

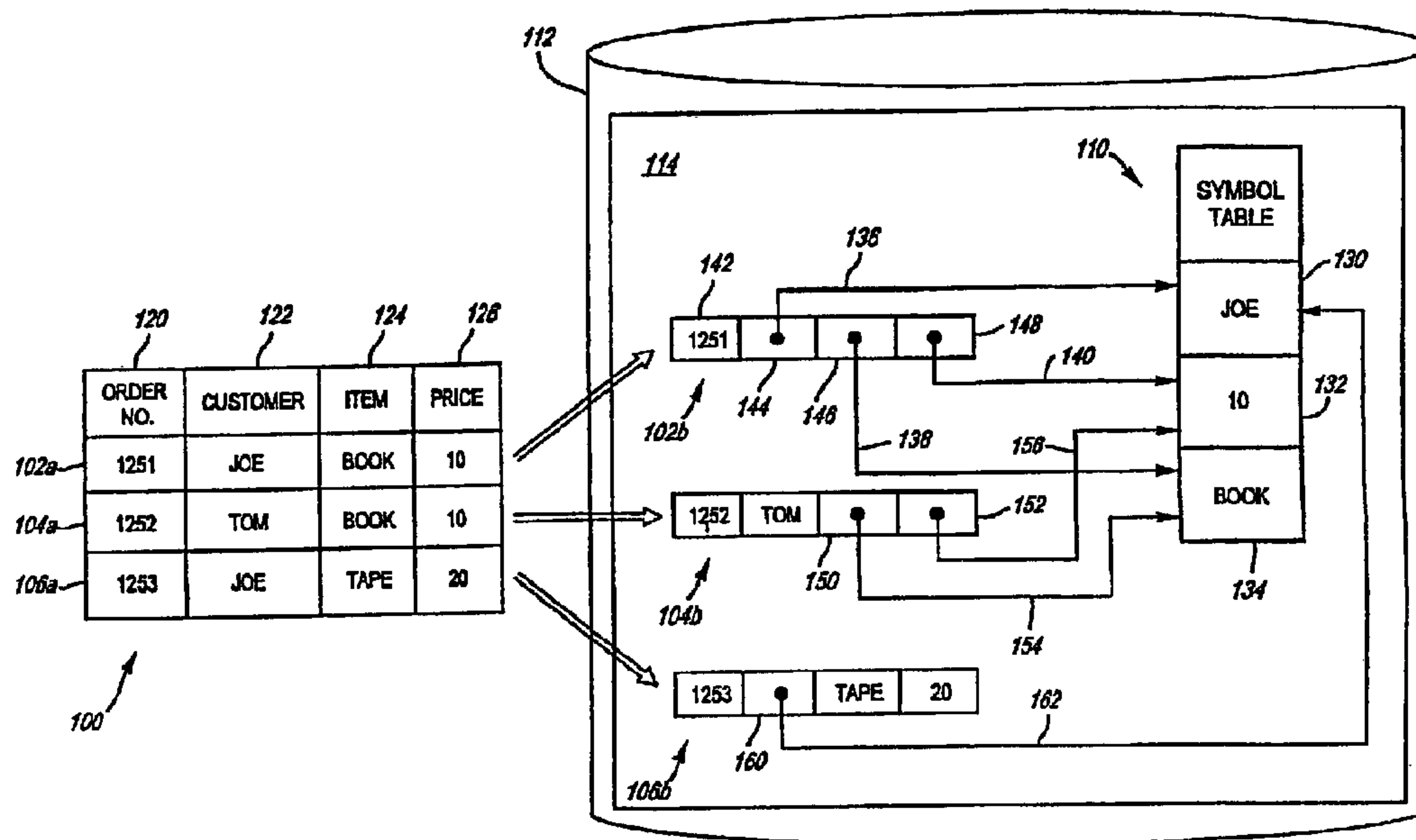


(86) Date de dépôt PCT/PCT Filing Date: 2003/05/12
(87) Date publication PCT/PCT Publication Date: 2003/11/20
(45) Date de délivrance/Issue Date: 2013/10/08
(85) Entrée phase nationale/National Entry: 2004/11/09
(86) N° demande PCT/PCT Application No.: US 2003/014834
(87) N° publication PCT/PCT Publication No.: 2003/096230
(30) Priorités/Priorities: 2002/05/10 (US10/144,689);
2003/04/29 (US10/426,452)

(51) Cl.Int./Int.Cl. *G06F 17/30* (2006.01),
G06F 7/00 (2006.01)
(72) Inventeurs/Inventors:
POTAPOV, DMITRY M., US;
OZBUTUN, CETIN, US
(73) Propriétaire/Owner:
ORACLE INTERNATIONAL CORPORATION, US
(74) Agent: BORDEN LADNER GERVAIS LLP

(54) Titre : PROCEDE ET DISPOSITIF POUR LE STOCKAGE DE DONNEES ET L'ACCES AUX DONNEES, ET POUR
L'AMELIORATION DE LA PERFORMANCE DES FORMULATIONS DE LANGAGE D'INTERROGATION DE BASE
DE DONNEES

(54) Title: STORING AND QUERYING RELATIONAL DATA IN COMPRESSED STORAGE FORMAT



(57) Abrégé/Abstract:

A method and mechanism is disclosed for implementing storage and retrieval of data in a computing system. Data compression is performed on stored data by reducing or eliminating duplicated values in a database block. Rather than writing the duplicated data values to the data storage unit, the on-disk data is configured to reference a copy of each duplicated data value stored in a symbol table. A method and mechanism of improving performance of database query language statements is also disclosed. Information describing data duplication within a data block is maintained. In one embodiment, the data duplication information is used to reduce number of predicate evaluations on data in the data block when a database query language statement is received against the data in the data block. In another embodiment, the data duplication information is used to reduce amount of data accessed by a database query language statement.



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

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
20 November 2003 (20.11.2003)

PCT

(10) International Publication Number
WO 2003/096230 A3

(51) International Patent Classification⁷: **G06F 17/30**

(21) International Application Number:

PCT/US2003/014834

(22) International Filing Date: 12 May 2003 (12.05.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

10/144,689 10 May 2002 (10.05.2002) US
10/426,452 29 April 2003 (29.04.2003) US

(71) Applicant: **ORACLE INTERNATIONAL CORPORATION** [US/US]; 500 Oracle Parkway, Redwood Shores, CA 94065 (US).

(72) Inventors: **POTAPOV, Dmitry, M.**; 3515 Oak Knoll Drive, Redwood City, CA 94062 (US). **OZBUTUN, Cetin**; 360 Elm Street, San Carlos, CA 94070 (US).

(74) Agents: **MEI, Peter, C.** et al.; Bingham McCutchen LLP, Three Embarcadero Center, Suite 1800, San Francisco, CA 94111-4067 (US).

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

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

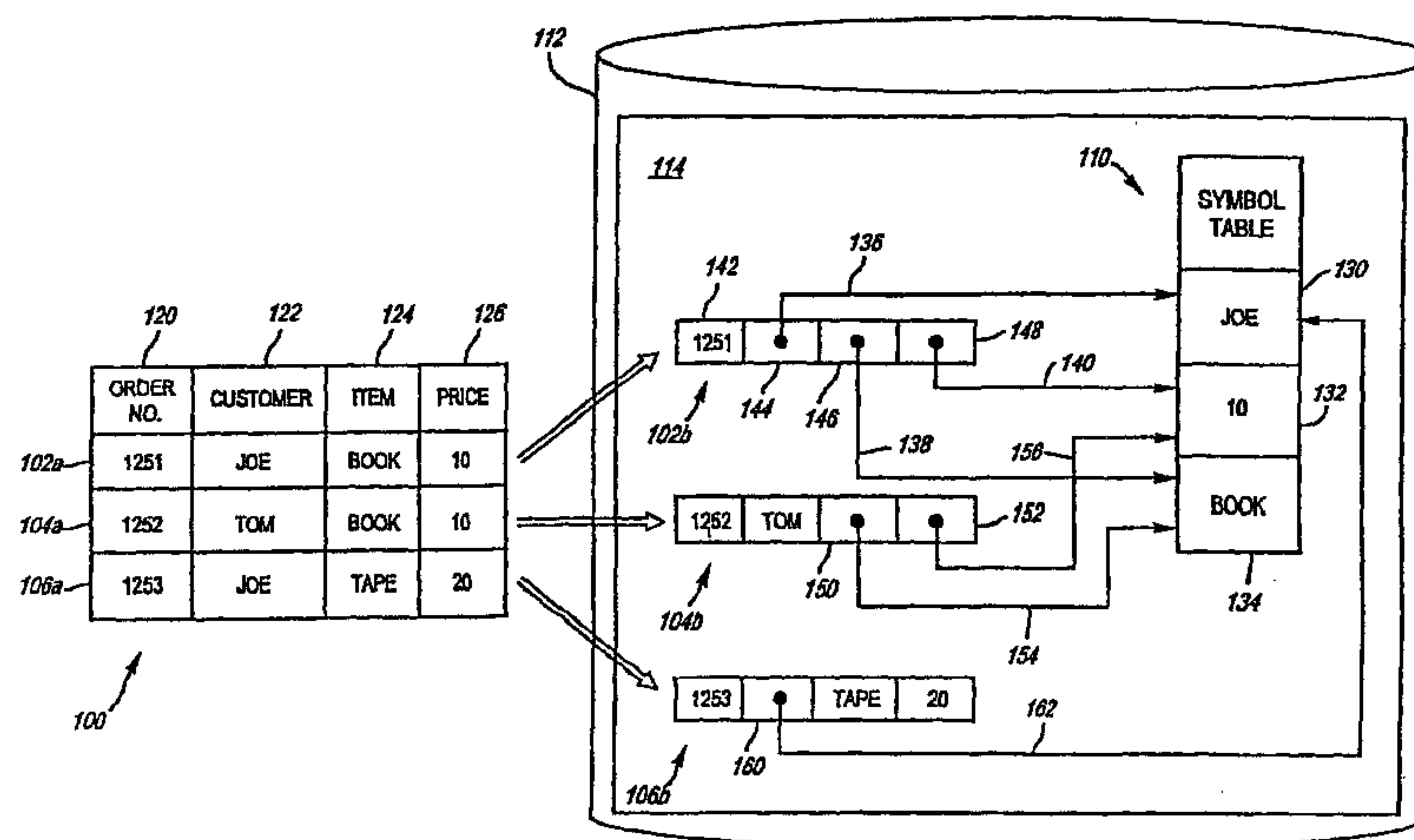
Published:

- with international search report
- before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments

(88) Date of publication of the international search report:
15 July 2004

[Continued on next page]

(54) Title: STORING AND QUERYING RELATIONAL DATA IN COMPRESSED STORAGE FORMAT



(57) **Abstract:** A method and mechanism is disclosed for implementing storage and retrieval of data in a computing system. Data compression is performed on stored data by reducing or eliminating duplicated values in a database block. Rather than writing the duplicated data values to the data storage unit, the on-disk data is configured to reference a copy of each duplicated data value stored in a symbol table. A method and mechanism of improving performance of database query language statements is also disclosed. Information describing data duplication within a data block is maintained. In one embodiment, the data duplication information is used to reduce number of predicate evaluations on data in the data block when a database query language statement is received against the data in the data block. In another embodiment, the data duplication information is used to reduce amount of data accessed by a database query language statement.

WO 2003/096230 A3

WO 2003/096230 A3



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

STORING AND QUERYING RELATIONAL DATA IN COMPRESSED STORAGE FORMAT

BACKGROUND AND SUMMARY

The present invention is related to the field of database management systems.
5 ore particularly, the present invention is directed to a method and mechanism of
storing and retrieving data in a computer system and improving performance of
database query language statements.

The quantity of data that must be stored in databases and computer systems
is increasing as many modern businesses and organizations increase their need to
10 access greater amounts of information. A significant portion of the expense for
storing a large quantity of information is related to the costs of purchasing and
maintaining data storage systems. Given this expense, approaches have been
suggested to reduce the amount of space that is needed to store a given quantity of
data.

15 Data compression is a technique used in many modern computer systems to
reduce the storage costs for data. A common approach for implementing
compression is to compress data at the granularity of the file. For example,
traditional compression approaches such as the Unix-based *gzip* or DOS-based *zip*
compress an entire file into a more-compact version of that file. A drawback with
20 this type of approach is that if an entire file is compressed, all or a large part of the
file must be decompressed before any part of it can be used, even if only a small part
of the file is actually needed by a user. This is a problem that particularly exists with
respect to compressing files in database systems, in which a single database file may
contain large quantities of database records, but only a small portion of the
25 individual records may be needed at any moment in time. Thus, the granularity of
compression or decompression may not realistically match the granularity at which
data is desirably used and accessed in the system.

However, compression at other granularities may result in storage
inefficiencies. For example, certain page-at-a-time compression approaches may
30 lead to compressed

pages of different sizes that are inefficiently mapped onto physical pages. In addition, many traditional compression techniques do not even guarantee that data size will not increase after compression.

Moreover, the very acts of compressing and decompressing data could
5 consume an excessive amount of overhead. The overhead is typically related to the specific compression algorithm being used as well as the quantity of data being compressed or decompressed. This overhead could contribute to significant latency when seeking to store, retrieve, or update information in a database system. For example, some compression techniques store compression
10 information separate from the compressed data. Hence, for a simple read access, multiple locations in the database may need to be accessed and expensive decompression operations may need to be performed.

Given the latency problems, as well as less-than-certain compression gains, the trade-off between time and space for compression is not always
15 attractive in a database or other type of computing system. Hence, there is a need for a compression technique that not only results in the reduction of disk space used, but also has no negative impact on the performance of database query language statements against compressed data.

U.S. Patent No. 6,006,232 discloses a table that includes a sub table
20 block, wherein the sub table block includes compressed data and common information associated with the compressed data.

"Compressing relations and indexes" by Goldstein J. et al. discloses a compression algorithm using a page level compression and a file level
25 compression. During the page level compression, values within a range are represented by bits. During the file level compression, objects in a file is partitioned before the file is compressed.

EP 0 079 465 A discloses a method of storing record content, which includes determining a summary list of all data elements from a data base, and mapping the data elements to an index table using the summary list.

The present invention provides a method and mechanism of storing and retrieving data in a compressed format. In one embodiment, data compression is performed on stored data by reducing or eliminating duplicate data values in a data block or other storage unit. In another embodiment, information describing data duplication within the data block is maintained. The present invention also provides a method and mechanism of improving performance of database query language statements. In one embodiment, data duplication information maintained is maintained and used to reduce number of predicate evaluations. In another embodiment, the data duplication information is used to reduce amount of data accessed by a database query language statement.

Further details of aspects, objects, and advantages of the invention are described below in the detailed description, drawings, and claims. Both the foregoing general description and the following detailed description are exemplary and explanatory, and are not intended to be limiting as to the scope of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are included to provide a further understanding of the invention and, together with the Detailed Description, serve to explain the principles of the invention.

5 Fig. 1 shows a data storage structure according to an embodiment of the invention.

Fig. 2 shows a flowchart of an embodiment of a process for storing data.

Fig. 3 shows a flowchart of an embodiment of a process for retrieving data.

10 Fig. 4 illustrates recursive referencing of entries in a symbol table according to an embodiment of the invention.

Fig. 5 illustrates column reordering and recursive referencing of entries in a symbol table according to an embodiment of the invention.

Fig. 6 shows a flowchart of an alternate embodiment of a process for storing data.

15 Fig. 7 illustrates removal of trailing NULL values according to an embodiment of the invention.

Fig. 8 illustrates column reordering and removal of trailing NULL values according to an embodiment of the invention.

20 Fig. 9 shows a process of transforming an uncompressed data block into a compressed data block according to one embodiment of the invention.

Fig. 10 is a flow chart of a method of improving performance of database query language statements according to one embodiment of the invention.

Fig. 11 depicts a process of maintaining data duplication information according to an embodiment of the invention.

25 Fig. 12 illustrates maintaining data duplication information according to one embodiment of the invention.

Fig. 13 shows a flow chart of a method of maintaining data duplication information according to another embodiment of the invention.

Fig. 14 illustrates updating data duplication information according to one embodiment of the invention.

Fig. 15 is a process flow of a method of improving performance of database query language statements according to an embodiment of the invention.

5 Fig. 16 depicts a flow chart of another method of improving performance of database query language statements.

Fig. 17 illustrates an example of how performance of a database query language statement may be improved according to one embodiment of the invention.

10 Fig. 18 shows a process flow of a method of improving performance of database query language statements according to another embodiment of the invention.

Fig. 19 depicts a process of reducing the amount of data accessed by a database query language statement according to an embodiment of the invention.

15 Fig. 20 is an example of how data duplication information may be used to improve performance of database query language statements according to one embodiment of the invention.

Fig. 21 illustrates a flow chart of a method of improving performance of database query language statements according to a further embodiment of the invention.

20 Fig. 22 shows an example of how data duplication information may be used to satisfy a database query language statement according to one embodiment of the invention.

Fig. 23 is a diagram of a computer system with which embodiments of the present invention can be implemented.

Fig. 24 shows an example search tree according to an embodiment of the invention.

25

DETAILED DESCRIPTION

Storing and retrieving data from a data storage system is disclosed. Rather than compressing data at the file level or some other granularity, which may lead to storage inefficiencies and/or increased execution time of database query language statements, data compression is achieved by reducing or eliminating duplicate values in a data block

or other storage unit. Improving performance of database query language statements is also disclosed. By maintaining information describing data duplication within data blocks, which previously had to be recomputed every time a block was accessed, the data duplication information can be used to improve performance of database query language statements.

Database query language statements are often used to create, search, access, retrieve, modify, organize, maintain, manipulate, define, control, add, and/or delete data and structures within databases. One popular database query language is known as the Structured Query Language (SQL). For purposes of explanation only, and not by way of limitation, the following description is made with particular reference to database statements involving SQL.

The term “disk” or “disk system,” which is used throughout this document, refers to data storage systems. The inventive concepts disclosed herein may also be applied to other types of storage systems besides disk-based systems. In addition, the following description will be made with respect to the storage or retrieval of relational data from a database. It is noted, however, that the present invention is applicable to managing other types and granularities of data in a computing system, and thus is not to be limited to compression of relational data.

According to an embodiment of the invention, an identification is made of duplicated values within a set of data that is to be stored within a particular data storage unit. Instead of writing all of these duplicated data values to the data storage unit, a symbol table is created to store one copy of each duplicated data value. Each occurrence of a duplicated data value is configured to reference the corresponding duplicated data value in the symbol table.

As illustrated herein, the data storage units are shown as data blocks or disk pages. However, other data storage units may also be used. In one embodiment, data blocks are self-contained, e.g., the information that is needed to recreate the uncompressed data in a data block is available within the block.

Fig. 1 shows a database table 100 that is to be stored into a block 114 on disk 112. Each row in table 100 includes four columns of information, including an order number column 120, a customer column 122, an item column 124, and a price column 126. Three rows of data are shown in table 100. Row 102a has the value “1251” in

order number column 120, the value "Joe" in customer column 122, the value "Book" in item column 124, and the value "10" in price column 126. Row 104a has the value "1252" in order number column 120, the value "Tom" in customer column 122, the value "Book" in item column 124, and the value "10" in price column 126. Row 106a
5 has the value "1253" in order number column 120, the value "Joe" in customer column 122, the value "Tape" in item column 124, and the value "20" in price column 126.

It is noted that the value "Joe" is repeated in customer column 122 for both rows 102a and 106a. Similarly, the value "Book" is repeated in item column 124 for rows 102a and 104a. The value "10" is repeated in price column 126 for rows 102a and 104a.

10 A symbol table 110 is created to store each duplicated data value. In particular, the duplicated value "Joe" is stored in symbol table entry 130, the duplicated value "10" is stored in symbol table entry 132, and the duplicated value "Book" is stored in symbol table entry 134.

When each row of data in table 100 is stored to block 114 on disk 112, the
15 repeated data values are not again written to disk. Instead, corresponding portions of each row are configured to reference the appropriate symbol table entry in symbol table 110 that contains the correct data value, e.g., using pointer or linking structures.

To illustrate, consider row 102a in table 100. Row 102a includes the following column values: "1251", "Joe", "Book", and "10". It is noted that symbol table 110
20 includes entries for the "Joe", "Book", and "10" values, which are the duplicated data values in table 100 that appear in row 102a.

As row 102a is written to disk, only the unique value "1251" is individually written to disk. This is shown in on-disk row structure 102b, which corresponds to row 102a in table 100. On-disk row structure 102b includes a first portion 142
25 corresponding to order number column 120 in row 102a, which includes the unique data value "1251".

For duplicated data values, pointers or links may be inserted into the appropriate portions of the on-disk row structure to point to the appropriate entry in the symbol table 110. On-disk row structure 102b includes a portion 144 corresponding to customer
30 column 122 in row 102a that is configured to include a pointer 136 to symbol table entry 130, which matches the duplicated data value "Joe" in customer column 122 of row

102a. On-disk row structure 102b also includes a portion 146 corresponding to item column 124 in row 102a that is configured to include a pointer 138 to symbol table entry 134, which matches the duplicated data value "Book" in item column 124 of row 102a. On-disk row structure 102b further includes a portion 148 corresponding to price
5 column 126 in row 102a that is configured to include a pointer 140 to symbol table entry 132, which matches the duplicated data value "10" in price column 126 of row 102a.

Rows 104a and 106a in table 100 are each similarly represented when stored into block 114 on disk 112. Thus, row 104a in table 100 is written to disk as on-disk row structure 104b. The unique data values of row 104a--"1252" and "Tom"--are
10 individually written into on-disk row structure 104b. Row 104a also includes duplicated row values "Book" and "10" in columns 124 and 126, respectively. Portion 150 of on-disk row structure 104b corresponding to column 124 in row 104a is configured to include a pointer 154 to entry 134 in symbol table 110, which matches the duplicated data value "Book". Portion 152 of on-disk row structure 104b corresponding to column
15 126 in row 104a is configured to include a pointer 156 to entry 132 in symbol table 110, which matches the duplicated data value "10".

Row 106a in table 100 is written to disk as on-disk row structure 106b. In row 106a, the unique data values are "1253", "Tape", and "20", which are individually written into on-disk row structure 106b. Row 106a also includes the duplicated row
20 value "Joe" in column 122. Portion 160 of on-disk row structure 106b corresponding to column 122 in row 106a is configured to include a pointer 162 to entry 130 in symbol table 110, which matches the duplicated data value "Joe".

Fig. 2 shows a flowchart of an embodiment of a process for storing data onto disk. At 202, a request is received to store data onto disk. In an embodiment, whether
25 the data is compressed is an option selected by a user or system. If compression is selected, at 204, the data is analyzed to identify repetitive data values for the portions of the data that is to be stored within a given data block. Based upon the identified data duplicates, a symbol table is created that includes some or all of the duplicate data values (206). Each on-disk row structure in the data block is thereafter created. The on-
30 disk row structures are formatted to exclude some or all of the duplicated data values (208). In particular, on-disk row structures that are associated with a duplicated data

value are configured to reference the appropriate entry or entries in the symbol table that stores the relevant data value.

If the compression approach is performed off-line, then the analysis and formatting of rows are performed in batch mode, such that the entire contents of a database block is pre-organized before writing to disk. Off-line processing allows
5 precise determination of the number of rows that will fit into a given data block. Alternatively, the compression approach can be performed on-line, with each on-disk row structure individually prepared and written to disk. In either case, the block metadata is configured to indicate whether the data in the block, or portions of the data
10 in the block, have been compressed. In an embodiment, the header of the block includes a data structure that indicates whether the block or data in the block is compressed. In one embodiment, a single bit is sufficient to indicate whether data in the block is compressed.

In another embodiment, rather than compressing data as it is loaded into a data
15 block, an uncompressed data block may be transformed into a compressed data. As shown in Fig. 9, a data block is analyzed to identify duplicated data values (9002). Based upon the identified data duplicates, a symbol table is created that includes some or all of the duplicated data values (9004). Some or all of the duplicates of the data values in the symbol table are removed or deleted from the data block (9006). On-disk
20 row structures in the data block are then configured to reference the appropriate entry or entries in the symbol table that stores the relevant data value(s) (9008).

If the likelihood of data repetition within a single block is low, the expected compression ratio is also likely to be low. If the likelihood of such repetition is higher, the expected compression ratio will be higher. Thus, rows can be reorganized to
25 increase the likelihood of such repetitions. One way to do this for a single column table is to order table rows by the column value. This can be achieved in the Structured Query Language (SQL) by executing the statement "CREATE TABLE AS SELECT" with an order by clause. This type of technique can also be applied to a table where one column has low cardinality and other columns have high cardinalities, by ordering table
30 rows by low cardinality column.

For a table with multiple low cardinality columns, tuning to achieve a better compression ratio can be performed by first determining the column with the lowest

cardinality. The cardinality of a column can be determined by executing the statement “SELECT COUNT(DISTINCT(c)) from T;”. Once the lowest cardinality column is determined (assume it is column C1), the cardinalities of other columns for fixed values of the chosen column are determined. This can be measured by executing the statement

5 “SELECT SUM(count(*) * COUNT(DISTINCT(cK))) from T group by C1;” for all columns cK other than C1. Assume column C2 has the lowest such measure. The column with the lowest cardinality when the first two columns are fixed is then determined. This can be measured by executing the statement “SELECT SUM(count(*) * COUNT(DISTINCT(cK))) from T group by C1, C2;” for all columns cK other than

10 C1 and C2. By continuing this process, one would determine some column sequence C1, C2, . . . , Cn. This sequence can be used as the sequence of columns in the ORDERED BY clause of the create table compress statement.

Fig. 3 shows a flowchart of an embodiment of a process for retrieving data from disk. At 302, a request is received to retrieve data from disk. The specific block or row

15 that is being sought is identified at 304. Any conventional indexing or search techniques may be employed to perform operation 304. At 306, a determination is made whether the relevant database block is being stored in compressed format. In an embodiment, the block header is accessed to make this determination. If the block is not compressed, then the requested row information is immediately retrieved from disk

20 (310) and returned to the requesting entity (312). If the block is compressed, then the relevant on-disk row structure for the requested row is accessed to retrieve duplicated data values from the symbol table (308). In addition, the individually stored data values from the on-disk row structure is also retrieved (310). At 312, the complete set of requested data is returned to the requesting entity.

25 Recursive symbol entries may be used in the symbol table. A recursive symbol entry is an entry in the symbol table that itself references one or more other symbol table entries. Fig. 4 illustrates the use of recursive symbol entries to format a data block to store table 100 of Fig. 1. Like symbol table 110 of Fig. 1, recursive symbol table 410 includes an entry 430 to store the “Joe” data value, an entry 432 to store the “10” data

30 value, and an entry 434 to store the “Book” data value.

Unlike symbol table 110 of Fig. 1, recursive symbol table 410 includes a recursive symbol table entry 480 that references other entries within symbol table 410.

In particular, recursive symbol table entry 480 includes a first portion 482 that is associated with a link 486 to the data value "Book" in entry 434. Recursive symbol table entry 480 also includes a second portion 484 that is associated with a link 488 to the data value "10" in entry 432. By combining first portion 482 and second portion 484, entry 480 logically creates an entry that includes both referenced data values in sequence, i.e., having the data value of "Book" directly before the data value "10".

One advantage of combining multiple entries in this manner is that there are sequences of column values for rows stored in the block that may match these combinations. Rather than having multiple links or pointer structures between an on-disk row structure and the individual data values in a symbol table, a single link or pointer structure can be used to point to a combined entry that recursively links to multiple other entries.

Referring to table 100 in Fig. 1, columns 124 and 126 in both row 102a and row 104a present the data values "Book" and "10" in sequence. In Fig. 1, on-disk row structure 102b (which corresponds to row 102a) includes a link 138 to the data value "Book" and a separate link 140 to the data value "10". Similarly, on-disk row structure 104b (which corresponds to row 104a) includes a link 154 to the data value "Book" and a separate link 156 to the data value "10". This is a total of four links from these on-disk row structures to entries in symbol table 110.

In the recursive approach of Fig. 4, the on-disk row structure 402b (which corresponds to row 102a) includes a single link 472 to the combined, sequential data values "Book" and "10" represented by recursive symbol table entry 480. This allows the values of both portions 446 and 448 in on-disk row structure 402b (corresponding to columns 124 and 126 in row 102a) to be represented with a single link or pointer structure. Similarly, on-disk row structure 404b (which corresponds to row 104a) includes a single link 476 to the combined, sequential data values "Book" and "10" represented by recursive symbol table entry 480. Again, this allows the values of both portions 450 and 452 in on-disk row structure 404b (corresponding to columns 124 and 126 in row 104a) to be represented with a single link or pointer structure. The end result is that the approach of Fig. 4 requires fewer links or pointers between the on-disk row structure and the symbol table than is required by the approach of Fig. 1.

The approach of Fig. 4 to reduce the number of links or pointers may also be employed with non-recursive combinational entries in symbol table 410. In particular, entry 480 can be configured such that portion 482 explicitly stores the data value “Book”, rather than being associated with a recursive link 486 to entry 434. Portion 484
 5 of entry 480 can also be configured to explicitly store the data value “10”, rather than being associated with a recursive link 488 to entry 432. This approach still allows a reduction in the number of links between the on-disk row structure and symbol table 410, but at the cost of possibly increasing the amount of disk space needed to explicitly store the duplicated data values in symbol table 410. Entries in symbol table 410 may
 10 also include various combinations of recursive links and explicitly stored data values. While the example of Fig. 4 shows only two data values combined into a recursive symbol entry 480, the inventive concepts can be extended to include any number of data values in a combined symbol table entry.

In one embodiment, column reordering may be performed to improve
 15 compression ratios. As noted in the last section, if there are repeating sequences of column values for rows stored in a given block, then the number of links between the symbol table and on-disk row structures can be reduced by creating links to combined entries in the symbol table. Column reordering can be performed to increase the number of duplicated sequences of column values. In an embodiment, column reordering is
 20 performed at the granularity level of the data block. Alternatively, column reordering can be performed at other granularities, e.g., across multiple blocks.

To illustrate, consider database table 600 shown in Fig. 5. Table 600 includes three rows of data, with each row having an Order Number column, a Price column, an Item column, and a Quantity column. Row 602a has the following sequence of data
 25 values: “1254”, “5”, “Book”, and “2”. Row 604a has the following sequence of data values: “1255”, “10”, “Tape”, and “3”. Row 606a has the following sequence of data values: “1256”, “10”, “Movie”, and “3”.

Consider the first situation if columns are not reordered. It is noted that the data values “10” and “3” are duplicated among these rows. Thus, when this table is written
 30 to disk 680, a symbol table 650 can be created having a first entry 652 with the data value “10” and a second entry 654 with the data value “3”. As rows 602a, 604a, and 606a are written to disk, each on-disk row structure for these rows that is associated

with these duplicated data values are configured to reference the appropriate entry in symbol table 650. Thus, on-disk row structure 604b, which corresponds to row 604a, has a portion 641 (corresponding to the data value in the Price column) that is associated with a link 640 to entry 652 in symbol table 650 for the data value "10". On-disk row structure 604b also has a portion 643 (corresponding to the data value in the Quantity column) that is associated with a link 642 to entry 654 in symbol table 650 for the data value "3". On-disk row structure 606b, which corresponds to row 606a, has a portion 645 (corresponding to the data value in the Price column) that is associated with a link 646 to entry 652 in symbol table 650 for the data value "10". On-disk row structure 606b also has a portion 647 (corresponding to the data value in the Quantity column) that is associated with a link 644 to entry 654 in symbol table 650 for the data value "3". Because row 602a does not include any of the duplicated data values, its associated on-disk row structure 602b explicitly stores each column value and does not include any reference to the symbol table.

Consider if column reordering is performed against table 600. The column reordering is performed to increase the possibility of repeating sequences of column values for the rows in the table. In the present example, by switching the ordering between the Item column and the Quantity column in table 600, it can be seen that a repeating sequence of values is created between the Price column and the Quantity column. The result of this column reordering is shown as table 620 in Fig. 5. In particular, both rows 624a and 626a now have the same sequence of data values "10" and "3" for the Price and Quantity columns. Thus, when symbol table 660 is created on disk 682, a combined entry 666 can be inserted that includes, whether directly or indirectly, the data values "10" and "3" in the sequence shown in table 620 for rows 624a and 626a. If combined entry 666 indirectly includes these values, then entry 666 includes a first portion 668 that recursively references entry 662 and a second portion 670 that recursively references entry 664.

When row 624a is written to disk, its on-disk row structure 624b is associated with a single link 632 to combined entry 666 for the sequence of values "10" and "3". This is in contrast to the approach taken by on-disk row structure 604b for the same row without column reordering, which uses two links 640 and 642 to reference the same two data values. Likewise, when row 626a is written to disk, its on-disk row structure 626b

is associated with a single link 636 for the sequence of values “10” and “3”. Again, this is in contrast to the approach taken by on-disk row structure 606b for the same row without column reordering, which uses two links 646 and 644 to reference the same two data values.

5 Fig. 6 shows a flowchart of an embodiment of a process for storing data to disk utilizing column reordering. At 602, a request is received to store data onto disk. If compression is turned on, at 604, the data is analyzed to identify repetitive data values for the portions of the data that is to be stored within a given data block. The data analysis also includes an analysis of possible column reordering combinations that may
10 be performed to increase the likelihood of repeating sequences of data values. A determination is made whether there exists column sequences that may be advantageously obtained through column reordering (606). If so, then the appropriate columns are reordered (608). Based upon the identified data duplicates, a symbol table is created to include some or all of the duplicated data values (610). Each on-disk row
15 structure in the database block is thereafter created and formatted to exclude the duplicated data values (612) and are written to disk (614). In particular, each on-disk row structure that is associated with a duplicated data value is configured to include or reference a link to the appropriate entry or entries in the symbol table that stored the relevant data value. The block metadata is configured to indicate whether the data in the
20 block, or portions of the data in the block, have been compressed. In addition, the block metadata is also configured to indicate whether and how column reordering was performed to the stored data. In an embodiment, the header of the block includes a data structure that indicates whether the block or data in the block are compressed and/or reordered. When the data is retrieved from the block, the header information is accessed
25 to allow the data to be reordered and uncompressed back to its original form.

 An optimization that can be applied is to remove trailing NULL values from the on-disk versions of stored rows. Rows in a database table may include NULL values in the one or more columns at the end of the row. When these rows are written to disk, additional compression can be achieved by not allocating or writing any space to disk
30 for these NULL values. Instead, the storage system is configured to recognize that if the on-disk version of a particular database row does not include one or more stored values

for columns at the end of the row, then those column values are assumed to be NULL values.

To illustrate, consider table 700 in Fig. 7, which is to be stored to block 722 on disk 720. It is noted that rows 702a and 706a both have a NULL value in the last
5 column of the row. When these rows are written to disk 720, the on-disk row structures for these rows are configured to exclude the trailing NULL values. On-disk row structure 702b, which corresponds to row 702a, does not contain a space allocated for the trailing NULL value in the Preferences column. Instead, it only includes allocated space for the leading "1260", "Tom", and "10" values that appear in row 702a. On-disk
10 row structure 706b, which corresponds to row 706a, does not contain a space allocated for the trailing NULL value. Instead, it only includes allocated space for the leading "1262", "Harry", and "30" values that appear in row 706a.

Column reordering can be performed to increase the likelihood of trailing NULL values appearing in rows to be stored on disk. The column values for a set of data is
15 analyzed to sort the columns based upon the number of NULL values in each column. The columns are reordered so that the columns with the larger number of NULL values are ordered to the end of the rows, and the columns with the smaller number of NULL values are ordered closer to the beginning of the rows. Thus, the columns with the largest number of NULL values may be reordered to the end of the table. The column
20 with the next-largest number of NULL values may be reordered as the second-to-last column of the table, with these sequences of actions continued until an efficient ordering of columns is achieved.

To illustrate possible inefficiencies if column reordering is not performed, reference is made to table 800 in Fig. 8. Table 800 includes a set of rows 802a, 804a,
25 and 806a that is to be written to disk. Each row includes one or more NULL values. Rows 802a and 806a includes NULL values in the Style column and Preferences column. Row 804a includes a NULL value in the Style column. Since none of the NULL values in these rows are trailing NULLs, when these rows are written to disk, space may be wasted in each corresponding on-disk row structure because of the NULL
30 values.

As shown, on-disk row structure 802b, which corresponds to row 802a, includes two spaces 810 and 812 that have been allocated for the NULL values corresponding to

the values in the Style and Preferences columns for row 802a. On-disk row structure 804b, which corresponds to row 804a, includes a space 814 that has been allocated for the NULL value corresponding to the value in the Style column of row 804a. On-disk row structure 806b, which corresponds to row 806a, includes two spaces 816 and 818
5 that have been allocated for the NULL values corresponding to the values in the Style and Preferences columns for row 806a.

Column reordering can be performed to increase the compression efficiency of storing table 800 to disk. The first step is to sort the columns to identify specific columns having larger numbers of NULL values. Here, it can be seen that the Style
10 column in table 800 has the largest number of NULL values (with NULL values for every row). The Preferences column has the next largest number of NULL values (with NULL values in rows 802a and 806a). None of the other columns include NULL values.

The columns are reordered to place the columns with the larger number of
15 NULL values closer to the end of the rows. Table 900 illustrates the result of performing this action on table 800. In particular, the columns are reordered such that the Style column, which has the largest number of NULL values, is placed at the end of the rows. The Preferences column, which has the next largest number of NULL values, is placed second-to-last in the sequence of columns. None of the other columns are
20 reordered since they do not contain NULL values.

This type of reordering has maximized the number of trailing NULL values in the table. Now rows 902a and 906a, which correspond to rows 802a and 806a, respectively, in table 800, both include two trailing NULL values. Rows 802a and 806a did not contain any trailing NULL values. Row 904a in table 900, which corresponds to
25 row 804a in table 800, includes one trailing NULL value. Row 804a did not contain any trailing NULL values.

When rows 902a, 904a, and 906a in table 900 are written to disk, the trailing NULL values can be removed from the on-disk versions of each row. Thus, on-disk row structure 902b, which corresponds to row 902a in table 900, excludes the NULL
30 values for the Preferences and Style columns. On-disk row structure 902b only includes three fields to store the data values "1270", "Tom", and "5" for the Order Number, Customer, and Price columns, respectively. In contrast, on-disk row structure 802b,

which is its equivalent prior to column reordering, includes five fields to store the data values “1270”, “Tom”, “NULL”, “NULL”, and “5” for the Order Number, Customer, Style, Preferences, and Price columns, respectively.

On-disk row structure 904b, which corresponds to row 904a in table 900,
5 excludes the NULL value for the Style column. On-disk row structure 904b only includes four fields to store the data values “1270”, “Dick”, “10”, and “Ground Ship” for the Order Number, Customer, Price, and Preferences columns, respectively. In contrast, on-disk row structure 804b, which is its equivalent prior to column reordering, includes five fields to store the data values “1270”, “Dick”, “NULL”, “Ground Ship”,
10 and “10” for the Order Number, Customer, Style, Preferences, and Price columns, respectively.

Like on-disk row structure 902b, on-disk row structure 906b, which corresponds to row 906a in table 900, excludes the NULL values for the Preferences and Style columns. On-disk row structure 906b only includes three field to store the data values
15 for the Order Number, Customer, and Price columns. In contrast, on-disk row structure 806b, which is its equivalent before column reordering, includes five fields to store the data values for the Order Number, Customer, Style, Preferences, and Price columns.

Illustrated in Fig. 10 is a method of improving performance of database query language statements. At 1002, information describing data duplication within a data
20 block is maintained. As discussed above, there are several ways of maintaining data duplication information, e.g., using a symbol table. Data within the data block need not be compressed in order to maintain the data duplication information. A database query language statement with at least one predicate is received against data in the data block at 1004. Number of predicate evaluations are reduced using the data duplication
25 information (1006). By reducing the number of predicate evaluations, performance of the database query language statement is improved.

One way of maintaining data duplication information is shown in Fig. 11. One or more duplicated data values are identified (1102) and the number of occurrences of each duplicated data value is calculated (1104). Each duplicated data value and its
30 corresponding number of occurrences are stored in an entry of a symbol structure (1106). One or more occurrences of at least one of the duplicated data values are eliminated or removed from the data block (1108). Portions of the data block

corresponding to the eliminated occurrences are configured to reference the appropriate symbol structure entry (1110). A link or a pointer may be used to reference the appropriate symbol structure entry.

An example of maintaining data duplication information is depicted in Fig. 12.

5 A data block 1204 on disk 1202 contains four on-disk row structures 1206-1212. Each on-disk row structure corresponds to a row in a table (not shown). Two duplicated data values—"HR" and "LA"—are identified in data block 1204 and stored in symbol structure 1214 along with the number of occurrences of each duplicated data value. In the embodiment, symbol structure 1214 is stored in data block 1204 along with row

10 structures 1206-1212. Data value "HR" occurs twice--once in portion 1216 of row structure 1206 and once in portion 1220 of row structure 1208. Data value "LA" occurs three times—once in portion 1218 of row structure 1206, once in portion 1222 in row structure 1210, and once in portion 1224 of row structure 1212. As shown in Fig. 12, the duplicated data values in data block 1204 are eliminated and portions 1216-1224 are

15 configured to reference the appropriate entry in symbol structure 1214.

Fig. 13 illustrates another way of maintaining data duplication information. One or more duplicated data values are identified at 1302. The number of occurrences of each duplicated data value is calculated (1304). Each duplicated data value and its corresponding number of occurrences are stored in an entry of a symbol structure

20 (1306). The number of occurrences of one of the duplicated data values is updated when an occurrence of the duplicated data value is added or deleted (1308). A symbol structure entry is removed when the number of occurrences of the duplicated data value in the symbol structure entry is zero (1310).

In Fig. 14, information on data duplication within data block 1404 on disk 1402

25 are maintained within symbol structure 1406. In the example, portion 1418 of on-disk row structure 1416 is updated from data value "LA" to data value "SF". Accordingly, entry 1420 in symbol structure 1406, which corresponds to duplicated data value "LA", is updated to reflect the deletion of a data value "LA" from data block 1404.

Shown in Fig. 15 is an embodiment of a method of improving performance of

30 database query language statements. At 1502, information on data duplication is maintained. A database query language statement is received (1504). The database

query language statement includes at least one predicate. The predicate is evaluated on a duplicated data value at most once (1506).

Fig. 16 illustrates a process flow of another method of improving performance of database query language statements. In the embodiment, data duplication information is maintained (1602). A database query language statement with at least one predicate is received (1604) and a data value is accessed (1606). Before evaluating the at least one predicate on the data value, a determination is made as to whether the predicate has previously been evaluated on a duplicate of the data value (1608). If the predicate has not previously been evaluated on a duplicate of the data value, the predicate is evaluated on the data value (1610). If the predicate has previously been evaluated on a duplicate of the data value, result of the previous predicate evaluation is used instead of re-evaluating the predicate on the same data value (1612).

An example of how performance of a database query language statement may be improved is shown in Fig. 17. A database query language statement 1702 is received against data in table 1704. Database query language statement 1702 seeks to find all records in table 1704 that satisfies predicate 1706, i.e., database query language statement 1702 seeks to find all records in table 1704 with data value "LA" in column 1708. Row 1710 is first accessed. Since there has not been any previous predicate evaluations, predicate 1706 is evaluated on data value 1718 in row 1710. The result of the predicate evaluation is stored in a context 1726. Row 1710 is returned as the predicate evaluation result is true.

Row 1712 is then accessed. Before evaluating predicate 1706 on data value 1720 in row 1712, a determination is made as to whether predicate 1706 has previously been evaluated on a duplicate of data value 1720. In one embodiment, context 1726 is checked to determine if there is a previous predicate evaluation result for the data value "DC"—a duplicate of data value 1720. Since the only data value that predicate 1706 has previously been evaluated on is "LA", predicate 1706 is evaluated on data value 1720. The result of the predicate evaluation is stored in context 1726. Row 1712 is not returned as the predicate evaluation result is false.

Next, row 1714 is accessed. Before evaluating predicate 1706 on data value 1722 in row 1714, a determination is made as to whether predicate 1706 has previously been evaluated on a duplicate of data value 1722. Since context 1726 includes a

previous predicate evaluation result for the data value “LA”, which is a duplicate of data value 1722, predicate 1706 is not evaluated on data value 1722. Instead, the previous predicate evaluation result in context 1726 is used. Row 1714 is returned as the previous predicate evaluation result was true. Row 1716 is accessed and processed in similar fashion. Thus, performance of the database query language statement is improved as predicate 1706 was only evaluated on two data values rather than on four data values.

Illustrated in Fig. 18 is a method of improving performance of database query language statements according to one embodiment of the invention. Information on data duplication within a data block is maintained (1802). A database query language statement with at least one predicate is received at 1804. Number of predicate evaluations (1806) and amount of data accessed by the database query language statement (1808) are reduced using the data duplication information.

Fig. 19 depicts one way of reducing amount of data accessed by a database query language statement. At 1902, total number of data values on which the predicate will evaluate to true is calculated. A counter is then set equal to the calculated total at 1904. A data value is accessed (1906) and the predicate is evaluated on the data value (1908). If the result of the predicate evaluation is false, then the process returns and accesses another data value at 1906. If the result of the predicate evaluation is true, then the counter is reduced by one (1910). At 1912, a determination is made as to whether the counter is now equal to zero. If the counter is not equal to zero, then the process returns to 1906 and accesses another data value. If the counter is equal to zero, the process ends.

An example of how data duplication information may be used to improve performance of database query language statements is shown in Fig. 20. In Fig. 20, a database query language statement 2002 is received against data in table 2004, which is stored in data block 2006. Database query language statement 2002 seeks to find and return all rows of table 2004 that includes data value “LA” in Office column 2016. Table 2004 includes ten rows and 4 columns—ID column 2010, Name column 2012, Department column 2014, and Office column 2016. There are no duplicated data values in either ID column 2010 or Name column 2012. Department column 2014 and Office column 2016 each have three duplicated data values.

In the embodiment of Fig. 20, data duplication information is stored in a symbol table 2008. Symbol table 2008 includes a copy of each duplicated data value in table 2004 along with the corresponding number of occurrences of the duplicated data value. Each entry in symbol table 2008 also includes an entry number that can be used in data block 2006 to reference the symbol table entry. As illustrated in Fig. 20, each occurrence of a duplicated data value in data block 2006 is replaced with the corresponding symbol table entry number rather than a pointer or a link to the corresponding symbol table entry.

To evaluate database query language statement 2002, row structure 2018b in data block 2006, which corresponds to row 2018a in table 2004, is accessed. Since portion 2020 of row structure 2018b, which corresponds to column 2016 of row 2018a, references entry number #3 in symbol table 2008, duplicated data value "LA" is retrieved from symbol table 2008 and evaluated against the predicate in database query language statement 2002. The result of the predicate evaluation is stored in a context 2024. A counter 2026 is then set to "3", which is equal to the number of occurrences of duplicated data value "LA" in table 2004, since "LA" is the only data value in which the predicate will evaluate to true. Row 2018a of table 2004 is then returned and counter 2026 reduced by 1. When counter 2026 becomes zero, sequential row access of data block 2006 can end since the maximum number of rows that will satisfy the database query language statement will have been returned.

Next, row structure 2028b in data block 2006, which corresponds to row 2028a in table 2004, is accessed. Portion 2030 of row structure 2028b, which corresponds to column 2016 of row 2028a, references entry number #4 in symbol table 2008. Before duplicated data value "DC" is retrieved from symbol table 2008 and evaluated against the predicate in database query language statement 2002, context 2024 is checked to determine whether there is a previous predicate evaluation result for entry number #4. Since context 2024 does not contain a result of a previous predicate evaluation for entry number #4, duplicated data value "DC" is retrieved from symbol table 2008 and evaluated against the predicate in database query language statement 2002. The result of the predicate evaluation is then stored in context 2024. Row 2028a is not returned and counter 2026 is not reduced because the result of the predicate evaluation is false.

Row structure 2032b in data block 2006, which corresponds to row 2032a in table 2004, is accessed. Portion 2034 of row structure 2032b, which corresponds to column 2016 of row 2032a, references entry number #3 in symbol table 2008. Since context 2024 already contains a result of a previous predicate evaluation for entry
5 number #3, the predicate need not be evaluated again. Given that the result of the previous predicate evaluation for entry number #3 is true, row 2032a is returned and counter 2026 is reduced by 1.

Row structure 2036b in data block 2006, which corresponds to row 2036a in table 2004, is accessed in much the same way as row structure 2032b. Like portion
10 2034 of row structure 2032b, portion 2038 of row structure 2036b, which corresponds to column 2016 of row 2036a, references entry number #3 in symbol table 2008. Since context 2024 already contains a result of a previous predicate evaluation for entry number #3, the predicate need not be evaluated again. Given that the result of the previous predicate evaluation for entry number #3 is true, row 2036a is returned and
15 counter 2026 is again reduced by 1. Sequential row access of data block 2006 then ends as all rows that will satisfy database query language statement 2002 has been returned.

In the example shown in Fig. 20, not only was the number of predicate evaluations reduced from four to two, but the number of rows accessed was also reduced from ten to four as a result of having the data duplication information. Thus,
20 performance of database query language statements may be greatly improved by maintaining information on data duplication within a data block.

Since only one unique data value could satisfy the predicate of database query language statement 2002 in Fig 20, the total number of data values in the data block on which the predicate will evaluate to true is calculated by simply looking at symbol table
25 2008. For a database query language statement where more than one unique data value could satisfy the predicate, the total number of data values in the data block on which the predicate will evaluate to true is calculated by totaling the number of occurrences of each unique data value in the data block that will satisfy the predicate. For example, if three different unique data values—X, Y, and Z--could satisfy the predicate and X
30 occurs five times in the data block, Y occurs eight times in the data block, and Z occurs three times in the data block, the total number of data values in the data block on which the predicate will evaluate to true is equal to five plus eight plus three, which is sixteen.

Hence, counter 2026 in Fig. 20 would be set to sixteen and reduced each time a satisfying data value is returned. Therefore, the amount of data accessed to satisfy a database query language statement may be significantly reduced.

Fig. 21 illustrates a process flow of another method of improving performance of database query language statements. Data duplication information within a data block is maintained (2102). A database query language statement with at least one predicate is received at 2104. The number of predicate evaluations is reduced using the data duplication information (2106). In addition, rather than accessing data values to satisfy the database query language statement, the data duplication information is updated instead (2108).

Shown in Fig. 22 is an example of how data duplication information may be updated to satisfy a database query language statement instead of accessing data values in a data block. A database query language statement 2202 is received against table 2204, which is stored in data block 2206 of disk 2208. Database query language statement 2202 seeks to change data values "LA" in Office column 2210 of table 2204 from "LA" to "SF". Since information on data duplication is already being maintained in symbol structure 2212, as described above with respect to Fig. 12, only entry 2214 in symbol structure 2212 need to be updated from "LA" to "SF" to satisfy database query language statement 2202. Thus, performance of database query language statements may be greatly improved when data duplication information is available.

SYSTEM ARCHITECTURE OVERVIEW

Fig. 23 is a block diagram that illustrates a computer system 2300 upon which a method of improving performance of database query language statements may be implemented. Computer system 2300 includes a bus 2302 or other communication mechanisms for communicating information, and a processor 2304 couple with bus 2302 for processing information. Computer system 2300 also includes a main memory 2306, such as a random access memory (RAM) or other dynamic storage device, couple to bus 2302 for storing information and instructions to be executed by processor 2304. Main memory 2306 also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor 2304. Computer system 2300 further includes read only memory (ROM) 2308 or other static storage device couple to bus 2302 for storing static information and instructions

for processor 2304. A storage device 2310, such as a magnetic disk or optical disk, is provided and coupled to bus 2302 for storing information and instructions.

Computer system 2300 may be coupled via bus 2302 to a display 2312, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device
5 2314, including alphanumeric and other keys, is coupled to bus 2302 for communicating information and command selections to processor 2304. Another type of user input device is cursor control 2316, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor 2304 and for controlling cursor movement on display 2312. This input device typically has two
10 degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

In one embodiment computer system 2300 is used to improve performance of database query language statements. According to one embodiment, such use is provided by computer system 2300 in response to processor 2304 executing one or more
15 sequences of one or more instructions contained in main memory 2306. Such instructions may be read into main memory 2306 from another computer-readable medium, such as storage device 2310. Execution of the sequences of instructions contained in main memory 2306 causes processor 2304 to perform the process steps described herein. One or more processors in a multi-processing arrangement may also
20 be employed to execute the sequences of instructions contained in main memory 2306. In other embodiments, hard-wired circuitry may be used in place of or in combination with software instructions. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

The term "computer-readable medium" as used herein refers to any medium that
25 participates in providing instructions to processor 2304 for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device 2310. Volatile media includes dynamic memory, such as main memory 2306. Transmission media includes coaxial cables, copper wire and fiber
30 optics, including the wires that comprise bus 2302. Transmission media can also take the form of acoustic or light waves, such as those generated during radio wave and infrared data communications.

Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punch cards, paper tape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other
5 memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor 2304 for execution. For example, the instructions may initially be carried on a magnetic disk of a remote
10 computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system 2300 can receive the data on the telephone line and use an infrared transmitter to convert the data to an infrared signal.

An infrared detector coupled to bus 2302 can receive the data carried in the
15 infrared signal and place the data on bus 2302. Bus 2302 carries the data to main memory 2306, from which processor 2304 retrieves and executes the instructions. The instructions received by main memory 2306 may optionally be stored on storage device 2310 either before or after execution by processor 2304.

Computer system 2300 also includes a communication interface 2318 coupled to
20 bus 2302. Communication interface 2318 provides a two-way data communication coupling to a network link 2320 that is connected to a local network 2322. For example, communication interface 2318 may be an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface 2318 may be a local
25 area network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface 2318 sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

Network link 2320 typically provides data communication through one or more
30 networks to other data devices. For example, network link 2320 may provide a connection through local network 2322 to a host computer 2324 or to data equipment operated by an Internet Service Provider (ISP) 2326. ISP 2326 in turn provides data

communication services through the world wide packet data communication network now commonly referred to as the "Internet" 2328. Local network 2322 and Internet 2328 both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link 2320 and through communication interface 2318, which carry the digital data to and from computer system 2300, are exemplary forms of carrier waves transporting the information.

Computer system 2300 can send messages and receive data, including program code, through the network(s), network link 2320 and communication interface 2318. In the Internet example, a server 2330 might transmit a requested code for an application program through Internet 2328, ISP 2326, local network 2322 and communication interface 2318. In accordance with the invention, one such downloaded application provides for managing, storing, and retrieving data from a storage system containing multiple data storage devices. The received code may be executed by processor 2304 as it is received, and/or stored in storage device 2310, or other non-volatile storage for later execution. In this manner, computer system 2300 may obtain application code in the form of a carrier wave.

ILLUSTRATIVE EMBODIMENT

The following provides implementation details for an embodiment of the invention. In this embodiment, a block data format is employed that allows for various data compression schemes. The data block format can be applied as an extension to existing block formats for existing systems. The data block format defines how row pieces are stored in data blocks. The block header includes fields that indicate whether any or all of the following features are used in a particular block:

- block level compression
- block level column reordering
- intra-row run-length NULL compression

Column Reordering

If the column reordering indicator is set, the block uses block-level column reordering. Having a set of row pieces N columns each, a block-level permutation of

the column numbers provides column reordering. If a block uses column reordering, two arrays can be stored directly after the block header. The first array stores the lengths of leading columns (ub2 Lens[fixed_width_column_count]). The first
 5 fixed_width_column_count ≥ 0 columns in a block in the present embodiment are fixed length columns within the block. In this embodiment, the value of fixed_width_column_count can be zero. The second array stores column number permutation for the block (ub1 Perms[block-level_column_count]).

Column reordering provides at least two advantages. First, column reordering allows higher compression by grouping appropriate columns together. Second, column
 10 reordering improve Nth column access by moving fixed-length columns in front of variable-length columns. If the first set of columns have fixed length within the block, these lengths can be stored once in the block instead of in each row. These lengths can be stored in the length array. These arrays can be considered part of block header for space computations purposes.

15 Another advantage of column reordering is that if a column is NULL in all row pieces in a block, then it is can be effectively "compressed out" from all row pieces because a NULL column does not have any data and its length can be stored in the length array. Moving block-level fixed column lengths from row pieces to a block-level array may provide relatively cheap compression, which can be particularly effective if
 20 column values are short, which occur if the data is highly numeric.

In one embodiment, rows that are column reordered should have equal column counts and the corresponding columns should have equal lengths. However, it is not necessary in all embodiments that all row pieces in a given block have these properties, since Insert and Update operations that violate this requirement may become inefficient
 25 under this approach. Therefore, in alternate embodiment, row pieces are allowed in a column-reordered block that do not satisfy the reordering requirement. For column-reordered block, these row pieces are excluded from reordering. These row pieces are not excluded from compression if the block is both reordered and compressed. Therefore, rows are allowed that only partially satisfies the fixed length requirement.
 30 Each row in column-reordered block is associated with a designated field identifying how many leading columns in the row have lengths obeying the block-level column length array parameter. In one embodiment, column reordering does not apply to

forwarding row pieces and deleted row pieces. Column reordering also does not apply to row pieces that are explicitly excluded from reordering.

The variable part of the new data layer header includes permutations stored in the perms array and fixed width column lengths stored in the lens array. For a block
 5 with N column rows having M fixed length columns, the variable part takes $\text{sizeof}(\text{ub1}) \times N + \text{sizeof}(\text{ub2}) \times M$ bytes. Assuming $\text{sizeof}(\text{ub1}) \times N + \text{sizeof}(\text{ub2}) \times M \leq 3 \times N$ the space taken (referred to as $\text{VARSPACE}(M, N)$) does not exceed $3 \times N$. By not storing fixed width column lengths inside row pieces, at least M bytes per row can be saved. For a block with K column-reordered rows the savings can be at least $(M - 1) \times K$ bytes
 10 $(M - 1$ because there is a per-row piece fixed length column count in a ub1). In an embodiment, the break-even point is at $K = 5$, with additional saving occurring for $K > 5$.

In one embodiment, the process should be configured such that column-reordered data does not consume more space than uncompressed data.

15 As noted above, the first array lens ($\text{ub2 lens}[\text{fixed_width_column_count}]$) contains fixed column lengths for the block for the first n columns (where $n = \text{fixed_width_column_count}$). Instead of storing column lengths in each element of the array, prefix sums can be stored for the column lengths using a non-commutative summation to work around possible NULL values, as follows:

20 $\text{lens}[0] \leftarrow \text{SIZEOF}(\text{column}[0]),$
 $\text{lens}[k] \leftarrow \text{lens}[k - 1] \oplus \text{SIZEOF}(\text{column}[k]), k > 0$

where $\text{SIZEOF}(\text{col}) \equiv 0x8000$ if the column is NULL, and the summation $\oplus$ operation carries 0x8000 bit from the right operand only, i.e. $\oplus$ is not commutative.

For example, consider a sequence of 10 fixed width column lengths as follows:

25 5, 5, 12, 7, NULL, NULL, 4, NULL, 4, 6,

the array will contain the following values:

5, 10, 22, 29, 32797, 32797, 33, 32801, 37, 43.

Extracting Nth column length from this array is performed as follows. First, for $k < \text{fixed_width_column_count}$, $\text{lens}[k] \& 0x8000$ is tested to detect if the column k is
 30 NULL. If it is not NULL, then its length is:

$$\text{len} \leftarrow \text{lens}[k] - (\text{lens}[k - 1] \& 0x7FFF), k > 0$$

or

$$\text{len} \leftarrow \text{lens}[0], k = 0$$

If there are many fixed width columns in a block, then the array can consume a significant amount of space. There are potential advantages for using fixed length column reordering, e.g., to compress out long sequences of NULL columns by rotating globally NULL columns to the end and subsequently performing truncation of trailing NULL values. In general, the more fixed length columns in a block, the higher the probability that several columns will have the same column lengths.

10 Block Formats

If the row compression indicator is set in the block header, then this indicates that all or part of the rows in the block is compressed. For only the purpose of the following explanation, assume that a given block is column reordered, the block's columns 0 through (fixed_width_column_count - 1) are of fixed length and the rest of the columns are of variable length. Further assume that columns 0 through (fixed_width_column_count - 1) have their lengths removed and stored in the block-level array lens. The rest of the columns have their lengths stored within the row itself. In an embodiment, the following are defined to the row piece formats for the block:

```
#define SHORT_INDEX 200
#define SHORT_LENGTH 250
#define LONG_INDEX 251
#define LONG_LENGTH SHORT_LENGTH
```

The value of SHORT_INDEX is an index into the symbol table. For values $L \in [\text{SHORT_INDEX}, \text{SHORT_LENGTH})$, $\text{len} \leftarrow (L - \text{KDRCS SHORT INDEX})$ is the length of an in-place stored column (i.e., uncompressed column). The value of LONG_INDEX indicates that the next 2 bytes contain an index value into the symbol table. The value of LONG_LENGTH indicates that an uncompressed column (with length bytes first) follows. The column count field in the row header contains count of column length first bytes present in the row.

In one embodiment, a symbol table is table[0] with rows in data layer sense. Rows in the symbol table do not have flag bytes or lock bytes. For the flag byte, locking of symbol table entries can be implemented, e.g., by obtaining data block locks. In one embodiment, concurrency loss can be incurred for symbol table operations if operations on the symbol table are not unduly frequent. Flag and lock bytes of symbol table rows can contain a reference count stored as ub2. In an embodiment, symbol table rows are similar to row piece formats, except for the following constants used to partition first length byte values into ranges:

```
#define SYMTAB_SHORT_INDEX 200
10  #define SYMTAB_SHORT_LENGTH 250
    #define SYMTAB_LONG_INDEX 251
    #define SYMTAB_LONG_LENGTH SYMTAB_SHORT_LENGTH
```

Interpretation for the first length bytes is the same as for row pieces for blocks as explained above. In addition, multiple column entries are allowed in the symbol table.

15 In an embodiment, compression of prefix, fixed length columns differs from compression of other columns. The difference occurs because the lengths are removed from these columns. The following can be performed for compression of prefix fixed length columns by using stop bytes Bi. The first byte B0 of a row piece data is an index into the symbol table, if $B0 < \text{SHORT_INDEX}$. Otherwise, $(B0 - \text{SHORT_INDEX})$ is the number of in-place stored (i.e. not compressed) fixed length columns that follow. 20 The next byte B1 either directly follows B0, or directly follows the data for pre-ceding $(B0 - \text{SHORT_INDEX})$ columns correspondingly. This definition continues recursively while the column(s) that follow Bk are still fixed length.

Compressor

25 In an embodiment, at least two different approaches to compression are available. Online compression will form compressed blocks incrementally. Offline compression buffers sufficient number of rows into buffers first, and then places as many rows as practicable into a compressed block. Online compression is attractive because it does not have to maintain additional buffers or perform additional memcopy operations. However, under certain circumstances, online compression could be very 30 inferior to offline compression in terms of compression quality. For example, forming

blocks incrementally could prevent complete pre-analysis of columns before processing, which could negatively impact the ability to efficiently perform block-level column reordering. Trying to work-around these limitations by performing block reorganizations (re-compressions) after a large portion of the block has already been
 5 formed could be excessively expensive.

The following description is with respect to offline compression, in which a bulk set of rows are processed to format one or more full data blocks. Each data column can be described as a pointer to its data in a buffer, e.g., in a source table's data blocks.

The present embodiment of the compressor is described using the following

10 Terms:

- SingleEntry (SE) is a representation of a column that has a pointer to data, data length and a few reference counts.

- RowModel (RM) is a representation of a row as an array of pointers to SE and a count of SE (column count).

15 - RowSetModel (RSM) is a representation of a set of rows as an array of pointers to RM and row count.

RSM Construction

As the Compressor module buffers rows for offline compression, it builds an RSM for the set of rows. Each row column is looked up in a hash table, and if not
 20 present, a new SE is allocated for the column, otherwise a pointer to an existing SE is used. Dynamic memory managers can be used to reduce fragmentation and memory allocation overheads. In one embodiment, the memory managers are for per-block temporary allocations that persist only for compression of a single block and per-segment memory that persists for the duration of segment compression.

25 RSM represents a set of rows as a matrix of SE pointers, where equal columns point to the same SEs. Each SE is a candidate to become a single column symbol table entry. Once the RSM is built, the symbol table can be generated consisting of single column entries based on SE reference counts. For some data sets, this will provide efficient compression as repeated SEs will be encoded through small byte indexes into

the symbol table. To achieve higher compression, column sequences can be replaced with indexes into the symbol table and create multiple-column symbol table entries.

MultiEntry (ME) is a representation of a multiple-column symbol table entry candidate. It is described using a (row, column) pair of its first column, column count a
 5 few more fields. In the present embodiment, the act of compression is related to the act of identifying a set of useful MEs for a given RSM.

Several steps can be taken to perform block-level column reordering computation. In a first step (Step (A)), an empirical rule can be used to compute block-level column reordering: order columns by (column cardinality, column length
 10 cardinality). The compressor determines block-level column reordering once per RSM using this rule. In one embodiment, the following two columns c1 and c2 are considered either equal $c1 \equiv c2$ by this ordering rule, or column c1 can alternatively be considered more acceptable than c2:

	1 1
15	1 2
	1 1
	1 2
	1 1
	1 2
20	1 1
	1 2
	1 1
	1 2
	1 1
25	2 2

Once step (A) is done, dictionary build-up can be performed (step (C)). This step involves identifying { ME } for RSM. An RSM can be thought of as a text over an alphabet consisting of pointers to SEs. For 8k blocks in an embodiment, the lengths of the text are approximately within 5000-10000 symbols (200 - 400 rows 25 columns

each) and the number of distinct symbols per block is approximately within 700-900. The problem of identifying { ME } can therefore be thought of as a substitutional compression problem over an integer alphabet with alphabet size about 3 times larger than regular 1-byte alphabet ($256 \cdot 3 \cdot 2 [700, 900]$).

5 In an embodiment, the Compressor uses an extra step (B) between steps (A) and (C), and is also referred to herein as the “Prefix Detection” step. This step captures distinctive properties of the data set being compressed, e.g., the property of the data set including a set of rows. A distinctive property of the data set is that it is a set of rows, where columns are ordered by column cardinality in step (A).

10 An N-ary digital compressed search tree can be built for the set of rows $\{ r_i \}_{i=1}^n$. This search tree has exactly N leaves for N rows, a root node and several branch nodes. The tree can be described by the following tree construction procedure:

- An empty tree contains a root node only.
- After adding the first row r_1 to the tree, the tree contains root node, one leaf
15 node and one edge connecting the root with the leaf. The edge is labeled with all symbols in r_1 .

- When adding row r_2 to the tree, add one leaf node.

If the first symbol in r_2 is the same as the first symbol in r_1 , delete the old edge, add a new branch node to the tree, connect root with the new branch and connect new
20 branch node with both leaves. Label the edge from the root to the new branch node with the longest common prefix of rows r_1 and r_2 . Label edges connecting new branch node with leaves with suffixes of rows r_1 and r_2 correspondingly.

If the first symbol in r_2 is not the same as the first symbol in r_1 , simply add a new edge from the root to the new leaf, labeling it with all symbols from r_2 . Tree
25 construction proceeds recursively for all rows in RSM.

As an example, suppose that 4 rows are added to the tree:

ABCDEFGHJKLMNOPQRSTUVWXYZ

ABCDEFGHJKLMNOP1QRSTUVWXYZ

1BCDEFGHJKLMNOPQRSTUVWXYZ

ABCDEFGHIJKLMNQRSTAAAAAA

The tree will have two edges coming from the root node, one of them connecting root with leaf node for r_3 and labeled with all symbols in r_3 , the other edge connecting root with branch node, edge labeled with symbols

5 ABCDEFGHIJKLMNO

The two edges coming out of this branch node will be labeled with

1QRSTUVWXYZ

and

PQRST

10 correspondingly. The first edge will connect the branch with leaf for row r_2 , the second edge will connect the branch with another branch. Finally that another branch will have two edges connecting it with leaves for r_1 and r_4 , labeled

UVWXYZ

and

15 AAAAAA

correspondingly. Fig. 24 shows an example illustration of this tree.

The tree has certain properties:

- Each branch node has at least two outgoing edges
- The tree has at most $2 \cdot N$ nodes and $2 \cdot N$ edges
- 20 • Catenation of edge labels from root to leaf gives the corresponding leaf's row symbols
- If two rows share a common prefix, then paths from the root to their leaves share common edges with cumulative length equal to the maximal common prefix length.
- 25 • There is a 1-1 correspondence between branch nodes and prefixes shared by at least two rows.

If each branch node of the tree is labeled with the number of leaf nodes in the subtree rooted in this node and cumulative edge length from the root to this node, then

for each branch node we can immediately tell how many rows share the prefix that ends in this node. Once the tree is built, the labeling is possible in a single tree depth-first traversal, which has $O(N)$ complexity in the number of rows N .

Once the tree is built and labeled, proceed with the prefix detection. Prefix
5 detection is based on a simple recursive procedure. Consider the following example for 20 rows:

	ABCDEFGHIJKLMNOPac
	ABCDEFGHIJKLMNOPad
	ABCDEFGHIJKLMNOPae
10	ABCDEFGHIJKLMNOPaf
	ABCDEFGHIJKLMNOPag
	ABCDEFGHIJKLMNOPah
	ABCDEFGHIJKLMNOPai
	ABCDEFGHIJKLMNOPaj
15	ABCDEFGHIJKLMNOPak
	ABCDEFGHIJKLMNOPal
	ABCDEFGHIJKLMNOPbm
	ABCDEFGHIJKLMNOPbn
	ABCDEFGHIJKLMNOPbo
20	ABCDEFGHIJKLMNOPbp
	ABCDEFGHIJKLMNOPbq
	ABCDEFGHIJKLMNOPbr
	ABCDEFGHIJKLMNOPbs
	ABCDEFGHIJKLMNOPbt
25	ABCDEFGHIJKLMNOPbu
	ABCDEFGHIJKLMNOPbv

the tree will have root node n_0 with one edge (n_0, n_1) coming out of it, labeled with

ABCDEFGHIJKLMN

node n_1 will have two edges (n_1, n_2) and (n_1, n_3) coming out of it, edge (n_1, n_2) labeled with

a

5 and edge (n_1, n_3) labeled with

b

Nodes n_2 and n_3 will be labeled with leaf node counts of 10 and their distance from root is 17. Node n_1 will be labeled with leaf node count of 20 and distance from root of 16.

10 For this example, the following choices can be presented in construction of { ME } :

Choice 1: { ME } = $\emptyset$ (no MEs in the symbol table)

Choice 2: { ME } = { {ABCDEFGHIJKLMN} } (one ME)

Choice 3: { ME } = { {ABCDEFGHIJKLMNa} ,
15 { ABCDEFGHIJKLMNb } } (two MEs)

Choice 4: { ME } = { {ABCDEFGHIJKLMN} , { #0a } , { #0b } } (three MEs)

Another tree traversal is performed to help to choose { ME }. The algorithm has a common step. At each step, one branch node is a current node. The common step
20 consists of choosing from the following alternatives:

Alternative 1: Do not create additional MEs for which the current node is a proper prefix.

Alternative 2: Create additional MEs for which the current node is a proper prefix; replicate the prefix that ends in the current node into these MEs.

25 Alternative 3: Create additional MEs for which the current node is a proper prefix; factor the prefix that ends in the current node into a separate ME and make additional MEs point to it.

Note that for the above 20-row example,

Choice 1: Is given by Alternative 1 when the current node is n_0 ;

Choice 2: Is given by Alternative 2 when the current node is n_0 and Alternative 1 when the current node is n_1

Choice 3: Is given by Alternative 2 when the current node is n_0 and Alternative 2
5 when the current node is n_1

Choice 4: Is given by Alternative 2 when the current node is n_0 and Alternative 3 when the current node is n_1

Therefore, the Prefix Detection algorithm traverses the tree, chooses from the above mentioned 3 alternatives at the current branch nodes by considering

- 10
- labels in the current node;
 - labels in all adjacent nodes in the current node's subtree
 - non-zero cost of creating new symbol table entries

and then proceeds into the subtrees. When this tree traversal reaches a leaf node, it "knows" if any prefix of the corresponding row compresses or not, and, if it compresses,
15 which ME it compresses with. Thus, in one approach, the Prefix Detection finds all prefix MEs from optimal set of { ME } for a given block; or finds no MEs if the optimal set is empty or does not include any prefix MEs.

In an embodiment, the compressor operates by first loading a standard data block and an RSM for it parallel. After that, the compressor compresses the RSM by trying to
20 create a new compressed block. If rows from the standard block do not fit into a new compressed block (i.e. no compression), the compressor writes a standard block and proceeds onto the next set of rows. Otherwise, an estimate is made of compression gain by comparing free space in the standard block to free space in the compressed block. If there is no or minimal compression gain, then the standard block is written and the
25 compressor proceeds onto the next set of rows. Otherwise, if the compression gain is substantial, then estimate the number of rows that will fit into the block through interpolation and switch into buffering mode. In the buffering mode, build RSM without loading any data into blocks. Once the estimated number of rows has been buffered into RSM, try to compress the RSM again. Three possible outcomes are:

- 30
- RSM does not fit into a single compressed block.

- RSM fits into a single compressed block and free space wastage is minimal.
- RSM fits into a single compressed block and there is still free space to load more rows.

In the first case, start reducing the number of rows in RSM that the
 5 compressor tries to load into a single block by repeatedly bisecting [known2fit,
 known2notfit] interval with the start condition known2fit = number of rows in
 uncompressed block, known2notfit = number of rows that we estimated but that did
 not fit. In the second case write a compressed block and proceed to the next set of
 rows. In the third case, make a new estimation for the number of rows and continue
 10 buffering the RSM.

Each individual block compression involves executing step (A), step (B),
 substituting detected prefix in RSM and feeding the resulting RSM to step (C).

Decompressor

The present invention allows very fast, low overhead decompression of
 15 compressed data. In an embodiment, column values/lengths are readily available in
 the block. Therefore, the decompression process is a matter of locating these values
 by interpreting block header bits, fixed column and column permutation arrays if
 present, and first length bytes in the row and symbol table entries this row
 references. These referenced data values can be returned without performing any
 20 costly decompression or reconstruction algorithms.

In the foregoing specification, the invention has been described with
 reference to specific embodiments thereof. It will, however, be evident that various
 modifications and changes may be made thereto. Accordingly, the scope of the
 claims should not be limited by the preferred embodiments set forth in the examples,
 25 but should be given the broadest interpretation consistent with the description as a
 whole. For example, the above-described process flows are described with reference
 to a particular ordering of process actions. However, the ordering of many of the
 described process actions may be changed without affecting the scope or operation
 of the invention. The specification and drawings are, accordingly, to be regarded in
 30 an illustrative rather than restrictive sense.

CLAIMS:

1. A method for storing data in a compressed format, comprising:
 receiving a request to store data onto one or more database blocks on disk,
 the data including one or more data items, in which a database block on disk
 5 comprises a storage unit on a storage device for a database, and each of the one or
 more data items comprises an item of data in the database;
 reordering the data to change an order of two or more columns of the data;
 analyzing the data to determine the existence of a redundant data item in the
 data, in which the redundant data item is determined to exist when a duplicate of the
 10 redundant item is identified in the one or more database blocks; and
 formatting an on-disk data structure corresponding to the data;
 characterized in that the method further comprises:
 creating a symbol structure in a non-transitory computer accessible
 storage medium or in the storage device to store the redundant data item, wherein
 15 the symbol structure comprises a data structure for storing the redundant data item;
 and
 associating the on-disk data structure with the redundant data item
 stored in the symbol structure by including a link to the redundant data item in the
 symbol structure for at least one entry in the on-disk data structure.
 20
2. The method of claim 1, in which a link or pointer, rather than the redundant
 data item, is stored in the on-disk data structure to reference the redundant data item
 stored in the symbol structure.
3. The method of claim 1, wherein the data comprises relational data having one
 25 or more rows.
4. The method of claim 3, wherein each row is processed online.
5. The method of claim 3, in which a group of rows are processed offline.
6. The method of claim 1, in which the data is reorganized to increase the
 likelihood of redundant data items.

7. The method of claim 6, in which the cardinality of the data is considered for reorganization.
8. The method of claim 1, in which the symbol structure comprises an entry that recursively references another entry in the symbol structure.
- 5 9. The method of claim 8, in which the entry references multiple other entries in the symbol structure.
- 10 10. The method of claim 8, in which the entry comprises both the stored redundant data item and the entry that recursively references another entry in the symbol structure.
- 10 11. The method of claim 8, comprising associating a second on-disk data structure that comprises a first data structure stored in the non-transitory computer accessible storage medium or in the storage device with the entry that recursively references the another entry in the symbol structure.
- 15 12. The method of claim 8, further comprising a second on-disk data structure in which multiple fields within the on-disk data structure are collectively associated with a reference with the entry that recursively references another entry in the symbol structure.
13. The method of claim 12, in which data columns are reordered.
- 20 14. The method of claim 13, in which the data columns are reordered to increase likelihood of repeating sequences for the multiple fields within the on-disk data structure.
15. The method of claim 12, in which column reordering is performed at the level of the database block.
- 25 16. The method of claim 13, in which a trailing NULL value is not included in the on-disk data structure.

17. The-method of claim 1, in which the database block comprises metadata indicating whether compression is being applied.
18. The method of claim 17, in which the metadata comprises a single bit.
19. The method of claim 1, in which the database block comprises metadata
5 indicating whether columns are being reordered.
20. A computer program product that includes a computer readable medium, the computer readable medium having stored thereon a sequence of instructions which, when executed by a processor, causes the processor to perform any of the methods of claims 1 to 19.
- 10 21. A system for storing data in a compressed format, the system comprising means for performing any of the methods of claims 1 to 19.

1/19

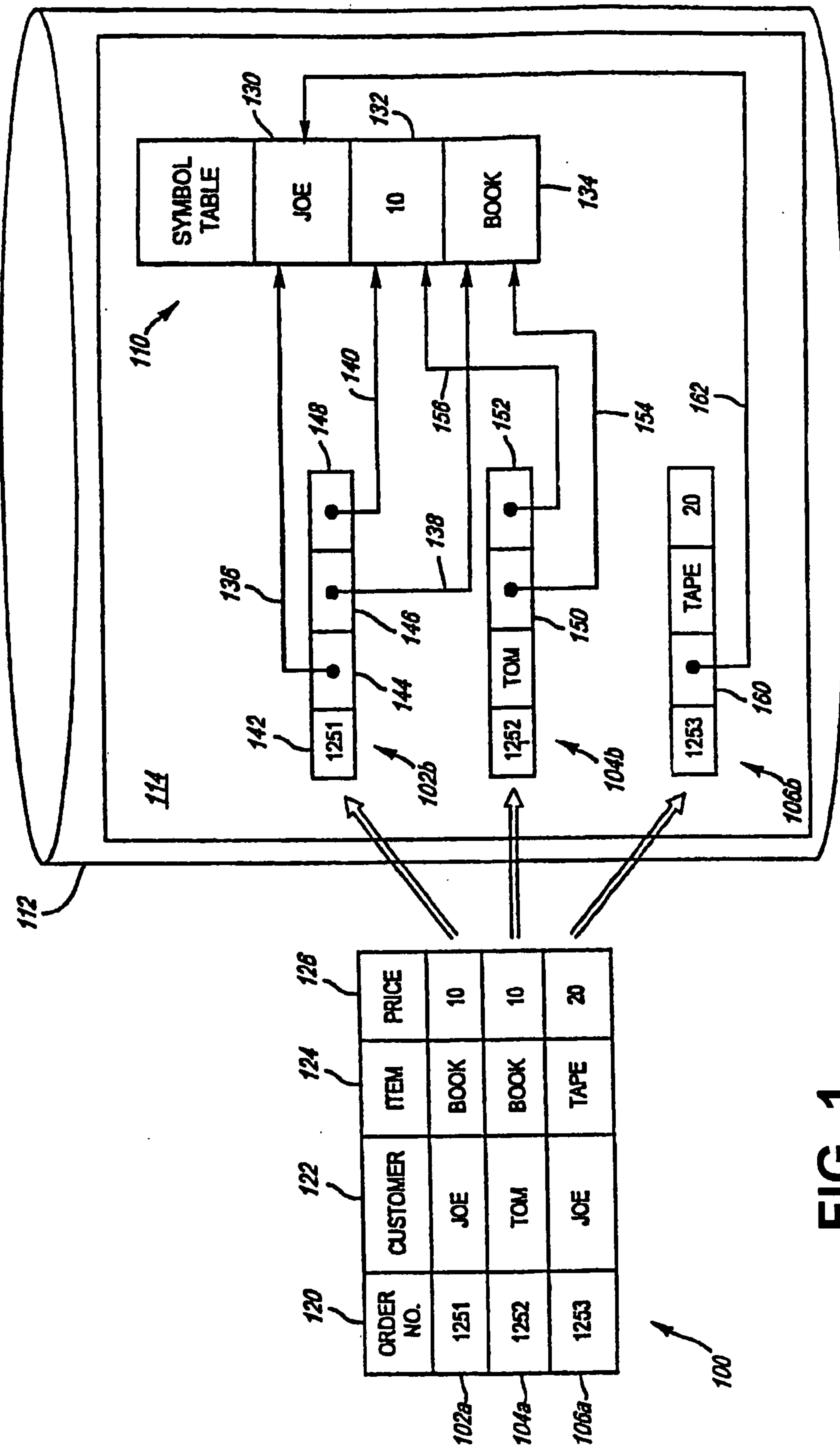


FIG. 1

2/19

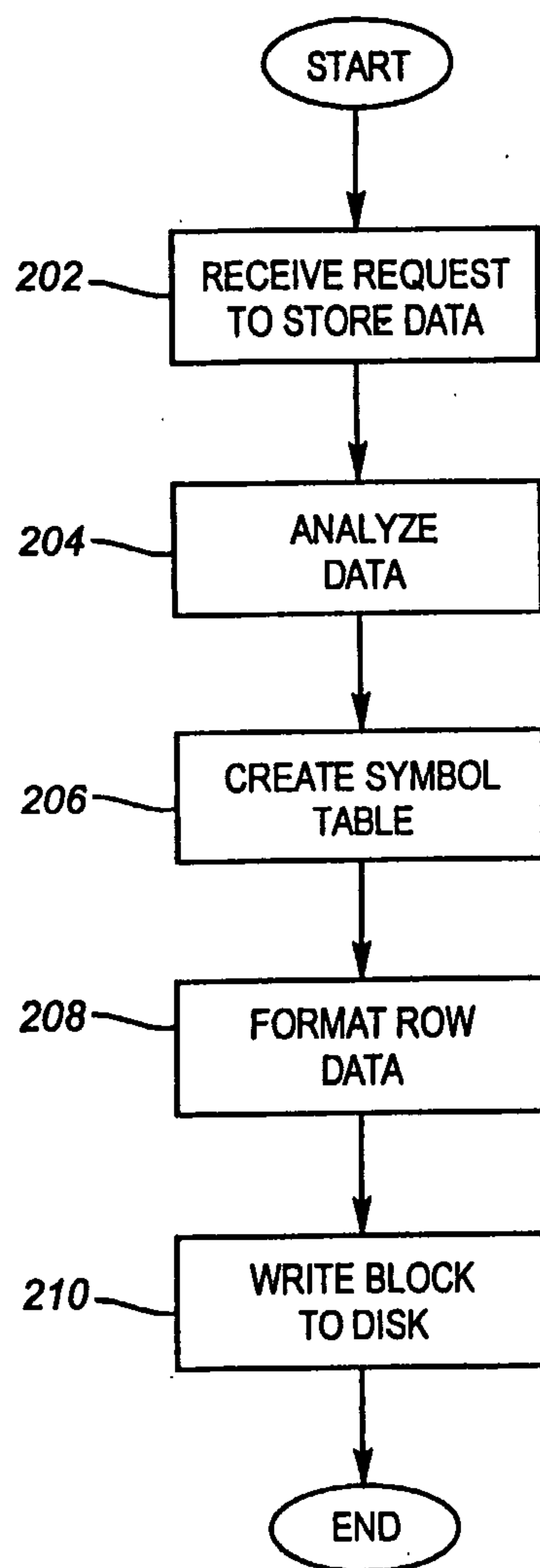


FIG. 2

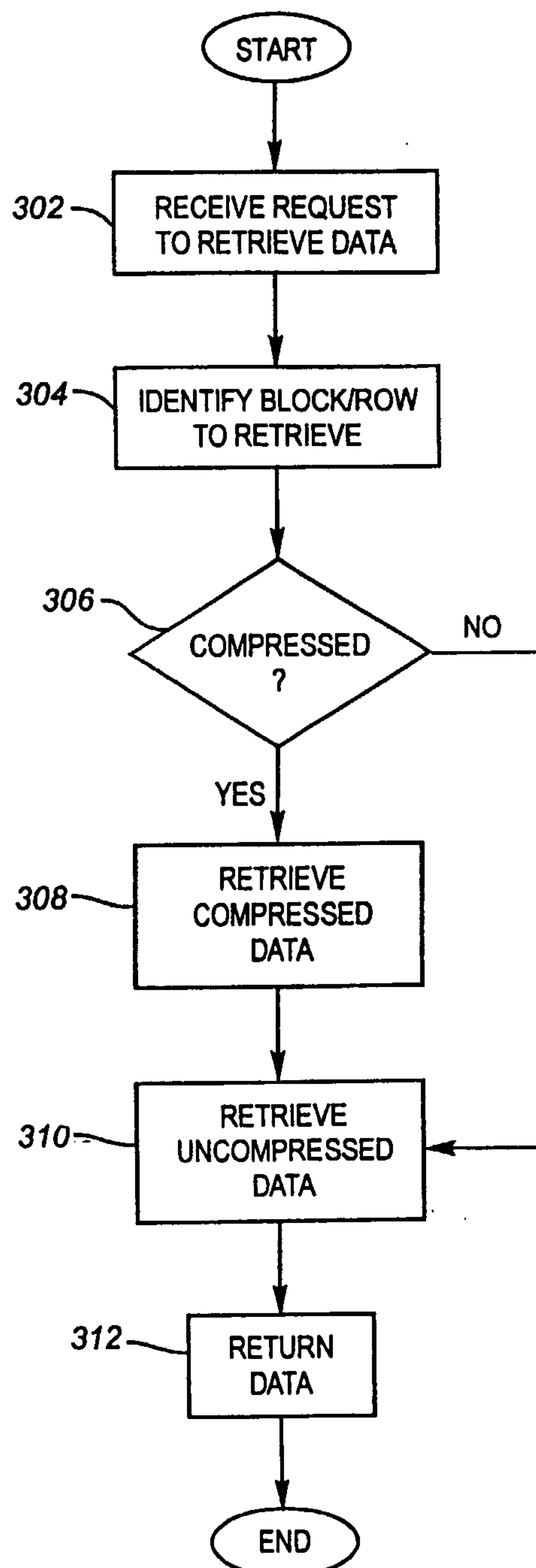


FIG. 3

3/19

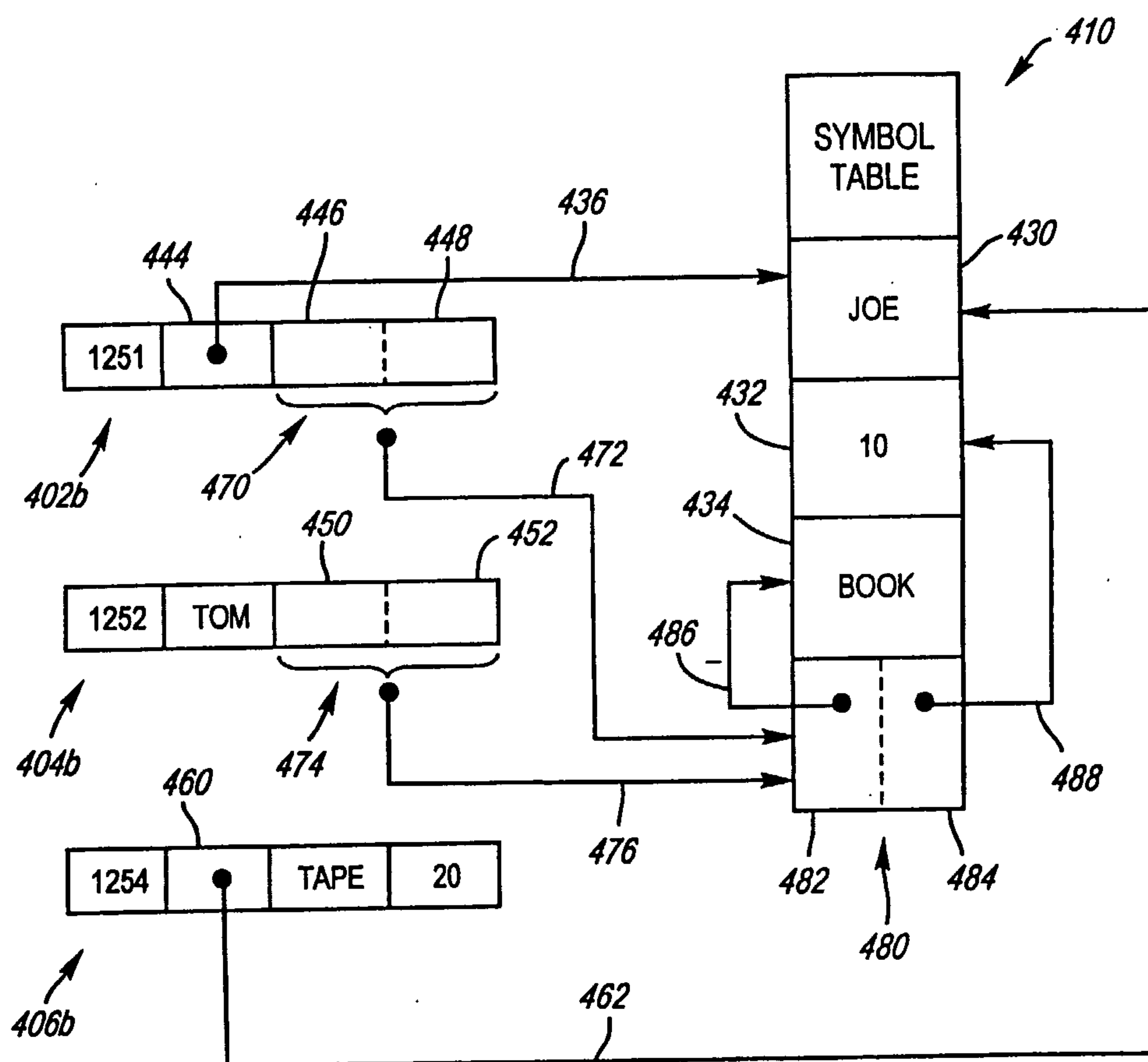


FIG. 4

4/19

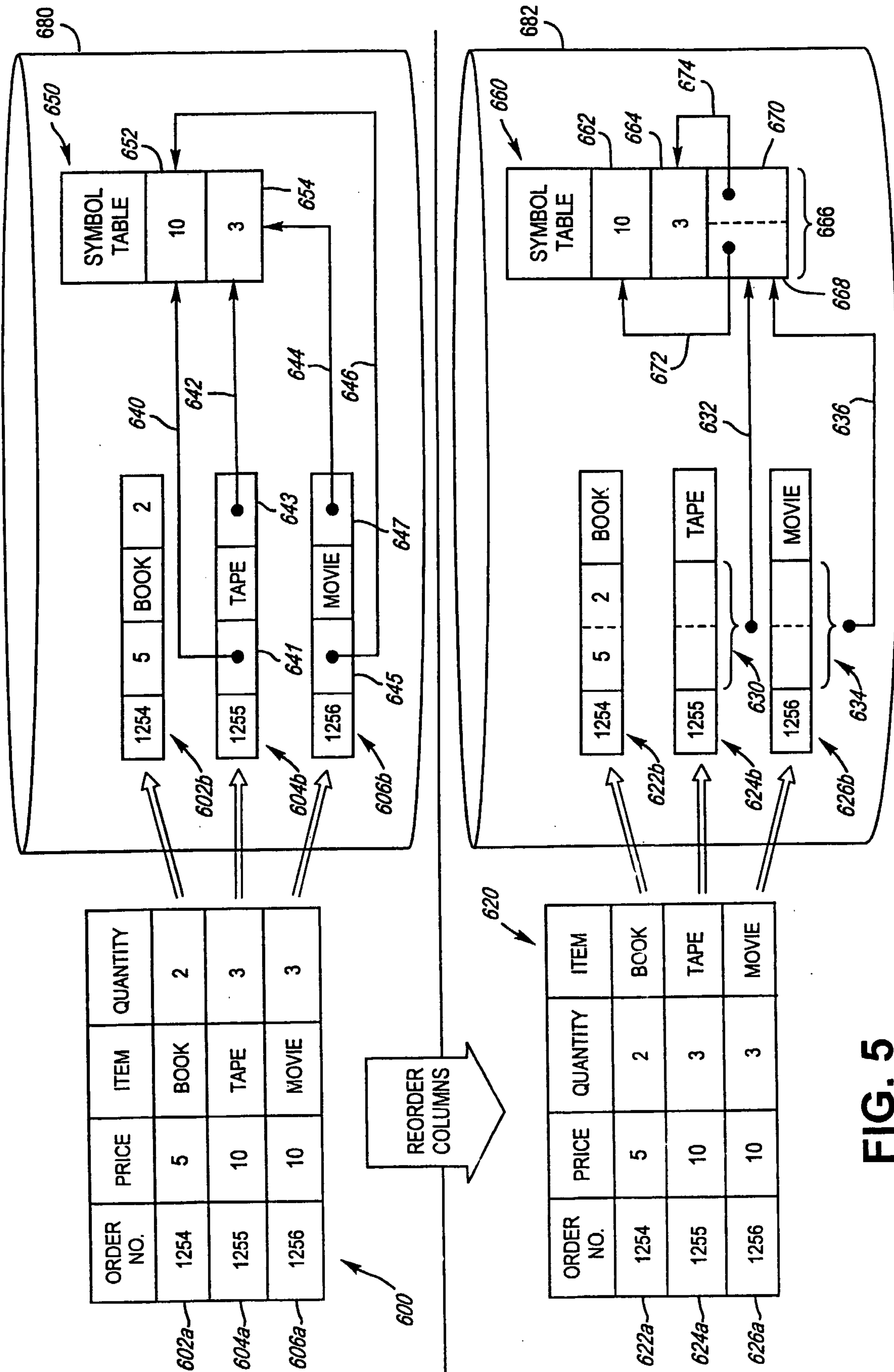
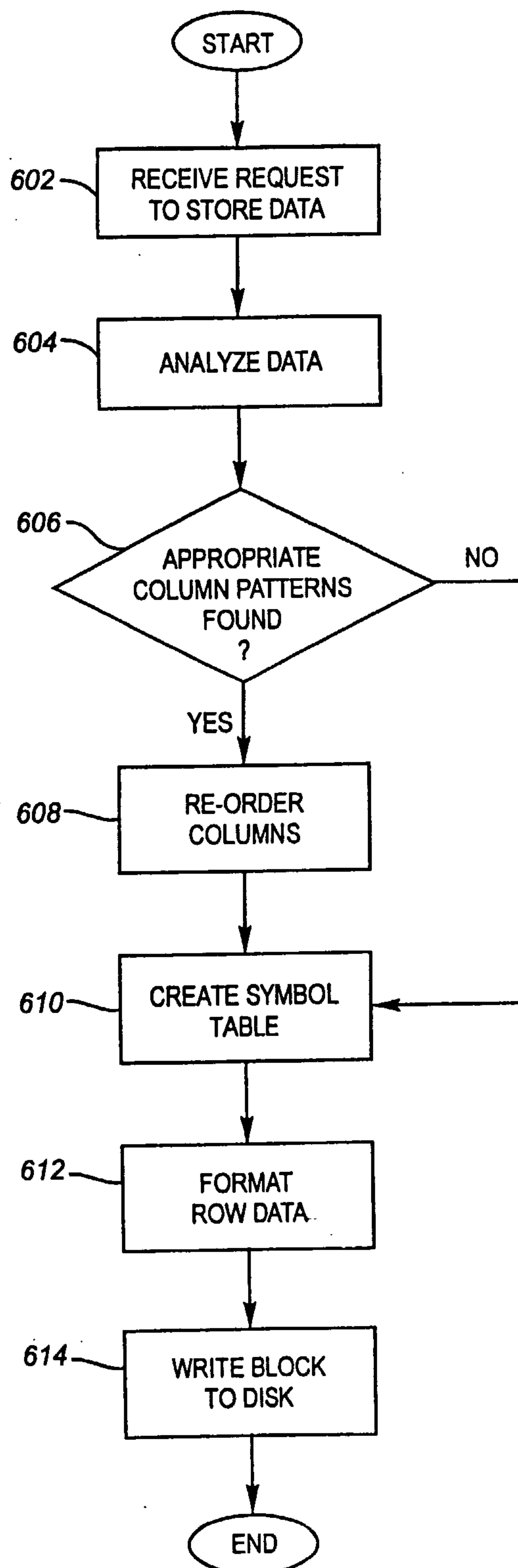


FIG. 5

5/19**FIG. 6**

6/19

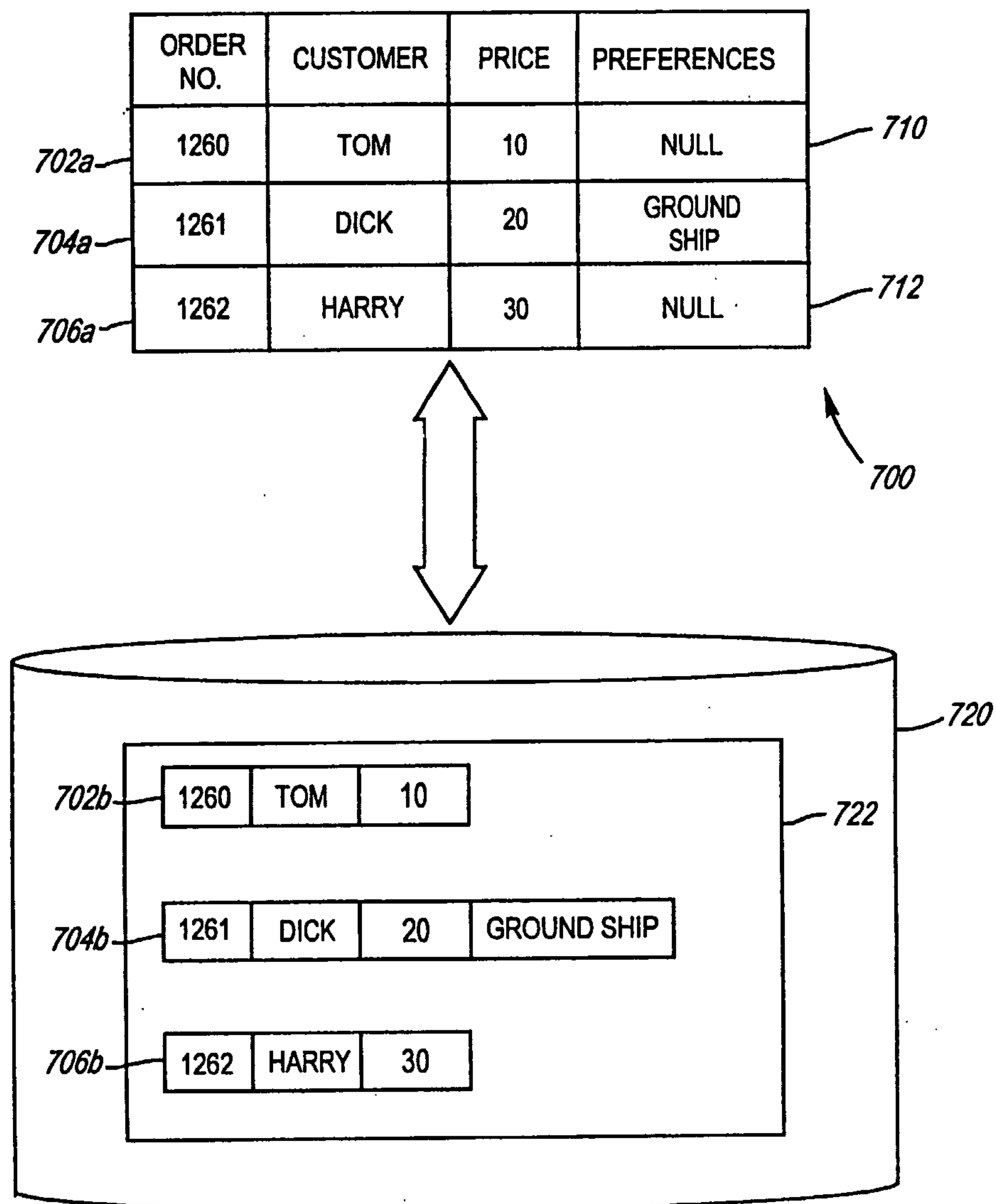


FIG. 7

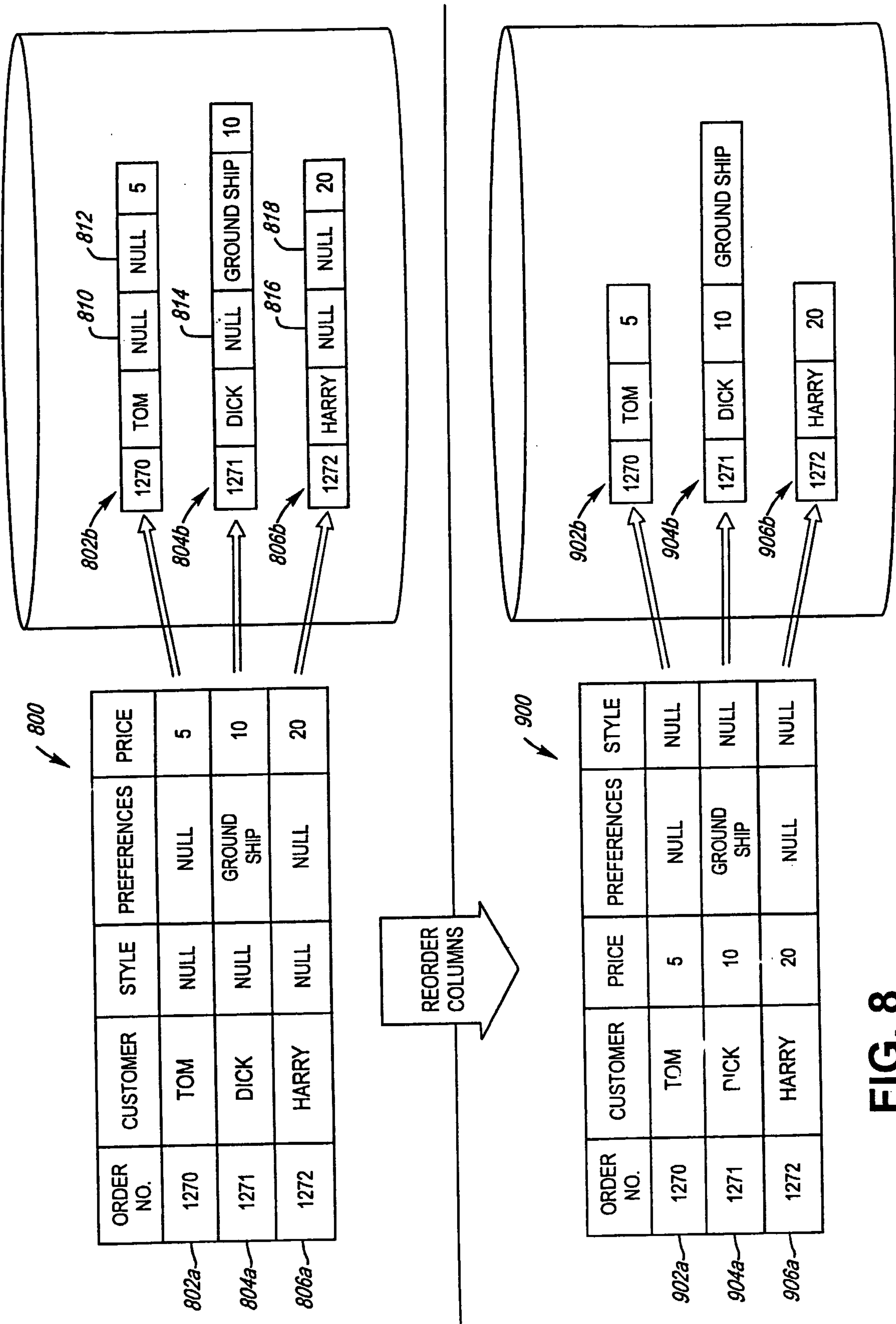


FIG. 8

8/19

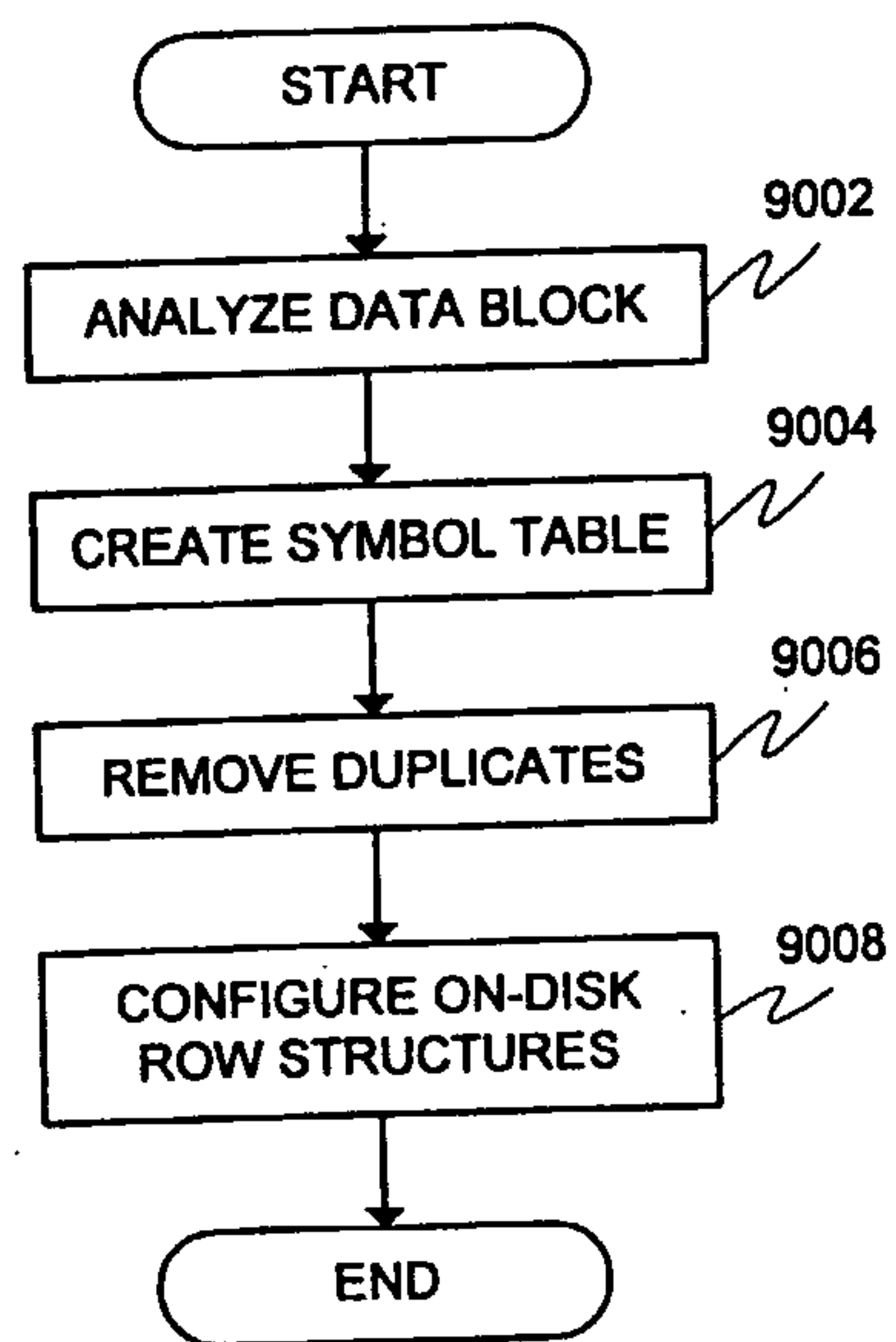


FIG. 9

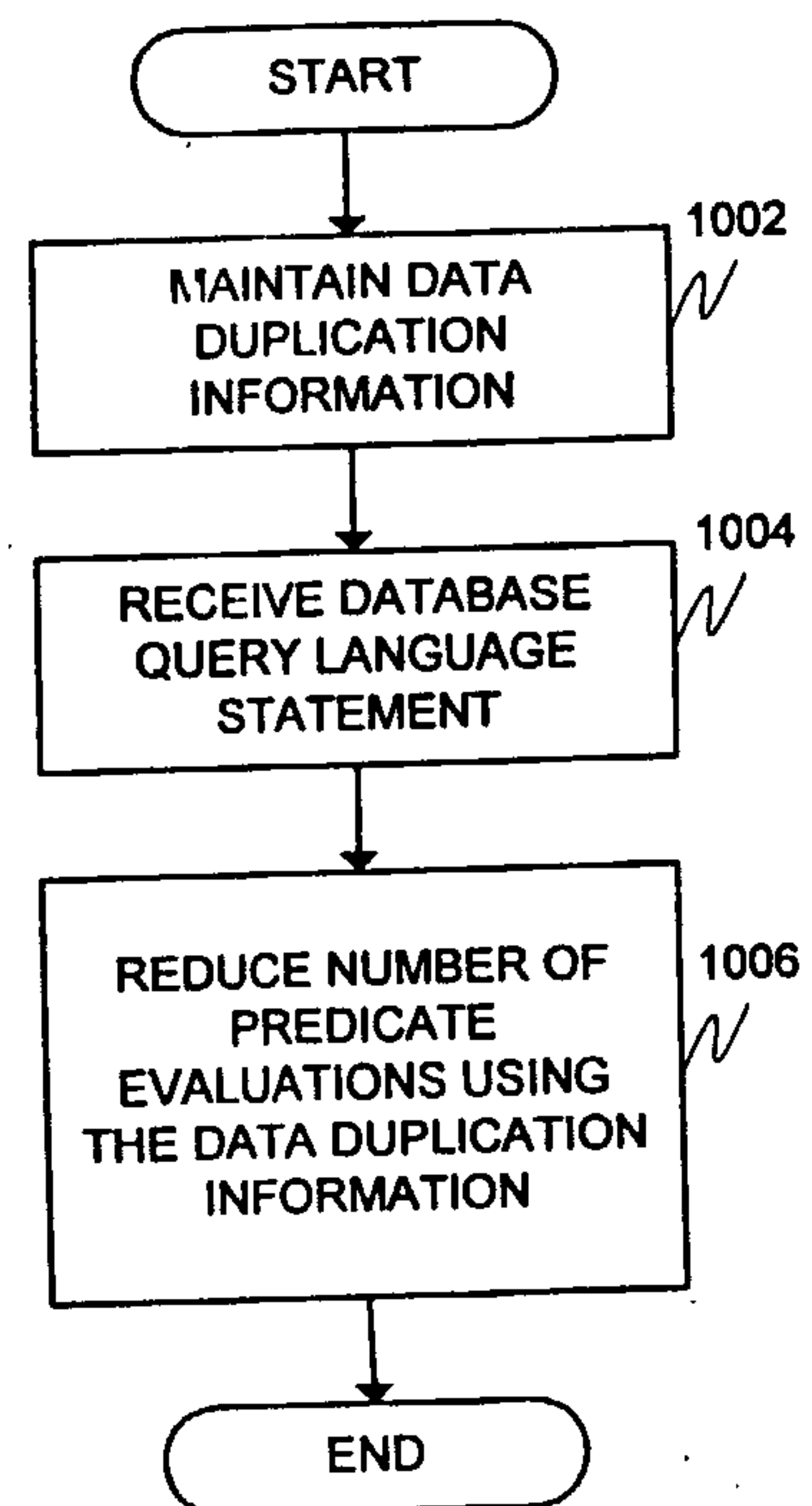
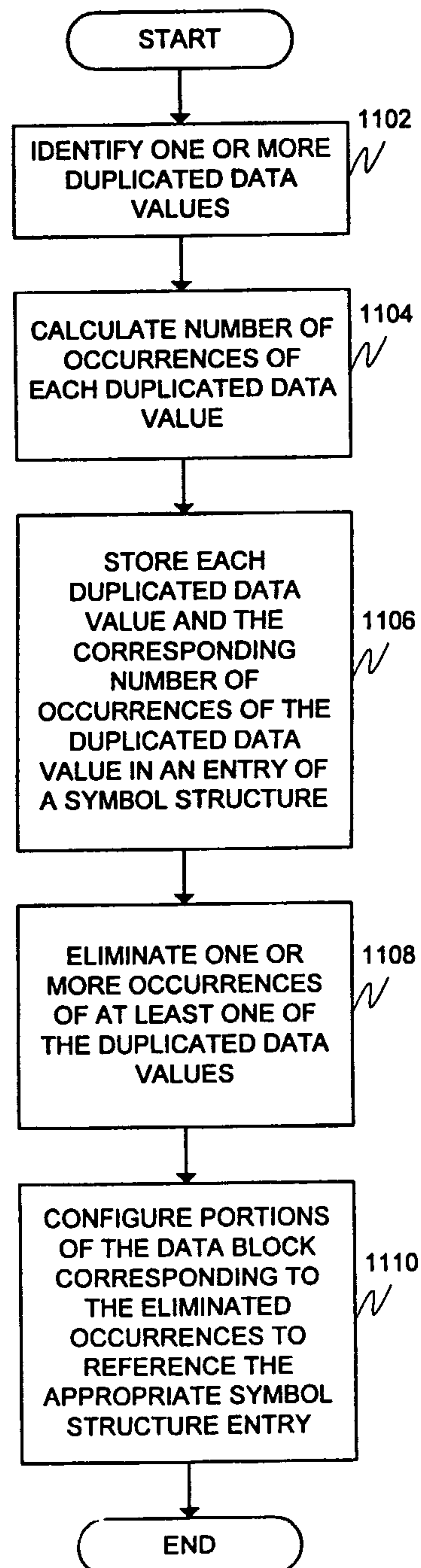


FIG. 10

9/19**FIG. 11**

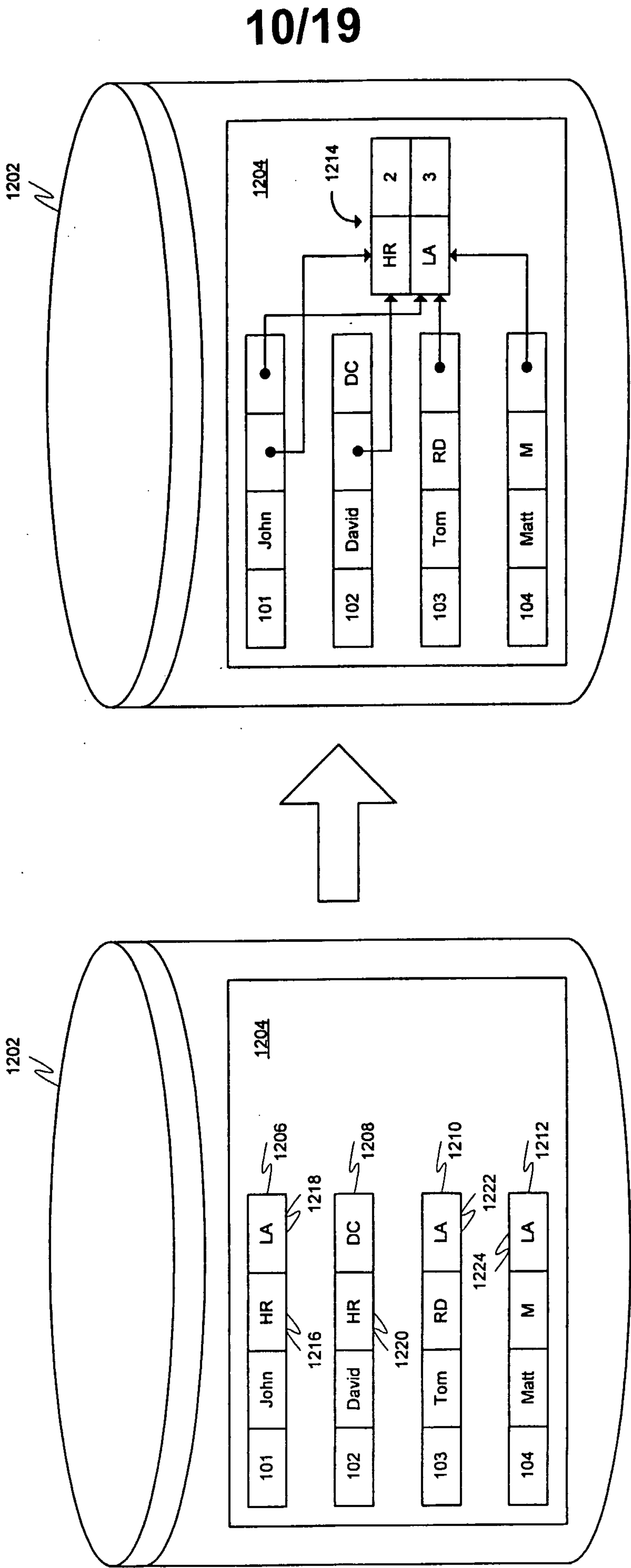
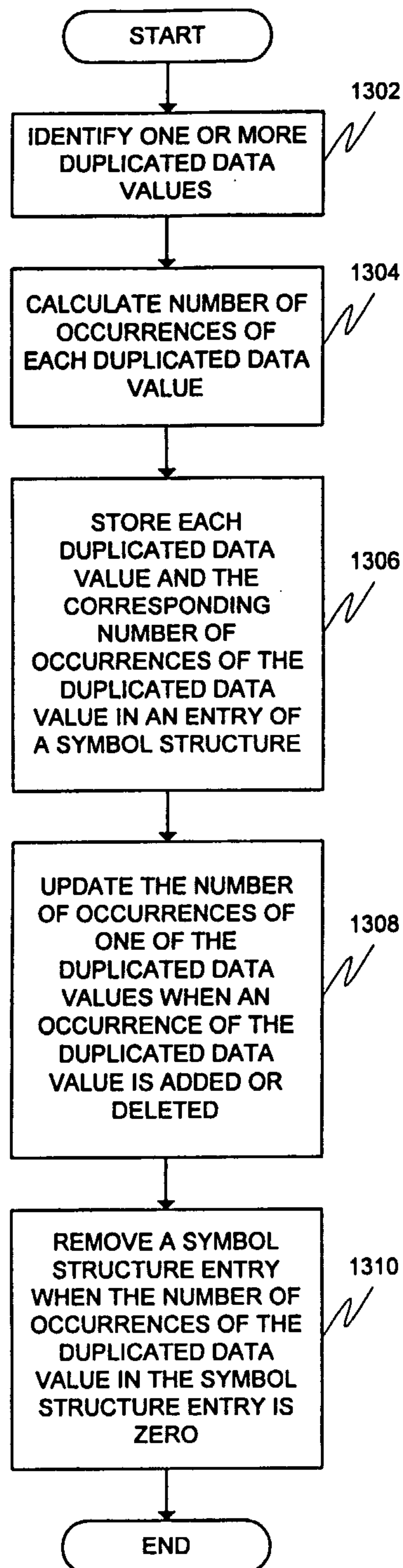


FIG. 12

11/19**FIG. 13**

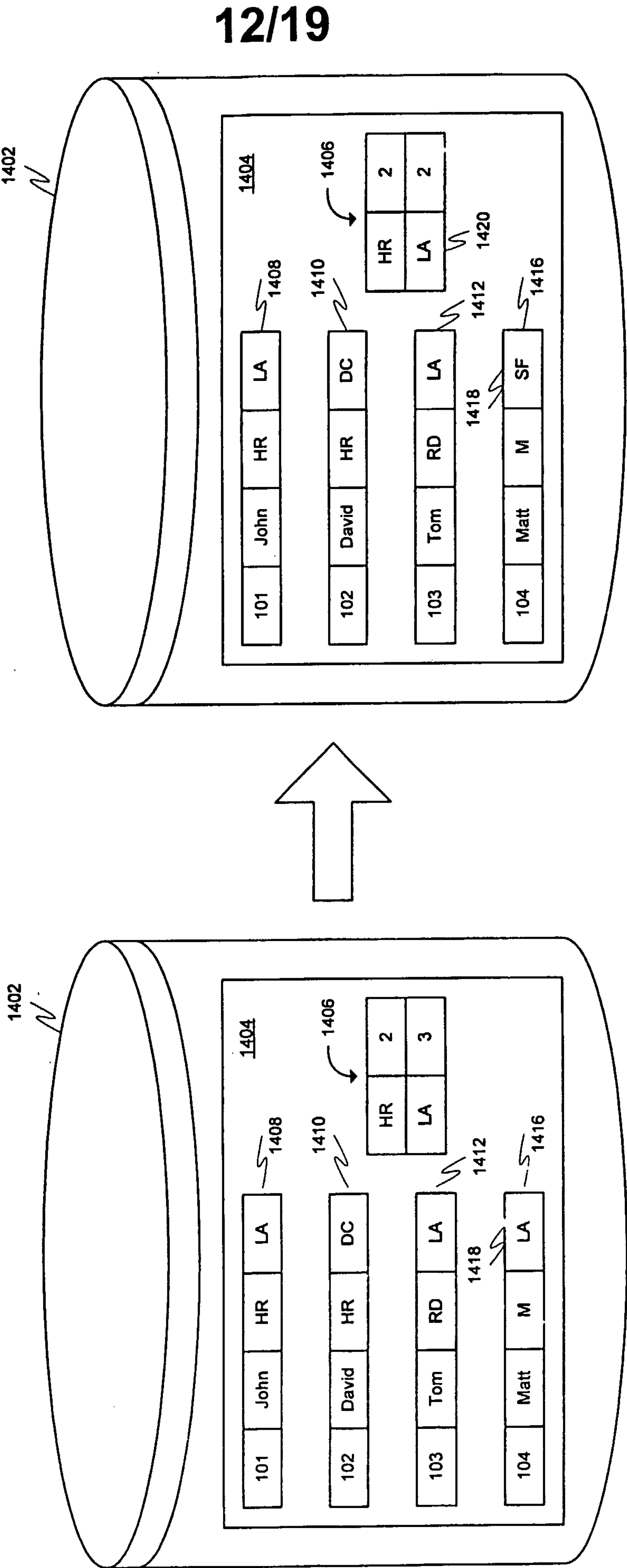


FIG. 14

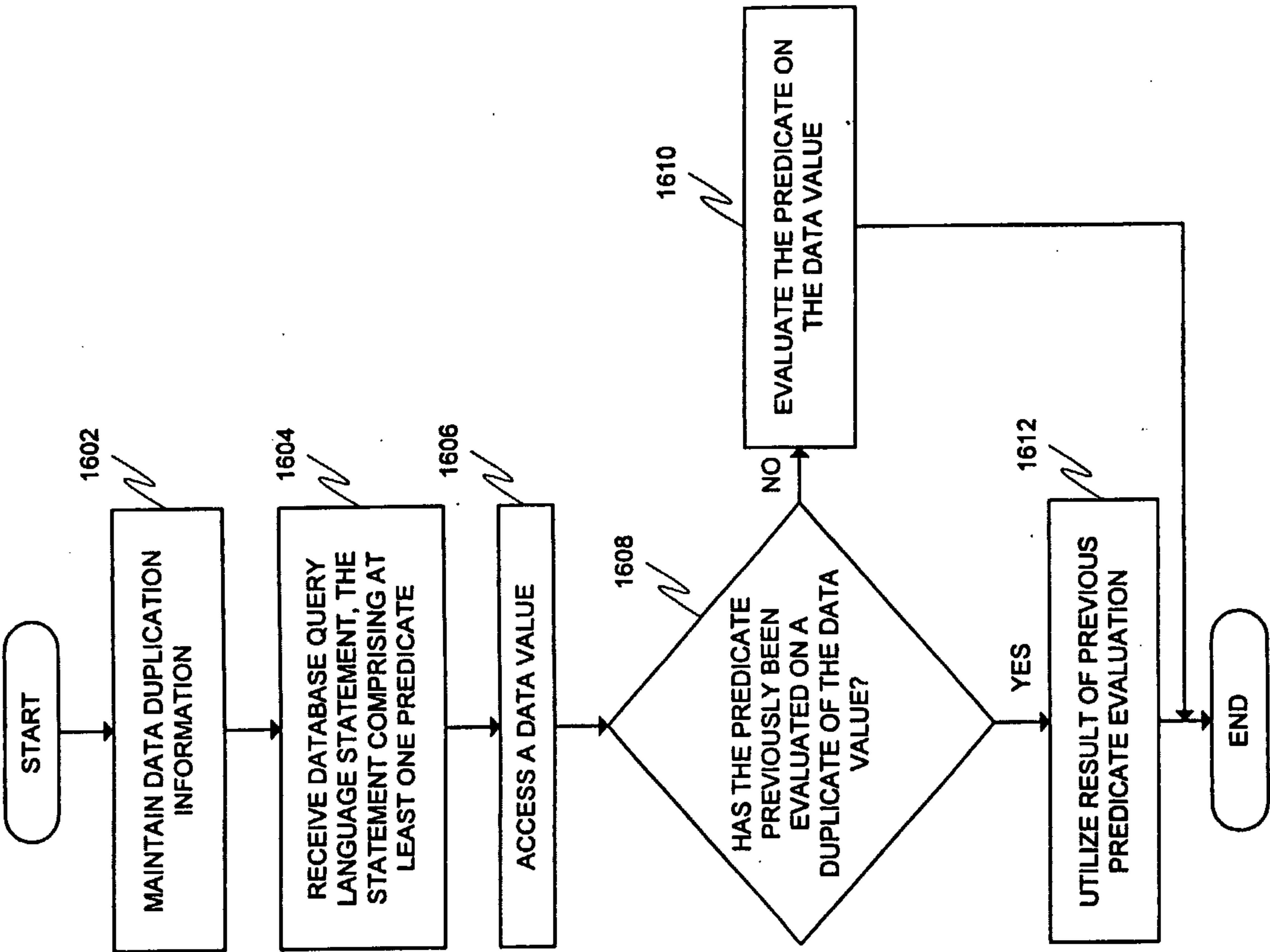


FIG. 16

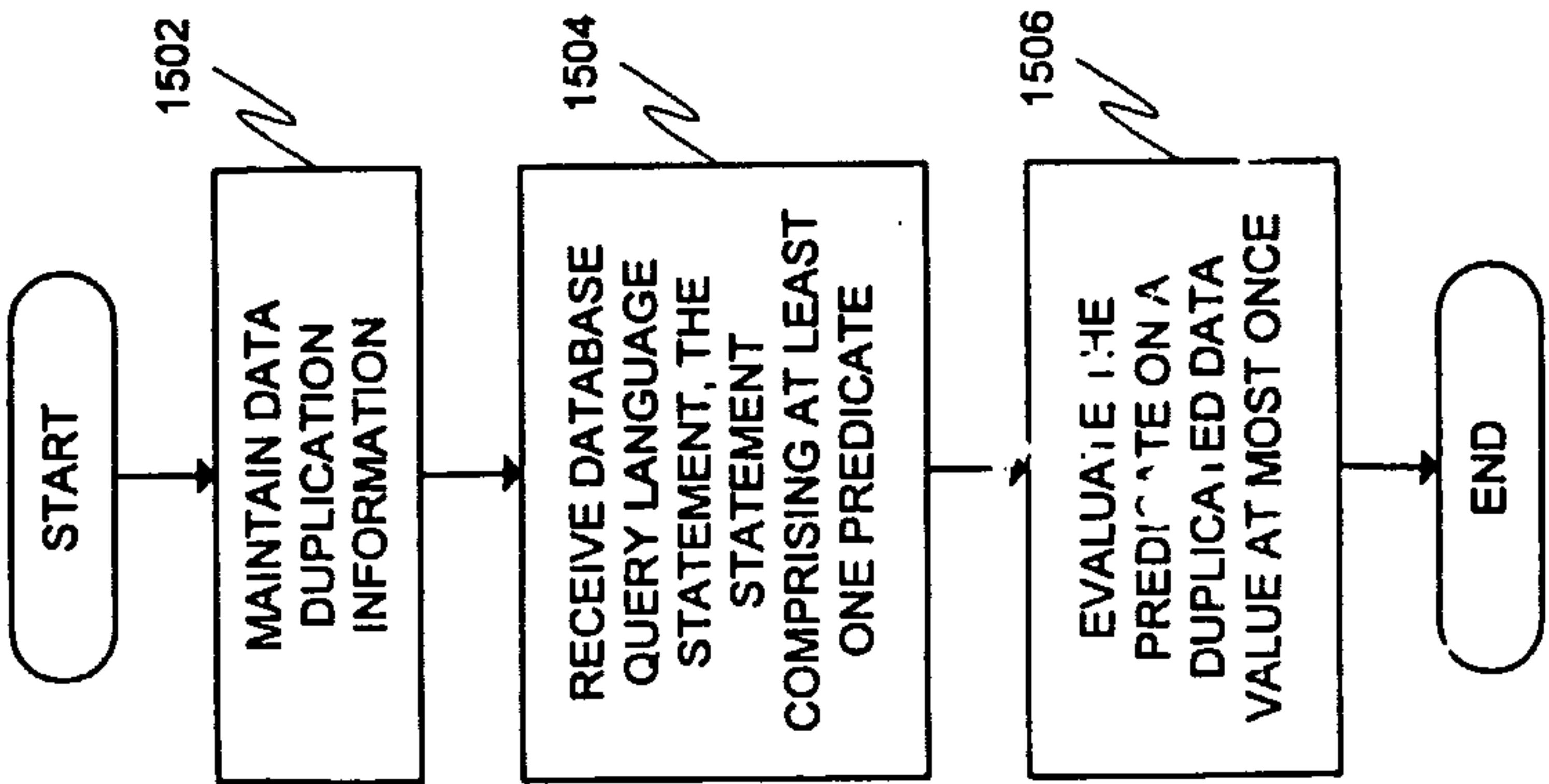
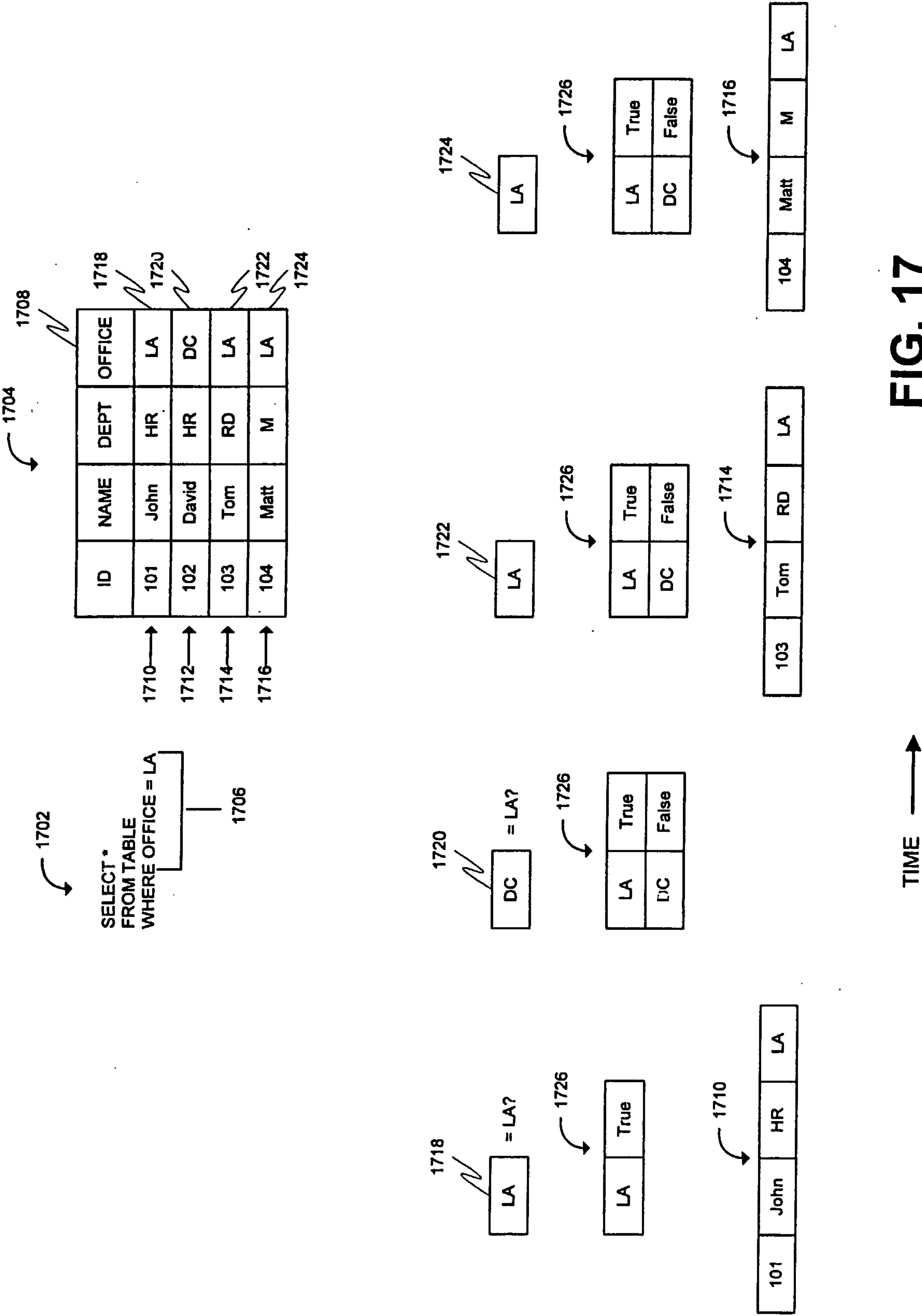


FIG. 15



15/19

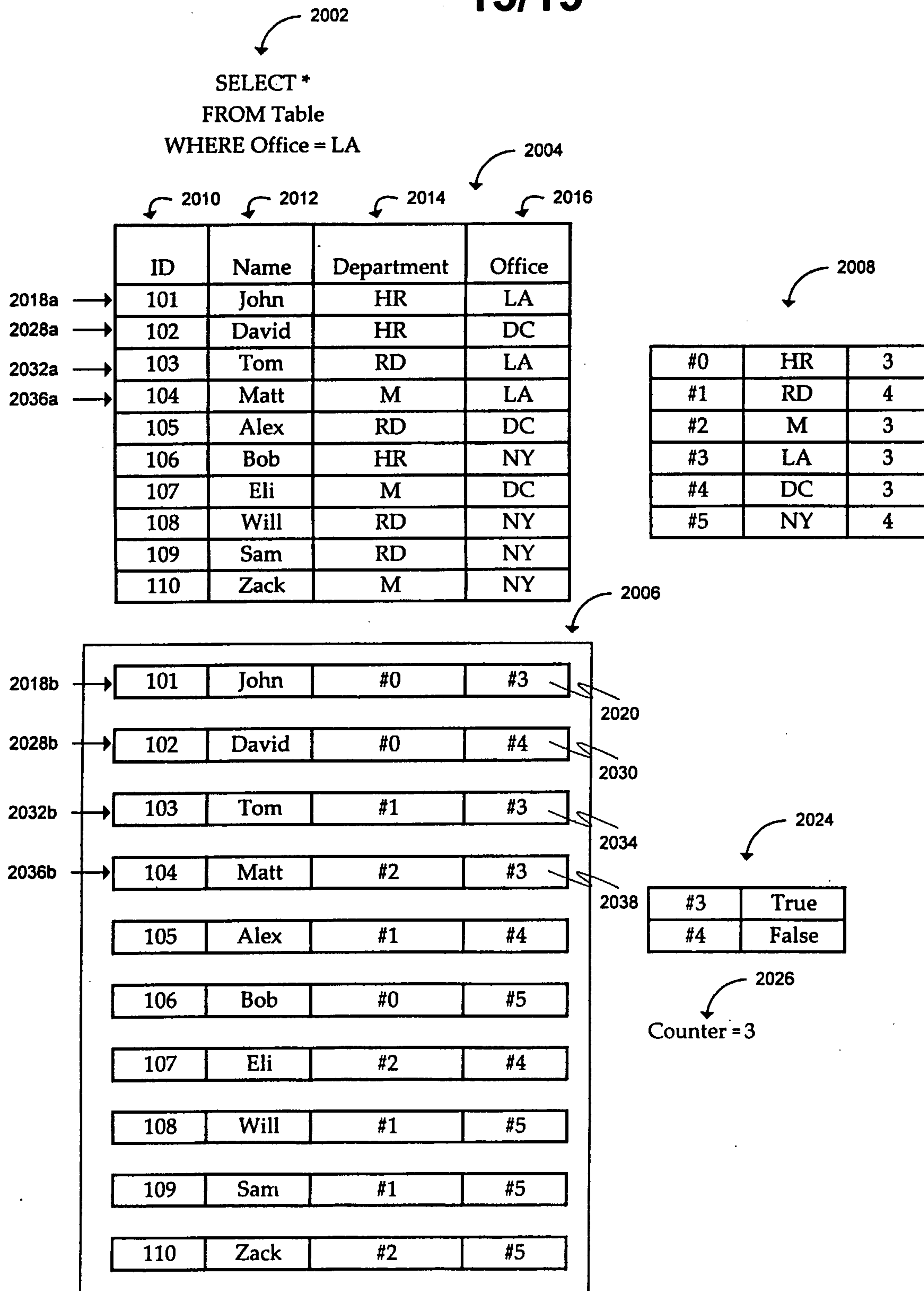
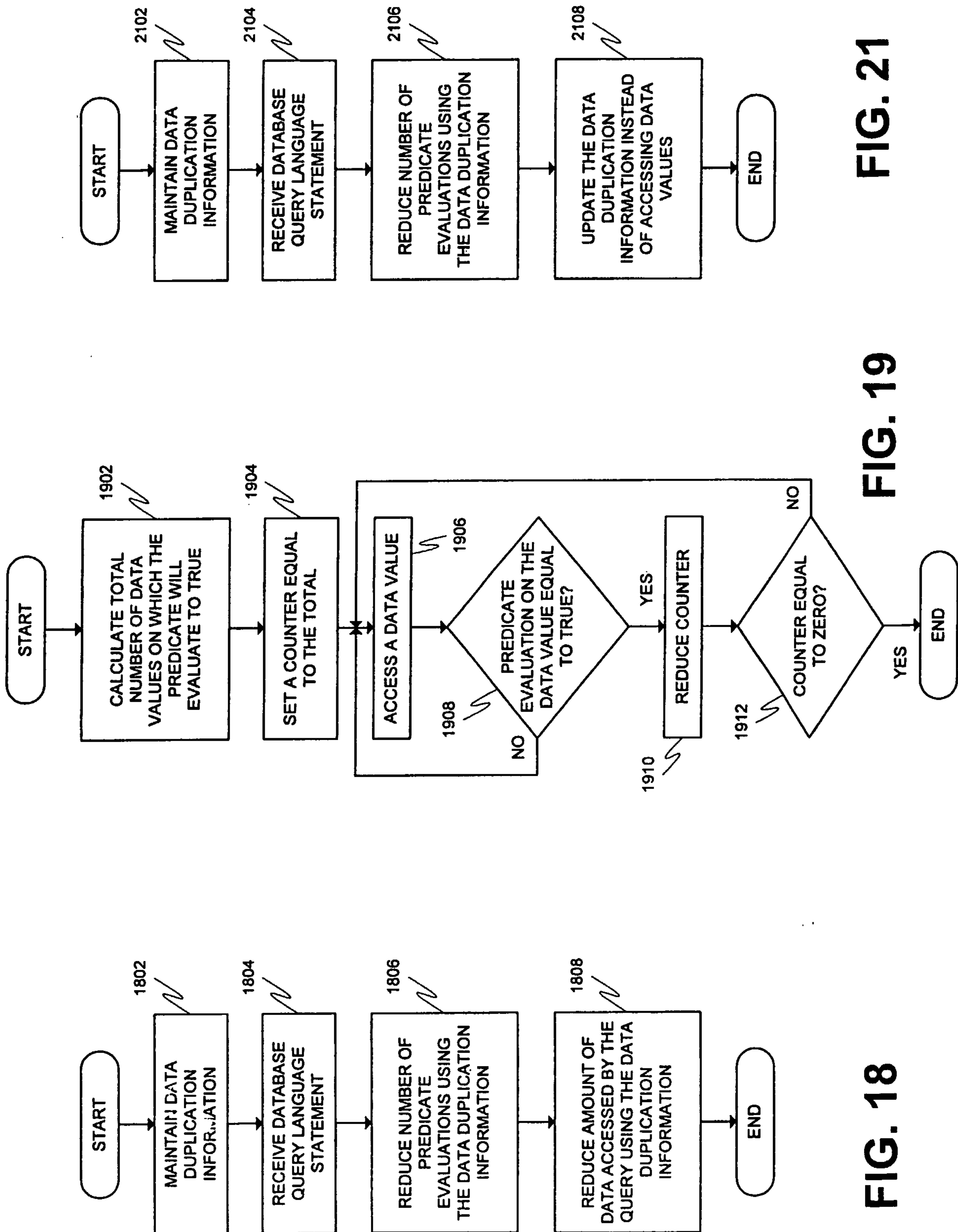


FIG. 20

16/19



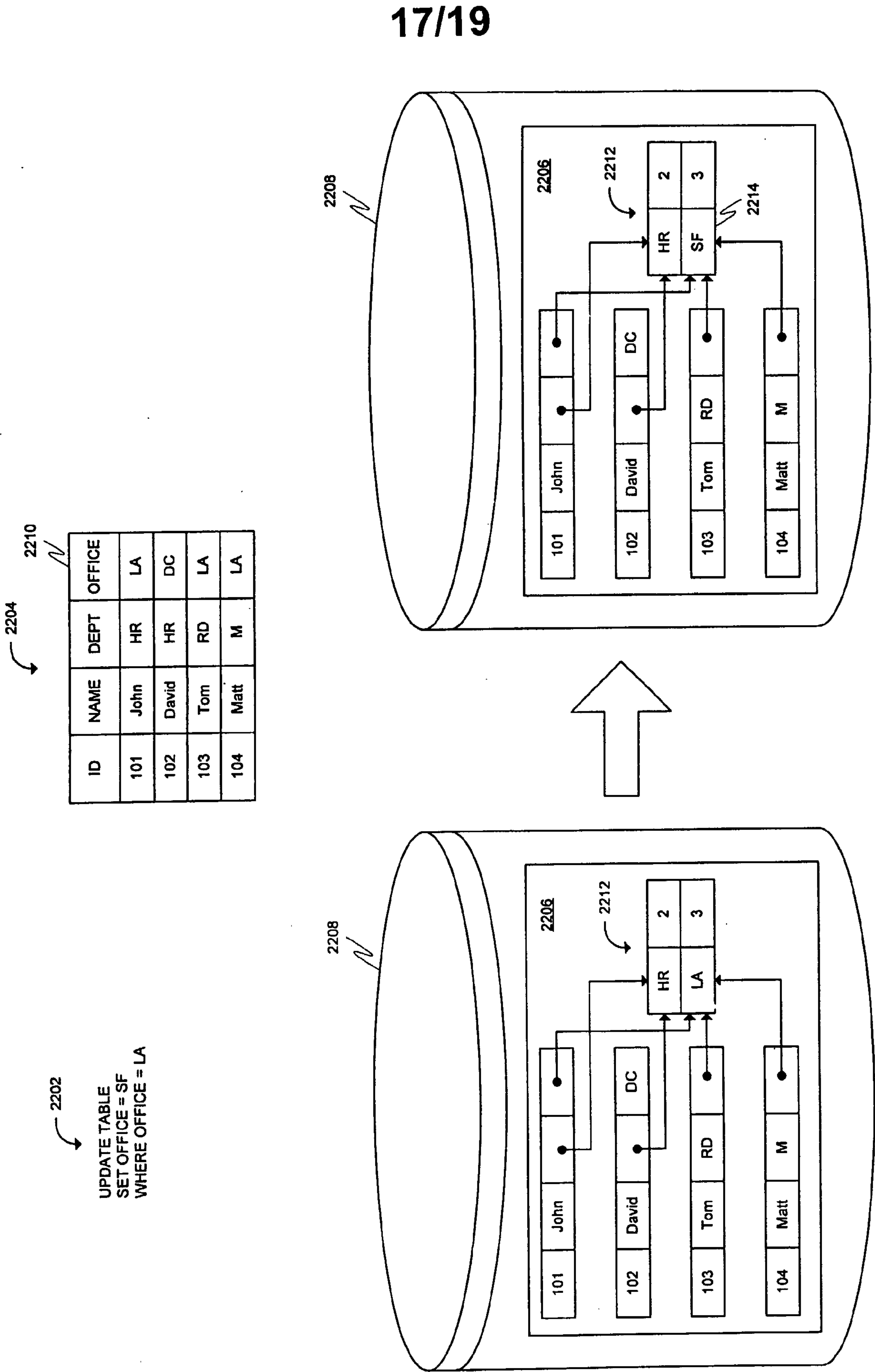


FIG. 22

18/19

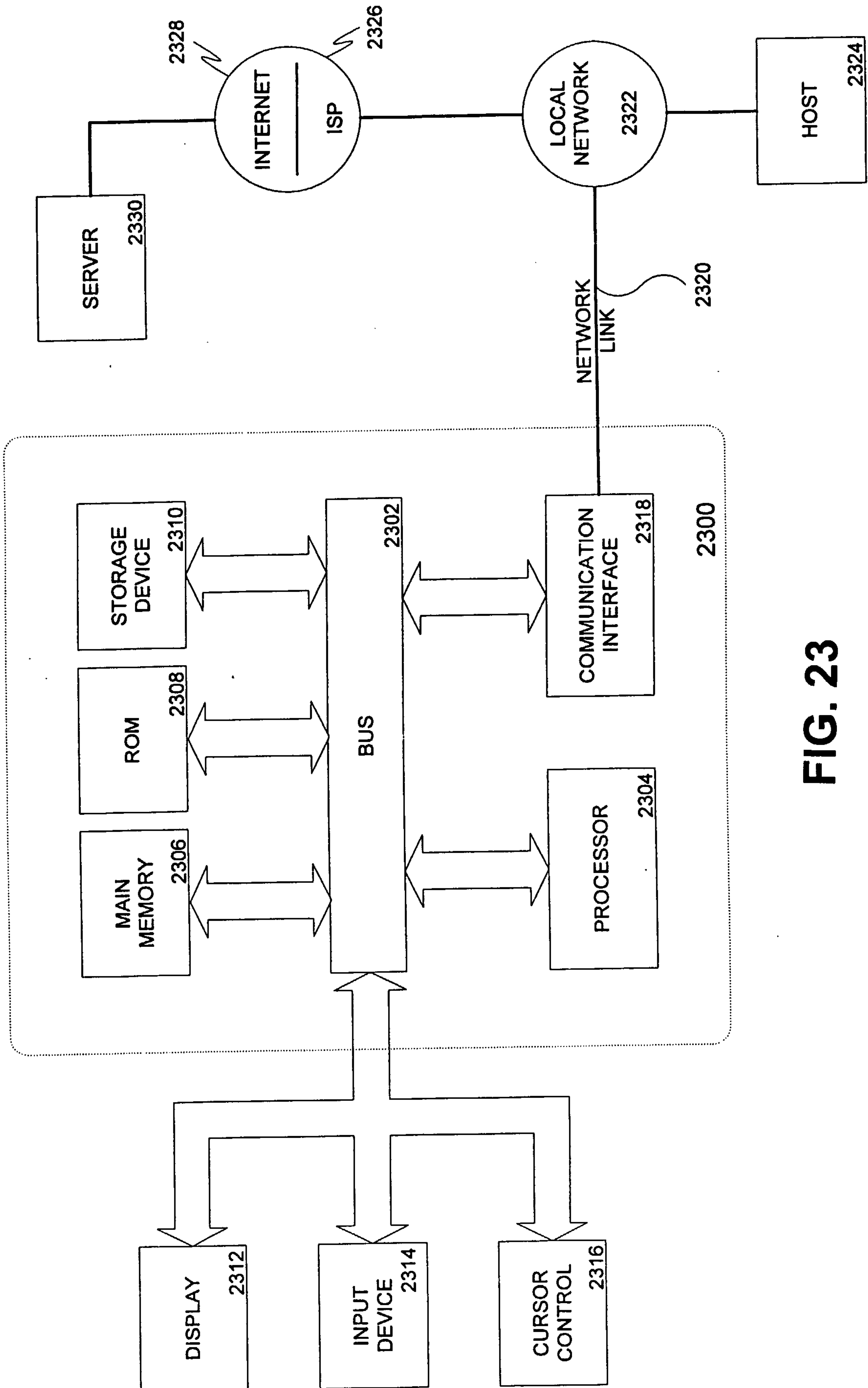
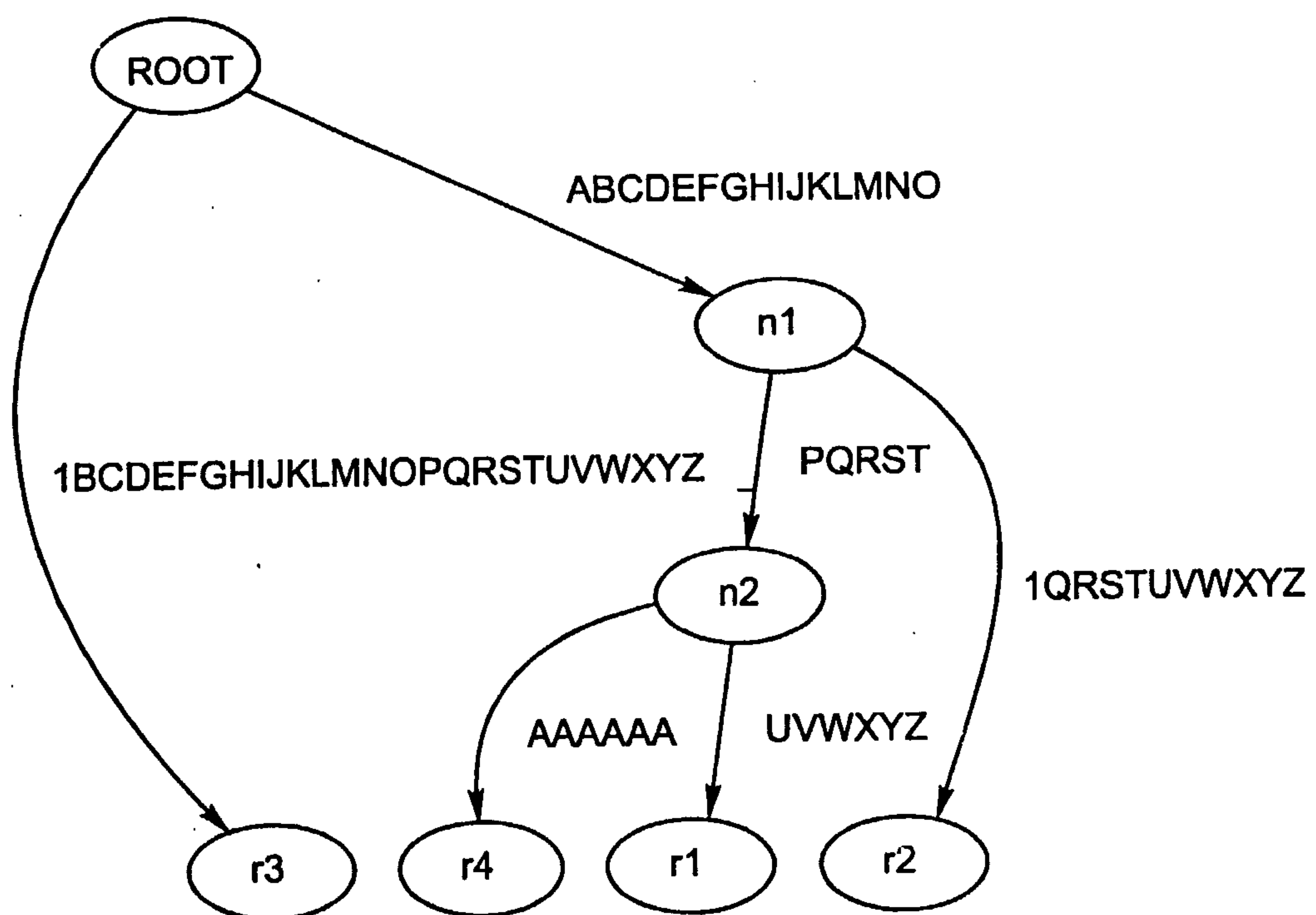


FIG. 23

19/19**FIG. 24**

	120	122	124	126
	ORDER NO.	CUSTOMER	ITEM	PRICE
102a	1251	JOE	BOOK	10
104a	1252	TOM	BOOK	10
106a	1253	JOE	TAPE	20

100

