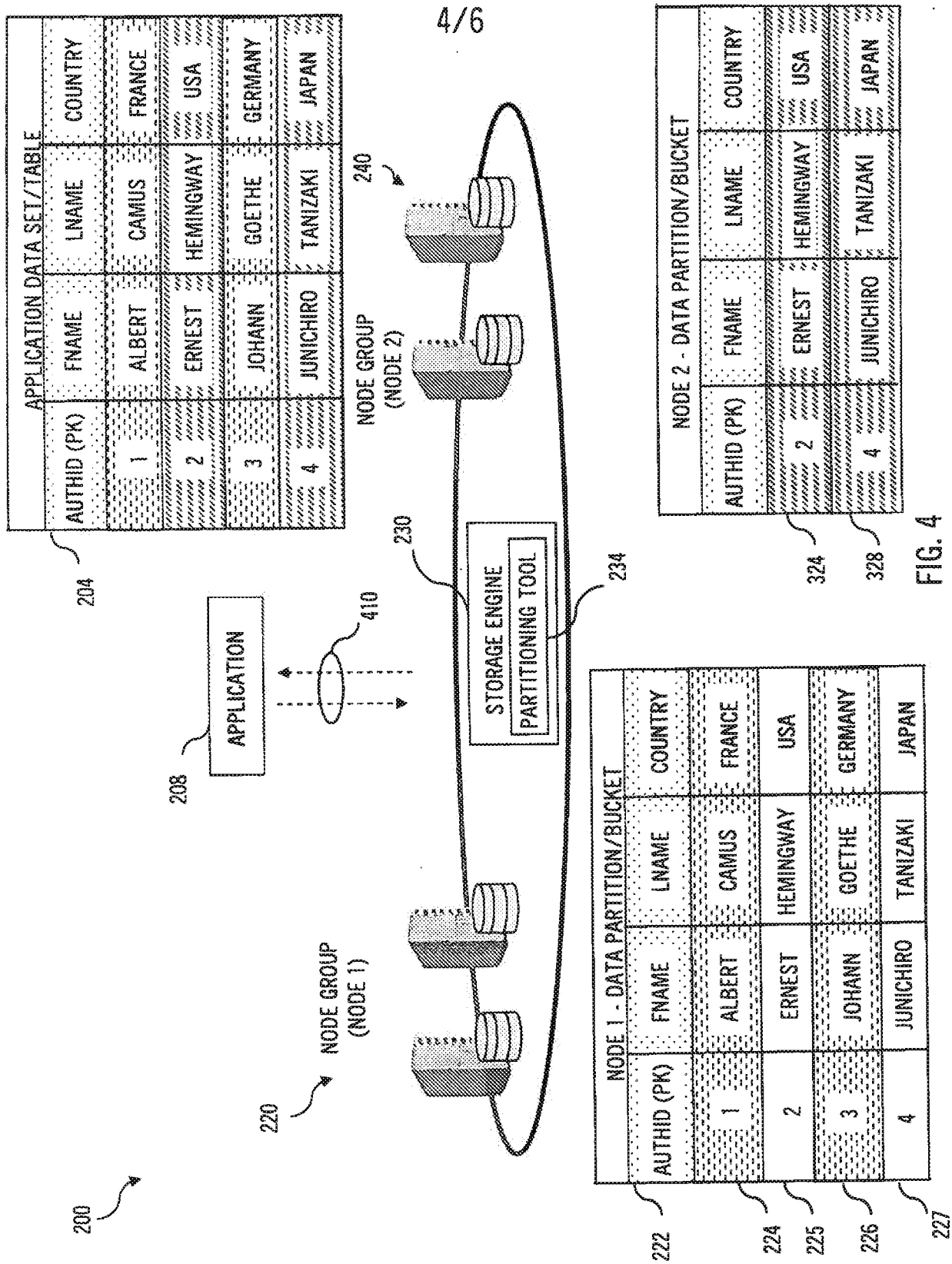
APPLICATION DATA SET/TABLE
AUTHID (PK)
FNAME
LNAME
COUNTRY
1 ALBERT CAMUS FRANCE
2 ERNEST HEMINGWAY USA
3 JOHANN GOETHE GERMANY
4 JUNICHIRO TANIZAKI JAPAN
204
NODE GROUP (NODE 2)
240
230
STORAGE ENGINE
PARTITIONING TOOL
234
APPLICATION
208
410
NODE GROUP (NODE 1)
220
200
NODE 1 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
1 ALBERT CAMUS FRANCE
2 ERNEST HEMINGWAY USA
3 JOHANN GOETHE GERMANY
4 JUNICHIRO TANIZAKI JAPAN
222
224
225
226
227
NODE 2 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
2 ERNEST HEMINGWAY USA
4 JUNICHIRO TANIZAKI JAPAN
324
328

FIG. 4

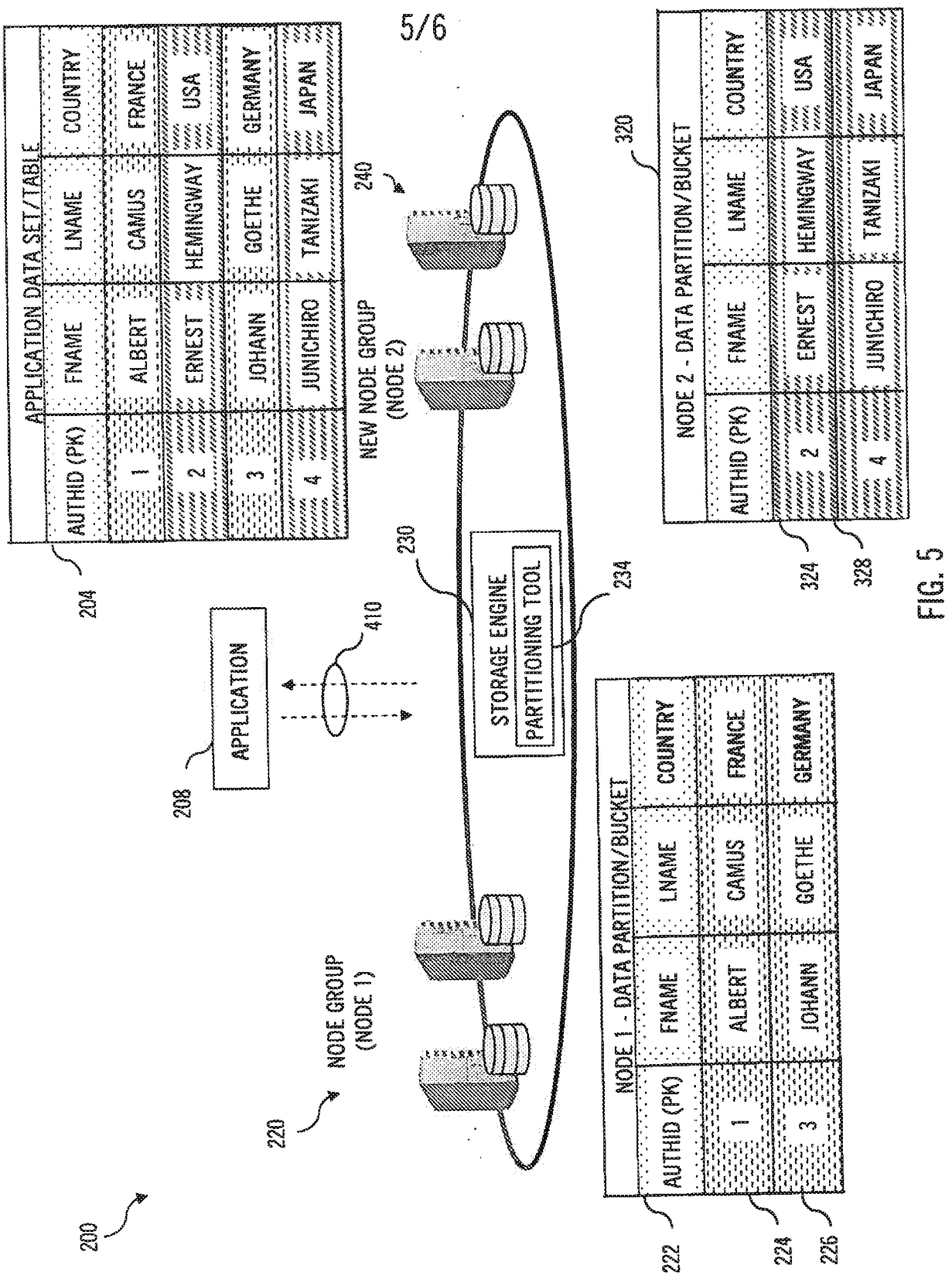
APPLICATION DATA SET/TABLE
AUTHID (PK)
FNAME
LNAME
COUNTRY
1 ALBERT CAMUS FRANCE
2 ERNEST HEMINGWAY USA
3 JOHANN GOETHE GERMANY
4 JUNICHIRO TANIZAKI JAPAN
204
NEW NODE GROUP
(NODE 2)
240
APPLICATION
208
410
NODE GROUP
(NODE 1)
220
230
STORAGE ENGINE
PARTITIONING TOOL
234
200
NODE 1 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
1 ALBERT CAMUS FRANCE
3 JOHANN GOETHE GERMANY
222
224
226
320
NODE 2 - DATA PARTITION/BUCKET
AUTHID (PK)
FNAME
LNAME
COUNTRY
2 ERNEST HEMINGWAY USA
4 JUNICHIRO TANIZAKI JAPAN
324
328

FIG. 5

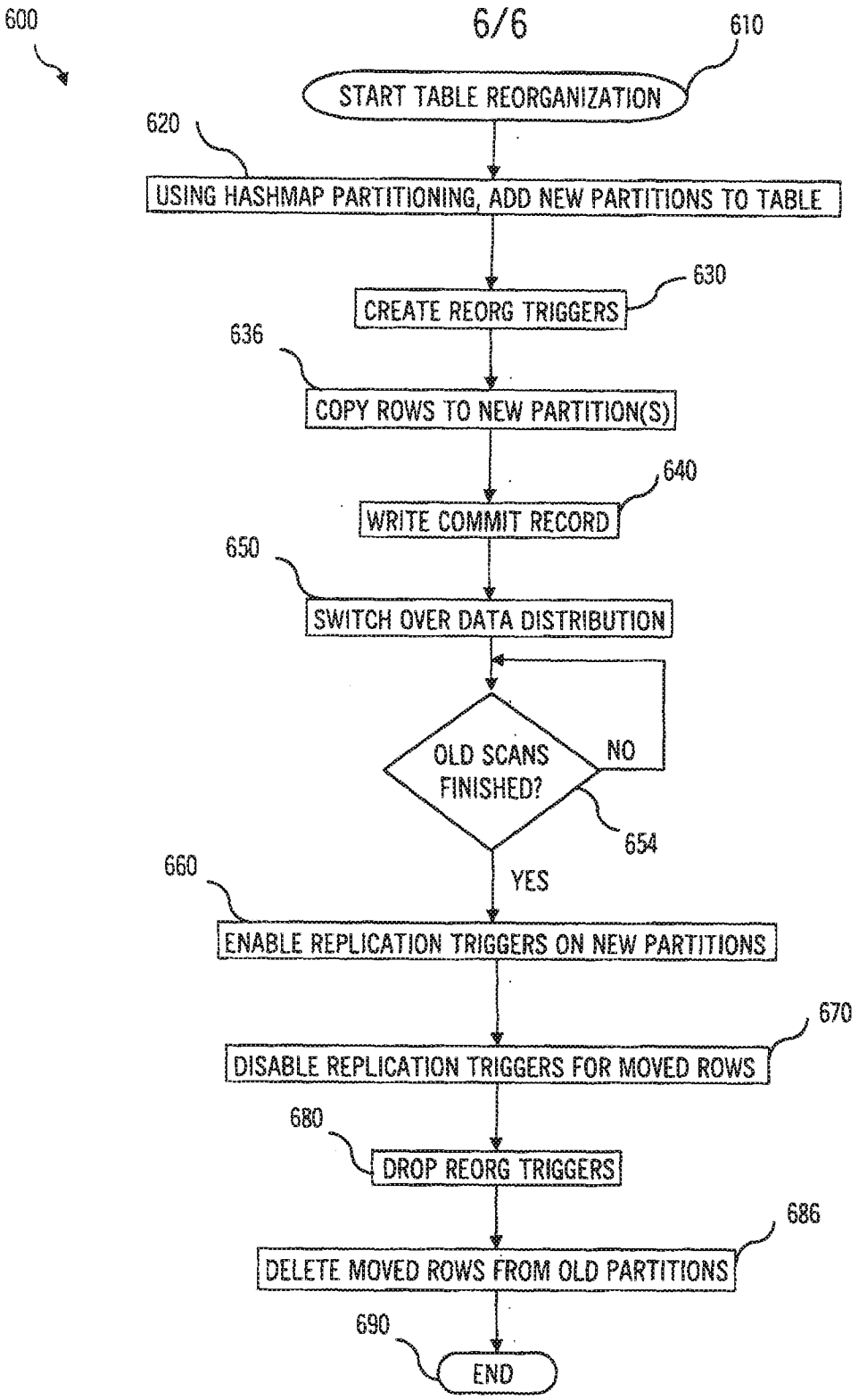
600
610
START TABLE REORGANIZATION
620
USING HASHMAP PARTITIONING, ADD NEW PARTITIONS TO TABLE
630
CREATE REORG TRIGGERS
636
COPY ROWS TO NEW PARTITION(S)
640
WRITE COMMIT RECORD
650
SWITCH OVER DATA DISTRIBUTION
OLD SCANS FINISHED?
NO
654
YES
660
ENABLE REPLICATION TRIGGERS ON NEW PARTITIONS
670
DISABLE REPLICATION TRIGGERS FOR MOVED ROWS
680
DROP REORG TRIGGERS
686
DELETE MOVED ROWS FROM OLD PARTITIONS
690
END

FIG. 6