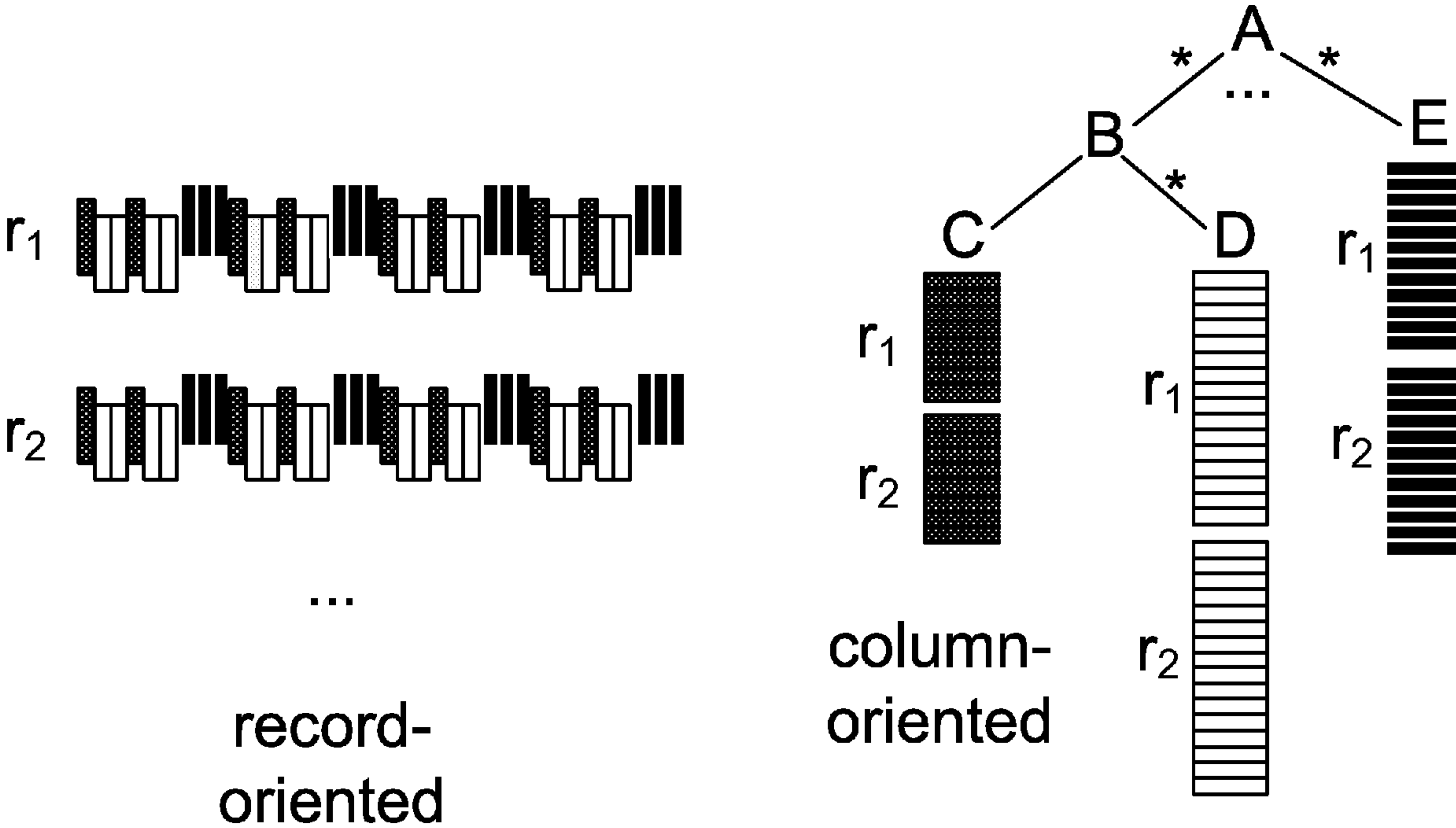




(86) Date de dépôt PCT/PCT Filing Date: 2011/04/04 (87) Date publication PCT/PCT Publication Date: 2011/10/13 (45) Date de délivrance/Issue Date: 2018/03/13 (85) Entrée phase nationale/National Entry: 2012/10/04 (86) N° demande PCT/PCT Application No.: US 2011/031123 (87) N° publication PCT/PCT Publication No.: 2011/126995 (30) Priorités/Priorities: 2010/04/05 (US61/321,106); 2010/04/07 (US61/321,688)	(51) Cl.Int./Int.Cl. <i>G06F 17/30</i> (2006.01) (72) Inventeurs/Inventors: GUBAREV, ANDREY, US; MELNIK, SERGEY, US; LONG, JING JING, US; ROMER, GEOFFREY M., US; SHIVAKUMAR, NARAYANAN, US; TOLTON, MATTHEW B., US; VASSILAKIS, THEODORE, US (73) Propriétaire/Owner: GOOGLE INC., US (74) Agent: GOWLING WLG (CANADA) LLP
--	---

(54) **Titre :** REPRESENTATIONS D'ENREGISTREMENTS MIS EN PAGE EN COLONNES
(54) **Title:** COLUMNAR STORAGE REPRESENTATIONS OF RECORDS



(57) **Abrégé/Abstract:**
A computer system accesses a collection of data records. Each record in the collection includes a plurality of data values and a plurality of data elements that identify semantics of corresponding data values from the plurality of data values. One or more of the data records each include multiple instances of a same data element and include data values that correspond to the multiple instances of the same data element. The computer system generates a collection of columnar stripes. The collection of columnar stripes includes the data values from each data record in the collection of data records. Each columnar stripe in the collection of columnar stripes includes all data values that correspond to a specific data element from each of the records in the collection of records.

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

(19) World Intellectual Property Organization
International Bureau(43) International Publication Date
13 October 2011 (13.10.2011)(10) International Publication Number
WO 2011/126995 A1(51) International Patent Classification:
G06F 17/30 (2006.01)(21) International Application Number:
PCT/US2011/031123(22) International Filing Date:
4 April 2011 (04.04.2011)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/321,106 5 April 2010 (05.04.2010) US
61/321,688 7 April 2010 (07.04.2010) US(71) Applicant (for all designated States except US):
GOOGLE INC. [US/US]; 1600 Amphitheatre Parkway,
Mountain View, California 94043 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **GUBAREV, An-**
drey [UA/US]; 3020 122nd Place NE, Bellevue, Wash-
ington 98005 (US). **MELNIK, Sergey** [RU/US]; 12916
NE 91st Ln, Kirkland, Washington 98033 (US). **LONG,**
Jing Jing [US/US]; 300 110th Ave NE #401, Bellevue,
Washington 98004 (US). **ROMER, Geoffrey M.**
[US/US]; 11428 NE 67th St #10, Kirkland, Washington
98033 (US). **SHIVAKUMAR, Narayanan** [US/US];
13925 SE 5th Street, Bellevue, Washington 98005 (US).**TOLTON, Matthew B.** [US/US]; 11808 93rd Lane NE,
Apt. 302, Kirkland, Washington 98034 (US). **VASSI-**
LAKIS, Theodore [US/US]; 13021 N.E. 100th Street,
Kirkland, Washington 98033 (US).(74) Agent: **HOFF, Christopher C.**; Fish & Richardson P.C.,
P.O. Box 1022, Minneapolis, Minnesota 55440-1022
(US).(81) Designated States (unless otherwise indicated, for every
kind of national protection available): AE, AG, AL, AM,
AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ,
CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO,
DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT,
HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP,
KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD,
ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI,
NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD,
SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR,
TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.(84) Designated States (unless otherwise indicated, for every
kind of regional protection available): ARIPO (BW, GH,
GM, KE, LR, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG,
ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ,
TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK,
EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU,
LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK,

[Continued on next page]

(54) Title: COLUMNAR STORAGE REPRESENTATIONS OF RECORDS

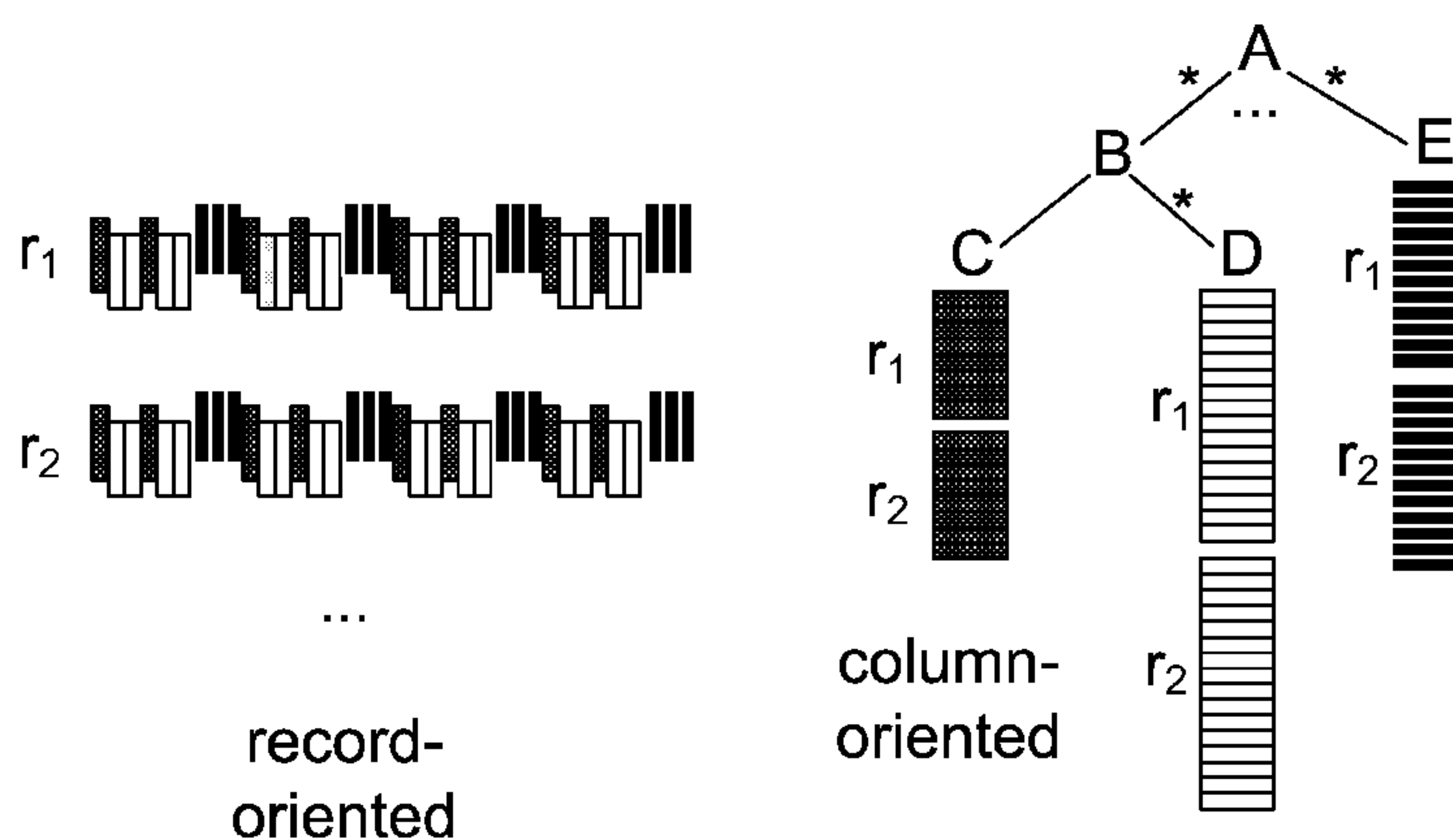


FIG. 1

(57) Abstract: A computer system accesses a collection of data records. Each record in the collection includes a plurality of data values and a plurality of data elements that identify semantics of corresponding data values from the plurality of data values. One or more of the data records each include multiple instances of a same data element and include data values that correspond to the multiple instances of the same data element. The computer system generates a collection of columnar stripes. The collection of columnar stripes includes the data values from each data record in the collection of data records. Each columnar stripe in the collection of columnar stripes includes all data values that correspond to a specific data element from each of the records in the collection of records.

WO 2011/126995 A1



SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, **Published:**
GW, ML, MR, NE, SN, TD, TG).

— *with international search report (Art. 21(3))*

COLUMNAR STORAGE REPRESENTATIONS OF RECORDS

CROSS-REFERENCE TO RELATED APPLICATION

[0001] This application claims the benefit of U.S. Provisional Application No. 61/321,106, filed April 5, 2010 and U.S. Provisional Application No. 61/321,688, filed April 7, 2010.

TECHNICAL FIELD

[0002] This document generally describes techniques, methods, systems, and mechanisms for generating and processing columnar storage representations of records.

BACKGROUND

[0003] The present disclosure generally relates to large-scale analytical data processing. Such data processing has become widespread in web companies and across industries, not least due to low-cost storage that enabled collecting vast amounts of business-critical data. Putting this data at the fingertips of analysts and engineers has grown increasingly important; interactive response times often make a qualitative difference in data exploration, monitoring, online customer support, rapid prototyping, debugging of data pipelines, and other tasks. Performing interactive data analysis at scale demands a high degree of parallelism. For example, reading one terabyte of compressed data in one second using today's commodity disks may require tens of thousands of disks. Similarly, CPU-intensive queries may need to run on thousands of cores to complete within seconds.

SUMMARY

[0004] A scalable, interactive ad-hoc query system is herein disclosed for analysis of data. By combining multi-level execution trees and columnar data layout, the described system and methods is capable of running rapid and efficient queries such as aggregation queries. A columnar storage representation for nested records, a prevalent data model that may be used in many web-scale and scientific datasets, is described. In accordance with an embodiment, a record is decomposed into column stripes, each column encoded as a set of blocks, each block containing field values and repetition and definition level information. Level information is generated using a tree of field writers, whose structure matches the field hierarchy in the record schema. The record can be assembled from the columnar data efficiently using a finite state machine that reads the field values and level information for each field and appends the values sequentially to the output records. If only a subset of fields need to be retrieved, a simpler finite state machine can be constructed that is cheaper to execute. Moreover, by storing additional metadata such as constraint information with the columnar storage representation, additional types of queries can be supported.

[0005] A multi-level serving tree is used to execute queries. In one embodiment, a root server receives an incoming query, reads metadata from the tables, and routes the queries to a next level in the serving tree. Leaf servers communicate with a storage layer or access the data on local storage, where the stored data can be replicated, and read stripes of nested data in the columnar representation. Each server can have an internal execution tree corresponding to a physical query execution plan, comprising a set of iterators that scan input

columns and emit results of aggregates and scalar functions annotated with level information. In another embodiment, a query dispatcher is provided which schedules queries based on their priorities and balances the load. The query dispatcher also provides fault tolerance when one server becomes much slower than others or as a replica becomes unreachable. The query dispatcher can compute a histogram of processing times for execution threads on the leaf servers and reschedule to another server when processing time takes a disproportionate amount of time.

[0006] The columnar data may be queried *in situ*. Maintaining the columnar data on a common storage layer and providing mechanisms to assemble records from the columnar data enables operability with data management tools that analyze data in a record structure. The system may scale to numerous CPUs and be capable of rapidly reading large amounts of data. Particular embodiments can be implemented, in certain instances, to realize one or more of the following advantages. Nested data may be operated on *in situ*, such that the data may be accessed without loading the data with a database management system. Queries of nested data may be performed in a reduced execution time than required by other analysis programs. A columnar storage data structure that is implemented on a common storage layer enables multiple different analysis programs to access the columnar storage data structure.

[0007] As an alternative to the attached claims and the embodiments described in the above description, the present invention could also be described by one of the following embodiments:

[0008] Embodiment 1 is directed to a computer-implemented method. The method comprises accessing, by a computing system, a collection of data

records that are stored in computer memory, each record in the collection of data records including a plurality of data values and a plurality of data elements that identify semantics of corresponding data values from the plurality of data values, one or more of the data records in the collection of data records each including multiple instances of a same data element and including data values that correspond to the multiple instances of the same data element. The method further comprises generating, by the computing system, a collection of columnar stripes, the collection of columnar stripes including the data values from each data record in the collection of data records, each columnar stripe in the collection of columnar stripes including all data values that correspond to a specific data element from each of the records in the collection of records.

[0009] Embodiment 2 is directed to the method of embodiment 1. The method further comprises generating, by the computing system and for each data value in each columnar stripe in the collection of columnar stripes, data that identifies a location of the respective data value in a respective data record from the collection of data records.

[0010] Embodiment 3 is directed to the method of embodiment 2, wherein the data consists of a repetition value and a definition value.

[0011] Embodiment 4 is directed to the method of any one of embodiments 2 through 3. The method further comprises reconstructing, from (i) the columnar stripes in the collection of columnar stripes, and (ii) the data, a set of records that contains just a subset of data elements from the records in the collection of data records.

[0012] Embodiment 5 is directed to the method of any one of embodiments 1 through 4, further comprising generating, for storage with the data values in the

collection of columnar stripes, a repetition value for each particular data value in the collection of columnar stripes, wherein a path for each particular data element includes any one or more parent data elements to the particular data element; wherein the repetition value for each particular data value identifies a most-recently repeated data element in the path of the particular data element that corresponds to the particular data value; wherein the most recently-repeated data element in the path of the particular data element is the data element in the path of the particular data element that is encountered for a second time during an analysis, of a particular data record that includes the particular data value, that works upwards towards a beginning of the particular data record from a position of the particular data value in the particular data record.

[0013] Embodiment 6 is directed to the method of any one of embodiments 1 through 5, wherein each particular data element, of the data elements that are included in the collection of data elements, are associated with a respective path that includes any one or more of the parent data elements to the particular data element. The method further comprises generating, for storage with the data values in the columnar stripes, a definition value for each particular path or portion of the particular path in the collection of data records. The definition level for the particular path or portion of the particular path identifies a quantity of data elements included in the particular path or portion of path.

[0014] Embodiment 7 is directed to the method of any one of embodiments 1 through 6. The method further comprises receiving, by the computing system, information from a collection of data sources, each data source including information that is unstructured according to a schema. The method further comprises generating, by the computing system, each data record in the

collection of data records by structuring the information in each data source according to the schema.

[0015] Embodiment 8 is directed to the method of any one of embodiments 1 through 7. The method further comprises performing, by the computing system, a query of the collection of columnar stripes. The method further comprises outputting, by the computing system and in response to performance of the query, a new columnar stripe that includes a subset of the values from a columnar stripe from the collection of columnar stripes that are identified by the query.

[0016] Embodiment 9 is directed to the method of embodiment 8, wherein the query of the collection of columnar stripes is performed without loading the data values included in the collection of columnar stripes into a database.

[0017] Embodiment 10 is directed to the method of any one of embodiments 1 through 9, wherein at least a first columnar stripe of the collection of columnar stripes includes multiple blocks of data, at least some of the multiple blocks of data each including an assertion value that defines a kind of values that are found in each block's values, so that the computing system, when performing a query of the first columnar stripe, avoids one or more blocks of data that do not include a data value that is specified by the query.

[0018] Embodiment 11 is directed to the method of any one of embodiments 1 through 10, wherein a first columnar stripe of the collection of columnar stripes includes, as first data values, all data values from among the data records in the collection of data records that correspond to instances of a first data element in the collection of data records. The first columnar stripe stores multiple of the first data values continuously in memory, wherein the

multiple first data values were not stored continuously in memory when the multiple first data values were stored in data records in the collection of data records.

[0019] Embodiment 12 is directed to the method of any one of embodiments 1 through 11, wherein at least one data record in the collection of data records includes data elements and corresponding data values that are stored according to a nested data model.

[0020] Embodiment 13 is directed to the method of any one of embodiments 1 through 12, wherein first data record includes a first data element and a second data element, the first data element being a parent data element to the second data element, and the second data element being a child data element to the first data element. A first columnar stripe includes a data value that corresponds to the first data element in the first data record and all other data values that correspond to instances of the first data element in the collection of data records. A second columnar stripe includes a data value that corresponds to the second data element in the first data record and all other data values that correspond to instances of the second data element in the collection of data records.

[0021] Other embodiments include corresponding computer-readable storage devices storing instructions that, when executed by one or more processing devices (e.g., a programmable computer processor), perform operations according to any one of the above-described embodiments 1 through 13. Other embodiments include systems and apparatus that include the described computer-readable storage devices storing instructions that, when

executed by one or more processing devices, perform operations according to any one of the above-described embodiments 1 through 13.

[0022] The details of one or more embodiments are set forth in the accompanying drawings and the description below. Other features, objects, and advantages will be apparent from the description and drawings, and from the claims.

DESCRIPTION OF DRAWINGS

[0023] Figure 1 illustrates record-wise v. columnar representation of nested data.

[0024] Figure 2 illustrates two sample nested records and their schema.

[0025] Figure 3 illustrates column-striped representations of the sample nested records.

[0026] Figure 4 is an algorithm for dissecting a record into columns.

[0027] Figure 5 illustrates an automaton for performing complete record assembly.

[0028] Figure 6 illustrates an automaton for assembly records from two fields, and the records that the automaton produces.

[0029] Figure 7 is an algorithm for constructing a record assembly automaton.

[0030] Figure 8 is an algorithm for assembling a record from columnar data.

[0031] Figure 9 depicts a sample query that performs projection, selection, and within-record aggregation.

[0032] Figure 10 illustrates a system architecture and execution inside a server node.

[0033] Figure 11 is a table illustrating the datasets used in the experimental study.

[0034] Figure 12 is a graph that illustrates the performance breakdown that may occur when reading from a local disk.

[0035] Figure 13 is a graph that illustrates execution of both MapReduce and the described system on columnar v. record-oriented storage.

[0036] Figure 14 is a graph that illustrates the execution time as a function of serving tree levels for two aggregation queries.

[0037] Figure 15 is a graph that illustrates histograms of processing times.

[0038] Figure 16 is a graph that illustrates execution time when the system is scaled from 1000 to 4000 nodes using a top-k query.

[0039] Figure 17 is a graph that illustrates a percentage of processed tables as a function of processing time per tablet.

[0040] Figure 18 is a graph that illustrates query response time distribution in a monthly workload.

[0041] Figure 19 is a block diagram of a system for generating and processing columnar storage representations of nested records.

[0042] Figure 20 is a flow chart of an example process for generating columnar data.

[0043] Figure. 21 is a block diagram of computing devices that may be used to implement the systems and methods described in this document, as either a client or as a server or plurality of servers.

[0044] Like reference symbols in the various drawings indicate like elements.

DETAILED DESCRIPTION

[0045] This document describes techniques, methods, systems, and mechanisms for generating and processing columnar storage representations of records. As an illustration, an organization may store data from web pages in records of nested information. The nested information may be compiled in a columnar data storage format that enables efficient queries of the data using a multi-level execution tree. The columnar data may be re-assembled into records for input into analysis programs that operate on record-oriented data.

[0046] More specifically, each record may be an instantiation of a schema that defines a formatting of records, where the records are created in accordance with the schema. For example, a schema may identify various fields for storing information about a web page and a structure for organizing fields in a record and their corresponding values. When a record for describing the characteristics of a web page is generated, the record may include for each field a data element and a corresponding value. The data element may define the semantics of the value in accordance with a definition in the schema. The term data element and field may be used interchangeably in this document. Field may also refer to a combination of a data element and a corresponding value.

[0047] A particular record may need not include all of the fields that are defined by a schema. Thus, the schema may serve as a 'template' from which fields may be selected for the particular record. For example, the schema may include a field for defining information about video content in a web page. If a web page does not include video content, then the record corresponding to the web page may not include the field from the schema that defines information about videos on the web page. Thus, some of the fields may be 'optional.'

[0048] Some of the fields in a record, however, may be 'required.' For example, a 'required' field in the schema may be a Uniform Resource Locator (URL) of a source location for the document that served the web page. The field may be required because every web page document may be retrieved from a source location (i.e., there is a URL available for every document) and because the field may be required to further process information on the web page (e.g., to determine if the content has changed).

[0049] A field may also be 'repeatable.' A field that is in the schema and that is defined as repeatable may be replicated at the location defined by the schema repeatedly in an instantiation of the schema (i.e., in a record). For example, a schema may include a field that is for defining documents that link to the web page. The schema may only specify the field a single time, but may indicate that the field is repeatable (e.g., because several documents may link to a particular web page). Thus, a record for the web page may include multiple fields that identify a value for a linking web page. The repeated fields may be located at a same level and nested beneath a parent field in the record (as discussed in more detail below).

[0050] The fields of the schema (and thus the fields in the records) may be nested. In other words, some fields may be children of other fields, which may be referenced as the parent fields, grandparent fields, etc. In some examples, children nodes are those nodes in the schema that are found within a pair of opening and closing curly brackets immediately following the parent node. Other implementations for nesting, however, may be utilized (e.g., the use of a start tag for the field and an end tag for the field). Thus, except for the fields that are at the

highest level (e.g., the fields that are not children of any other fields), each field may have a parent field.

[0051] Nesting may be helpful for organizing information into conceptually-related chunks of information. Returning to our earlier example, the schema may include a 'Video' field. The 'Video' field may include several children fields that may identify the characteristics of the video (e.g., how long the video is, the format of the video, and the resolution of the video). Thus, when a record is constructed, children nodes may not be placed in the record if their parent nodes are not present. In other words, a record for a web page that does not include a video may not include a 'VideoLength' field because the record does not include a 'Video' field (i.e., the parent of the 'VideoLength' field). Application programs that enable viewing and editing a record may visually nest the dependent children off of the parent children (e.g., indent the children to the right of the parent field).

[0052] Analyzing millions of records may be time consuming. In some examples a user is interested in a data from a single field, but each of the records must be accessed in its entirety. For example, a user may request that an analysis program check each of millions of records to identify those records that are associated with web pages that include videos that are longer than ten minutes and that have a 'High' resolution. Because each record may be stored as a separate data structure, each entire record may need to be loaded into a database management system in order to query the record to determine if the record includes the particular combination of video length and resolution.

[0053] Such a loading of every single record may be prohibitively expensive, both on the quantity of servers that are required to perform the task and an amount of time necessary to complete the query. Significant time savings

can be obtained by storing all of the values for a particular field – selected from across the millions of records – together in a contiguous portion of memory. Such storage of values from several records but for a particular field is called columnar storage. In contrast, the example where information for a particular record is stored contiguously in memory is referred to as record-oriented storage.

[0054] Columnar storage for nested records, however, poses unique difficulties. A field in a record may be identified by its path, which may include a listing of the field and the parent fields (e.g., GrandParent.Parent.Child). Because one or more of the fields in the path may be repeating, there may be several instances of a field with the same path name. Thus, when looking at a consecutive listing of columnar data for a particular field, a mechanism is needed to identify which values belong to which records, and for those records that include multiple values for a particular path, what is the respective location of the value in the record. In other words, given a sequence of values in a columnar structure, a mechanism is needed to reconstruct the structure of the record from the values.

[0055] The mechanism for reconstructing the structure of a record from columnar data includes storing, for each value in the columnar data, a ‘repetition’ level and a ‘definition’ level. Each ‘level’ may be a sequence of bits that represents a number. For example, a ‘level’ of 3 may be represented by two bits (e.g., ‘11’). In another example, a ‘level’ of 5 may be represented by three bits (e.g., ‘101’).

[0056] The ‘repetition’ level that is stored for a particular value indicates the field in the value’s path that has most recently repeated. As an illustration, a column of values may be stored for a field with the path ‘Video.Resolution.Width.’

A repetition level of '1' may indicate that the 'Video' field most recently repeated, while a repetition level of '2' may indicate that the 'Resolution' field most recently repeated. Recently repeating can indicate, from the position of the value in the record from which the value was selected and working upwards towards the beginning of the document, which field in the path 'Video.Resolution.Width' is the first to reach a count of two (e.g., which field is encountered for the second time first).

[0057] For example, working upwards from the location of the 'Width' value, each field is encountered a single time. Finding a second instance of each field requires traversing to the depths of the next, adjacent nested field (and possibly to further nestings). Thus, a 'Video' field may be encountered that does not include any 'Resolution' children (e.g., because the 'Resolution' field is optional or a repeating field). Thus, the 'Video' field has been encountered a second time and is thus the most recently repeated field. A repetition level of '1' is assigned to the value.

[0058] A repetition level of '0' may indicate that the field does not include a most recently repeated value (e.g., it has been encountered for the first time in the record during a top-down scan). In various examples, a 'required' field in a path does not have a repetition level. For example, if the 'Resolution' field is required for the 'Video.Resolution.Width' path, the range of resolution levels may be either '0' or '1.' 'Resolution' may not have a level because it is always present in the record when the 'Video' field is present. Thus, if 'Resolution' was assigned a level of '2,' it may always be encountered before 'Video' and thus a level of '1' may not ever be assigned. Thus, not including a repetition level for required fields

may enable a number of different resolution levels to be reduced, and a number of bits to represent the resolution level may be reduced.

[0059] If the field 'Width' in the above example is an 'optional' or 'repeating' field, a record may not always include a value for the 'Width' field. Thus, a column of values for the 'Video.Resolution.Width' path may use a mechanism to designate when a 'Video' or a 'Video.Resolution' path is found in the record but the 'Width' field has not been instantiated in the record. This mechanism may include storing, in the 'Video.Resolution.Width' column of data, a 'Definition' level for each 'Video' or 'Video.Resolution' field in the record regardless whether the 'Width' field is instantiated. The 'Definition' level may indicate how many of the fields in the 'Video.Resolution.Width' path that could be missing (e.g., because the field is optional or repeatable) are actually present.

[0060] Thus, if the field 'Video' is present in the record but no corresponding 'Resolution' child is instantiated, a definition level of '1' may be recorded in the 'Video.Resolution.Width' column. If the field 'Video.Resolution' is present in the record, but no corresponding 'Width' child is instantiated, a definition level of '2' may be recorded. If the field 'Video.Resolution.Width' is present in the record, a definition level of '3' may be recorded.

[0061] As such, a definition level may be generated for each data element path or portion of a data element path in a record. For example, the path 'Video.Resolution.Width' may have a definition level of '3' that identifies the quantity of data elements included in the path 'Video.Resolution.Width'. As another example, the path 'Video.Resolution' may have a definition level of '2' that identifies the quantity of data elements included in the path

'Video.Resolution,' even though the 'Width' data element was not instantiated in the data record.

[0062] Therefore, whenever the 'Definition' level (which represents the number of fields that could be undefined but are actually defined) is less than the number of fields that could be defined, a missing occurrence of the 'Width' field may be identified.

[0063] In various examples, a 'Definition' level is a Boolean value instead of an integer. A Boolean value may be used instead of an integer where knowing which containing object was undefined was not important. For example, a definition level for a value for the Width field of the 'Video.Resolution.Width' path may be 'NULL' where the system does not need to know which of the parent fields are instantiated. The combination of the 'Repetition' level and the 'Definition' level may enable the structure of the record to be reconstructed.

[0064] A column of data for a particular field (e.g., the 'Video.Resolution.Width' field) may include the values for the field from multiple records, corresponding repetition and definition levels (acknowledging that some 'missing' values may have a repetition and a definition level), and header information. In some examples, the values are stored consecutively and adjacent. In other words, if a value for one 'Video.Resolution.Width' field was '700' and the value for a next 'Video.Resolution.Width' field was '800,' a portion of the column as stored in memory may read '700800.' In this example, a header in the column may identify that the each value has a fixed width (e.g., a fixed binary representation to hold the numbers 700 and 800).

[0065] In some examples, the stored values are represented by strings. For example, instances of the 'Width' field may include the values 'Small' and

'Medium.' In some examples, the various string values may be a fixed length (e.g., a null value may be added to the beginning or end of the 'Small' value to make the string the same length as the 'Medium' value). In some examples, however, each stored string may include an identifier in a beginning portion of the string that identifies a length of the string. For example, the 'small' value may include an identifier that indicates that the string is five digits long (or a corresponding number of binary bits).

[0066] Because the values may be stored consecutively in the columnar stripe, the 'repetition' and 'definition' levels may be stored at the beginning of the columnar stripe. In some examples, the 'repetition' and 'definition' levels are stored in pairs for a particular value (whether instantiated or missing). As an illustration, a repetition level of 3 may be stored in the first four bits of a byte and a definition level of 1 may be stored in the last four bits of the byte. A next byte in the header may include a repetition level and a definition level for the next instance of the field in the record (or the first instance in the subsequent record).

[0067] The number of bits used to represent the repetition and definition levels may be based on a maximum level value. For example, if the maximum repetition level is 3, the repetition level may be represented with two bits. If the maximum repetition level is 4 the repetition level may be represented with three bits. The header may include information that identifies the length of the repetition and definition levels.

[0068] In various examples, the repetition levels may be stored consecutively in memory and the definition levels may be stored consecutively in memory (e.g., not in pairs). In various examples, the repetition and definition levels may be stored in a group with their corresponding value (if the value is

instantiated). In other words, a sequence of information in the columnar stripe may read

Value1:RepetitionLevel1:DefinitionLevel1:Value2:RepetitionLevel2:DefinitionLevel2, and so on.

[0069] The columnar stripes may be compressed into blocks of information. For example, each columnar stripe may be split into a set of blocks, with each block including its own respective header. A first block may include the first 800,000 values and a second block may include a second 800,000 values from a stripe of 1.6 million values. A block header may include the repetition and definition levels along with additional information that may be used to help analyze the portion of the columnar stripe that is represented by the block, and to reconstruct the columnar stripe.

[0070] In some examples, the block header includes an 'Assertion' value that defines a type of data that is found in the block's values. For example, a block for the 'Video.Resolution.Width' field may not include any values that list 'Large' width resolution. Thus, the 'Assertion' value may indicate that the values only include 'Small' and 'Medium' values. If a query is performed for records that include 'High' width resolution videos, then the described block may be avoided by the querying system.

[0071] The system described in this document may perform queries on columnar stripes without reconstructing the information in the columnar stripes into records, and without loading information from the columnar stripes into a database (e.g., without using 'Insert' clause). Thus, the data may be accessed *in situ*, which may provide computational analysis time savings on the order of magnitudes.

[0072] The querying system may employ many of the clauses employed for querying relational databases. Additional clauses that are specific to non-relational data, however, may be employed. For example, a WITHIN clause may allow for operations to be performed on multiple instances of a field within a single record or a portion of a record. A relational database, however, may be unable to store more than a single instance of a field in a row (e.g., a representation of a record). Thus, a query on a relational database may be fundamentally unable to perform queries 'within' a record.

[0073] As an example of the WITHIN clause, values for a particular field may be multiplied. Supposing that the query instructions request that all values for 'MutualFund.InterestRate' be multiplied together for a particular record (where each record may be for a particular account holder). The querying system may find all of the 'MutualFund.InterestRate' values within the single record and multiply them together.

[0074] Another example of a clause that may be specific to non-relational nested data is the OMIT IF clause. This clause may enable a record to be filtered to remove instances of fields if a particular condition is met (e.g., a new columnar stripe or record may be created with specified fields removed). As an illustration, a stripe of values that list employee salaries may be queried and a new stripe that removes employees with salaries above \$90,000 may be generated using the OMIT IF clause.

1. INTRODUCTION

[0075] Large-scale parallel computing may be performed using shared clusters of commodity machines. See L. A. Barroso and U. Holzle. *The*

Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan & Claypool Publishers, 2009. A cluster may host a multitude of distributed applications that share resources, have widely varying workloads, and run on machines with different hardware parameters. An individual computing machine in a distributed application may take much longer to execute a given task than others, or may never complete due to failures or preemption by a cluster management system. Hence, dealing with stragglers (e.g., computing tasks with significant latency) and failures is may achieve fast execution and fault tolerance. See G. Czajkowski. Sorting 1PB with MapReduce. Official Google Blog, Nov. 2008. At <http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html>.

[0076] The data used in web and scientific computing is often nonrelational. Hence, a flexible data model may be beneficial in these domains. Data structures used in programming languages, messages exchanged by distributed systems, web traffic logs, etc. may lend themselves to a nested representation. For example, a nested representation of data may include a multiple fields that each include several levels of children fields. Some of the children fields may include corresponding data. Normalizing and recombining such data at web scale may be computationally expensive. A nested data model underlies some of the structured data processing at major web companies.

[0077] This document describes a system that supports interactive analysis of very large datasets over shared clusters of commodity machines. Unlike traditional databases, it is capable of operating on *in situ* nested data. *In situ* refers to the ability to access data 'in place', for example, in a distributed file system like Google File System (see S. Ghemawat, H. Gobioff, and S.-T. Leung.

The Google File System. In *SOSP*, 2003) or another storage layer like Bigtable (see F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006).

[0078] The system can execute many queries over such data that may ordinarily require a sequence of MapReduce jobs, but at a fraction of the execution time. See J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004. The described system may be used in conjunction with MapReduce to analyze outputs of MapReduce pipelines or rapidly prototype larger computations. Examples of using the system include:

- Analysis of web logs and crawled web documents;
- Install data for applications served by an online marketplace;
- Crash data for application products;
- Multimedia playback statistics;
- OCR results from scans of books;
- Spam analysis;
- Debugging of map tiles;
- Tablet migrations in managed Bigtable instances;
- Results of tests run on a distributed build system;
- Disk I/O statistics for hundreds of thousands of disks;
- Execution logs of MapReduce jobs across several data centers; and
- Symbols and dependencies in a codebase.

[0079] The described system builds on ideas from web search and parallel database management systems. First, its architecture builds on the concept of a serving tree used in distributed search engines. See J. Dean. Challenges in

Building Large-Scale Information Retrieval Systems: Invited Talk. In *WSDM*, 2009. Like a web search request, a query gets pushed down the tree and rewritten at each step. The result of the query is assembled by aggregating the replies received from lower levels of the tree.

[0080] Second, the described system provides a high-level, SQL-like language to express ad hoc queries. In contrast to layers such as Pig (see C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-so-Foreign Language for Data Processing. In *SIGMOD*, 2008.) and Hive (Hive. <http://wiki.apache.org/hadoop/Hive>, 2009), the querying system executes queries natively without translating them into MapReduce jobs.

[0081] Lastly, the described system uses a column-striped storage representation, which enables it to read less data from secondary storage and reduce CPU cost due to cheaper compression. Column stores for analyzing relational data, (D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-Oriented Database Systems. *VLDB*, 2(2), 2009), are not believed to have extended to nested data models. The columnar storage format that is described may be supported by MapReduce, Sawzall (see R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4), 2005), and FlumeJava (see C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*, 2010).

[0082] In Section 4, this document describes a columnar storage format for nested data. Algorithms are presented for dissecting nested records into columns and reassembling them.

[0083] In Section 5, a query language for processing data in that is stored in the columnar storage format is described. The query language and execution of the language are designed to operate efficiently on column-striped nested data and do not require restructuring of nested records.

[0084] In Section 6, an illustration of applying execution trees that are used in web search serving systems to database processing is provided. The benefits for answering aggregation queries efficiently is explained.

[0085] In Section 7, experiments conducted on system instances are presented.

SECTION 2: EXAMPLE SCENARIO

[0086] Suppose that Alice, an engineer at a web-search company, comes up with an idea for extracting new kinds of signals from web pages. She runs a MapReduce job that cranks through the input data that includes content from the web pages and produces a dataset containing the new signals, stored in billions of records in a distributed file system. To analyze the results of her experiment, she launches the system described in this document and executes several interactive commands:

```
DEFINE TABLE t AS /path/to/data/*
```

```
SELECT TOP(signal1, 100), COUNT(*) FROM t
```

[0087] Alice's commands execute in seconds. She runs a few other queries to convince herself that her algorithm works. She finds an irregularity in signal1 and digs deeper by writing a FlumeJava program that performs a more complex analytical computation over her output dataset. Once the issue is fixed, she sets up a pipeline which processes the incoming input data continuously. She

formulates a few canned SQL queries that aggregate the results of her pipeline across various dimensions, and adds them to an interactive dashboard (e.g., a web page about a service that explains the service and details statistics on the service). Finally, she registers her new dataset in a catalog so other engineers can locate and query the dataset quickly.

[0088] The above scenario may require interoperation between the query processor and other data management tools. The first ingredient for such interoperation is a common storage layer. The Google File System is one such distributed storage layer that may be used. The Google File System manages very large replicated datasets across thousands of machines and tens of thousands of disks.

[0089] Replication helps preserve the data despite faulty hardware and achieve fast response times in presence of stragglers. A high-performance shared storage layer is a key enabling factor for *in situ* data management. It allows accessing the data without a time-consuming loading phase, which is a major impedance to database usage in analytical data processing (where it is often possible to run dozens of MapReduce analyses before a database management system is able to load the data and execute a single query). For example, when a database management system is used to analyze data, the database may need to be loaded with data using 'Insert' commands. Such loading may not be required by the described system. As an added benefit, data in a file system can be conveniently manipulated using standard tools, e.g., to transfer to another cluster, change access privileges, or identify a subset of data for analysis based on file names.

[0090] A second ingredient for building interoperable data management components is a shared storage format. Columnar storage is used for flat relational data but adapting columnar storage to a nested data model allows the technique to be applied to web data. Figure 1 illustrates the idea that all values of a nested field in a data structure are stored contiguously. For example, in the column-oriented representation of nested data, all values for a particular nested field within a data structure (e.g., the field A.B.C) are stored adjacent to each other and contiguously in memory. Hence, values for the field A.B.C can be retrieved from memory without reading values from the field A.E and values from the field A.B.D.

[0091] Additionally, values for the same particular field in different instances of a data structure (e.g., a 'record') may be stored contiguously. For example, the values for field A.B.C for the record 'r1' are stored adjacent to the values for the same field for the record 'r2.' To the contrary, in the 'record-oriented' representation of nested data, values for all fields within a particular record are stored contiguously. In other words, the data values for a particular field are not bunched together.

[0092] The challenge that the described columnar storage format addresses is how to preserve all structural information and be able to reconstruct records from an arbitrary subset of fields. This document next discusses the data model from which the fields in the columnar storage format may be filled, and then turn to algorithms for processing the columnar storage and query processing on data in the columnar storage.

SECTION 3: DATA MODEL

[0093] This section describes the data model used by the described system and introduces some terminology used later. The described Protocol Buffers data model originated in the context of distributed systems, and is available as an open source implementation. See (Protocol Buffers: Developer Guide. Available at <http://code.google.com/apis/protocolbuffers/docs/overview.html>). The data model is based on strongly-typed nested records. Its abstract syntax is given by:

$$\tau = dom \mid \langle A_1 : \tau [*?], \dots, A_n : \tau [*?] \rangle$$

where τ is an atomic type or a record type. Atomic types in dom comprise integers, floating-point numbers, strings, etc. Records consist of one or multiple fields. Field i in a record has a name A_i and an optional multiplicity label. Repeated fields (*) may occur multiple times in a record. They are interpreted as lists of values, i.e., the order of field occurrences in a record is significant. Optional fields (?) may be missing from the record. Otherwise, a field is required (e.g., must appear exactly once).

[0094] As an illustration, Figure 2 depicts a schema that defines a record type 'Document,' which represents a web document. The schema definition uses the Protocol Buffers syntax. A 'Document' has a required integer 'DocId' and optional 'Links,' containing a list of 'Forward' and 'Backward' entries holding 'DocIds' of other web pages. A 'Document' can have multiple 'Names,' which are different URLs by which the document can be referenced. A 'Name' contains a sequence of 'Code' and (optional) 'Country' pairs. Figure 2 also shows two sample records, r1 and r2, that conform to the schema. The record structure is outlined using indentation. The sample records r1 and R2 in Figure 2 are used explain the algorithms throughout this document. The fields defined in the

schema form a tree hierarchy. The full path of a nested field is denoted using a dotted notation, e.g., Name.Language.Code is the full path name for the 'Code' field depicted in Figure 2.

[0095] The nested data model backs a platform-neutral, extensible mechanism for serializing structured data. Code generation tools produce bindings for different programming languages such as C++ or Java. Cross-language interoperability is achieved using a standard binary on-the-wire representation of records, in which field values are laid out sequentially as they occur in the record. This way, a MapReduce program written in Java can consume records from a data source exposed via a C++ library. Thus, if records are stored in a columnar representation, assembling them fast may assist interoperation with MapReduce and other data processing tools.

SECTION 4: NESTED COLUMNAR STORAGE

[0096] As illustrated in Figure 1, a goal is to store all values of a given field consecutively to improve retrieval efficiency. In this section, the challenges of lossless representation of record structure in a columnar format (Section 4.1), fast encoding (Section 4.2), and efficient record assembly (Section 4.3) are addressed.

SECTION 4.1: REPETITION AND DEFINITION LEVELS

[0097] A consecutive list of values alone does not convey the structure of a record. Given two values of a field that is repeated in a record, a system may not be able to determine at what 'level' the value is repeated (e.g., whether the two values are from different records or are from the same record). Likewise, if an

optional field is missing from a record, values alone may not convey which enclosing records were defined explicitly and which were not. The concepts of repetition and definition levels are thus introduced. Figure 3 includes tables that summarize the repetition and definition levels for atomic fields in the sample records that are depicted in Figure 1.

[0098] Repetition Levels. Consider the field 'Code' in Figure 2. It occurs three times in record 'r1.' Occurrences 'en-us' and 'en' are inside the first 'Name' field, while 'en-gb' is in the third 'Name' field. To disambiguate these occurrences in the columnar structure, a repetition level is attached to each value that is to be stored in the columnar structure. The repetition level indicates at what repeated field in the field's path the value has repeated. For example, the field path Name.Language.Code contains two fields that are repeated, 'Name' and 'Language.' Hence, the repetition level of Code ranges between 0 and 2. Level 0 denotes the start of a new record, level 1 denotes a recent repetition at the 'Name' field, and level 2 denotes a recent repetition at the 'Language' field.

[0099] As an illustration of determining the level for a field, record 'r1' may be scanned from the top down. The value 'en-us' is first encountered and a check may be performed to identify the field in the Name.Language.Code path that has most recently repeated in the record. In this example, none of the fields have been repeated and thus, the repetition level is 0. The value 'en' is next encountered for the Name.Language.Code path and the field 'Language' is identified as the field that has most recently repeated. For example, scanning upwards from the value 'en,' the first field in the Name.Language.Code path that repeats is 'Language.' Thus, the repetition level is 2 (e.g., because '2' corresponds to the 'Language' field because 'Language' is the second field in the

Name.Language.Code path that repeats). Finally, when the value 'en-gb' is encountered, the field 'Name' has repeated most recently (the 'Language' field occurred only once after Name), so the repetition level is 1. In other words, the repetition level for a value may be a number that represents a most recently repeated field. Thus, the repetition levels of Code values in record 'r1' are 0, 2, 1.

[00100] Notice that the second 'Name' field in record 'r1' does not contain any values for the field 'Code.' To determine that 'en-gb' occurs as a value for a field nested within the third instance of the field 'Name,' and not in the second instance, a NULL value is added between the values 'en' and 'en-gb' as they are stored in the columnar structure (see Figure 3). 'Code' is a required child field of the 'Language' field, so the fact that a value for the 'Code' field is missing implies that the 'Language' field is not also not defined. In general though, determining the level up to which nested records exist may require additional information.

[00101] Definition Levels. Each value of a field with path 'p,' especially every NULL value, has a 'definition level' that specifies how many fields in the path 'p' that could be undefined (e.g., because the fields are optional or repeated) are actually present in the record. To illustrate, observe that record 'r1' has no 'Backward' fields for the 'Links' field. Still, the field 'Links' is defined (at a level of 1). To preserve this information, a NULL value with definition level of 1 is added to the 'Links.Backward' column.

[00102] In other words, specifying a level of 1 for the 'Links.Backward' path indicates that '1' field that was optional or repeated (i.e., the 'Links' field) was defined in a path that includes two fields that are optional or repeated (i.e., the 'Links' field and the 'Backward' field). Thus, a definition of '1' indicates that the 'Backward' field was not instantiated. Similarly, the missing occurrence of

'Name.Language.Country' in record 'r2' carries a definition level 1, while its missing occurrences in record 'r1' have definition levels of 2 (inside 'Name.Language') and 1 (inside 'Name'), respectively. The encoding procedure outlined above may preserve the record structure losslessly.

[00103] Encoding. As stored in memory, each column that corresponds to a particular field may be stored with a header that includes a contiguous listing of repetition and definition values, followed by a contiguous listing of the substantive values. Each repetition and definition value may be stored as bit sequences (e.g., in a single byte). For example, the first four bits of a byte may be used to represent the repetition level for a particular value and the last four bits may be used to represent the definition level. In some examples, the header may include definitions of lengths of the number of bits so that delimiters may not be used. Thus, bits may only be used as necessary. For example, if the maximum definition level is 3, two bits per definition level may be used.

[00104] Thus, a representation of columnar data for a single field (e.g., the 'Name.Language.Code' field) may be stored in memory with a sequence of bytes representing the repetition and definition levels for a corresponding sequence of values, followed by a sequence of values. NULL values, however, may not be stored explicitly as they may be determined by analyzing the definition levels. For instance, any definition level that is smaller than the number of repeated and optional fields in a field's path can denote a NULL. Thus, a system may be able to determine where in the listing of consecutive values a NULL value should be inserted or inferred. In some examples, definition levels are not stored for values that are always defined. Similarly, repetition levels may only be stored if required. For example, a definition level of 0 implies a repetition level of 0, so the latter may

be omitted. In fact, referencing the structures illustrated in Figure 3, no levels may be stored for the 'DocId' field.

[00105] A representation of columnar data in memory may be broken up into a set of blocks. Each block may include a header that includes the repetition and definition level information, and a subsequent listing of the values for the field. Each header may include a 'constraint' value that indicates an allowable range of values in the block. Thus, the described system may identify which blocks include data that the system is interested in. The constraint can also indicate other properties of the values, e.g., whether the values have been sorted. In general, the 'constraint' may be thought of as an 'assertion' about what kind of values are found in the block. Each block may be compressed.

SECTION 4.2: SPLITTING RECORDS INTO COLUMNS

[00106] The above description presented an encoding of the record structure in a columnar format. A challenge is how to produce column stripes with repetition and definition levels efficiently. The base algorithm for computing repetition and definition levels is provided below. The algorithm recurses into the record structure and computes the levels for each field value. As illustrated earlier, repetition and definition levels may need to be computed even if field values are missing. Many datasets are sparse and it may not be uncommon to have a schema with thousands of fields, only a hundred of which are used in a given record. Hence, it may be beneficial to process missing fields as cheaply as possible. To produce column stripes, a tree of field writers is created, whose structure matches the field hierarchy in the schema. The basic idea is to update field writers only when they have their own data, and not try to propagate parent

state down the tree unless absolutely necessary. To do that, child writers inherit the levels from their parents. A child writer synchronizes to its parent's levels whenever a new value is added.

[00107] An example algorithm for decomposing a record into columns is shown in Figure 4. Procedure 'DissectRecord' is passed an instance of a 'RecordDecoder,' which is used to traverse binary-encoded records. 'FieldWriters' form a tree hierarchy isomorphic to that of the input schema. The root 'FieldWriter' is passed to the algorithm for each new record, with 'repetitionLevel' set to 0. The primary job of the 'DissectRecord' procedure is to maintain the current 'repetitionLevel.' The current 'definitionLevel' is uniquely determined by the tree position of the current writer, as the sum of the number of optional and repeated fields in the field's path.

[00108] The while-loop of the algorithm (Line 5) iterates over all atomic and record-valued fields contained in a given record. The set 'seenFields' tracks whether or not a field has been seen in the record. It is used to determine what field has repeated most recently. The child repetition level 'chRepetitionLevel' is set to that of the most recently repeated field or else defaults to its parent's level (Lines 9-13). The procedure is invoked recursively on nested records (Line 18).

[00109] The document above referenced 'FieldWriters' accumulating levels and propagating them lazily to lower-level writers. This may be performed by each non-leaf writer keeping a sequence of (repetition, definition) levels. Each writer also has a 'version' number associated with it. Simply stated, a writer version is incremented by one whenever a level is added. It is sufficient for children to remember the last parent's version they synced. If a child writer ever

gets its own (non-null) value, it synchronizes its state with the parent by fetching new levels, and only then adds the new data.

[00110] Because input data may have thousands of fields and millions records, it may not be feasible to store all levels in memory. Some levels may be temporarily stored in a file on disk. For a lossless encoding of empty (sub)records, non-atomic fields (such as Name.Language in Figure 2) may need to have column stripes of their own, containing only levels but no non-NULL values.

SECTION 4.3: RECORD ASSEMBLY

[00111] Assembling records (e.g., records 'r1' and 'r2') from columnar data efficiently is critical for record-oriented data processing tools (e.g., Map Reduce). Given a subset of fields, a goal is to reconstruct the original records as if they contained just the selected fields, with all other fields stripped away. The key idea is to create a finite state machine (FSM) that reads the field values and levels for each field, and appends the values sequentially to the output records. An FSM state corresponds to a field reader for each selected field. State transitions are labeled with repetition levels. Once a reader fetches a value, the next repetition level is looked at to decide what next reader to use. The FSM is traversed from the start to end state once for each record.

[00112] Figure 5 shows an FSM that reconstructs the complete records in our running example using as input the blocks described in Section 4.1. In this example, the nodes are labeled with fields and the edges are labeled with repetition levels. The start state is 'DocId.' Once a 'DocId' value is read, the FSM

transitions to the 'Links.Backward' state. After all repeated 'Backward' values have been drained, the FSM jumps to 'Links.Forward,' etc.

[00113] To sketch how FSM transitions are constructed, let 'l' be the next repetition level returned by the current field reader for field 'f.' Starting at 'f' in the schema tree (e.g., the schema in Figure 2), its ancestor is found that repeats at level 'l' and select the first leaf field 'n' inside that ancestor. This provides an FSM transition ('f'; 'l') → n. For example, let 'l' = 1 be the next repetition level read by 'f' = 'Name.Language.Country.' Its ancestor with repetition level '1' is Name, whose first leaf field is 'n' = 'Name.Url.'

[00114] If only a subset of fields need to be retrieved, a simpler FSM that is cheaper to execute may be constructed. Figure 6 depicts an FSM for reading the fields 'DocId' and 'Name.Language.Country.' The figure shows the output records 's1' and 's2' produced by the automaton. Notice that the encoding and the assembly algorithm preserve the enclosing structure of the field 'Country.' This may be important for applications that need to access, e.g., the Country appearing in the first Language of the second Name. In XPath, this may correspond to the ability to evaluate expressions like /Name[2]/Language[1]/Country.

[00115] Construct FSM Procedure. Figure 7 shows an algorithm for constructing a finite-state machine that performs record assembly. The algorithm takes as input the fields that should be populated in the records, in the order in which they appear in the schema. The algorithm uses a concept of a 'common repetition level' of two fields, which is the repetition level of their lowest common ancestor. For example, the common repetition level of 'Links.Backward' and 'Links.Forward' equals 1. The second concept is that of a 'barrier', which is the

next field in the sequence after the current one. The intuition is that each field is attempted to be processed one by one until the barrier is hit and requires a jump to a previously seen field.

[00116] The algorithm consists of three steps. In Step 1 (Lines 6-10), the common repetition levels are processed backwards. These are guaranteed to be non-increasing. For each repetition level encountered, the left-most field in the sequence is picked—that is the field that is to be transitioned to when that repetition level is returned by a ‘FieldReader.’ In Step 2, the gaps are filled (Lines 11-14). The gaps arise because not all repetition levels are present in the common repetition levels computed at Line 8. In Step 3 (Lines 15-17), transitions for all levels are set that are equal to or below the barrier level to jump to the barrier field. If a ‘FieldReader’ produces such a level, the nested record may continue to be constructed and there may be no need to bounce off the barrier.

[00117] Assemble Record Procedure. An Assemble Record procedure (illustrated in Figure 8) takes as input a set of ‘FieldReaders’ and (implicitly) the FSM with state transitions between the readers. In other words, the algorithm operates on an FSM and columnar data and outputs constructed records. Variable reader holds the current ‘FieldReader’ in the main routine (Line 4). Variable Reader holds the last reader whose value is appended to the record and is available to all three procedures shown in Figure 7. The main while-loop is at Line 5. The next value is fetched from the current reader. If the value is not NULL, which is determined by looking at its definition level, the record being assembled is synchronized to the record structure of the current reader in the method ‘MoveToLevel,’ and the field value is appended to the record. Otherwise, the record structure may be adjusted without appending any value—which may

be done if empty records are present. On Line 12, a 'full definition level' is used. Recall that the definition level factors out required fields (only repeated and optional fields are counted). Full definition level takes all fields into account.

[00118] Procedure 'MoveToLevel' transitions the record from the state of the 'lastReader' to that of the 'nextReader' (see Line 22). For example, suppose the 'lastReader' corresponds to 'Links. Backward' in Figure 2 and 'nextReader' is 'Name.Language.Code.' The method ends the nested record Links and starts new records Name and Language, in that order. Procedure 'ReturnsToLevel' (Line 30) is a counterpart of 'MoveToLevel' that only ends current records without starting any new ones.

[00119] In their on-the-wire representation, records are laid out as pairs of a field identifier followed by a field value. Nested records can be thought of as having an 'opening tag' and a 'closing tag', similar to XML (actual binary encoding may differ). A description of 'starting' a record refers to writing opening tags, while 'ending' a record refers to writing closing tags.

SECTION 5: QUERY LANGUAGE

[00120] The described system may employ a query language that is based on SQL and is designed to be efficiently implementable on columnar nested storage. Aspects of the query language are described herein. Each SQL-like statement (and algebraic operators it translates to) takes as input one or multiple nested tables (e.g., a set of compressed blocks of columnar data that represents a table, as described in Section 4.1) and their schemas, and produces a nested table (e.g., a modified instance of the columnar data) and its output schema. Figure 9 depicts a sample query that performs projection, selection, and within-

record aggregation. The query is evaluated over the table $t = \{r1, r2\}$ from Figure 2. The fields are referenced using path expressions. The query produces a nested result although no record constructors are present in the query.

[00121] To explain what the query does, consider the selection operation (the WHERE clause). Think of a nested record as a labeled tree, where each label corresponds to a field name. The selection operator prunes away the branches of the tree that do not satisfy the specified conditions. Thus, only those nested records are retained where 'Name.Url' is defined and starts with 'http.' Next, consider projection. Each scalar expression in the SELECT clause emits a value at the same level of nesting as the most-repeated input field used in that expression. So, the string concatenation expression emits 'Str' values at the level of 'Name.Language.Code' in the input schema.

[00122] The COUNT expression illustrates within-record aggregation. The aggregation is done WITHIN each 'Name' subrecord, and emits the number of occurrences of 'Name.Language.Code' for each 'Name' as a non-negative 64-bit integer (uint64). Thus, the WITHIN statement enables intra-row aggregation. In other words, records of the same name may be aggregated in a same record or beneath a same child. In contrast, SQL, which may not be able to operate on nested data, may be unable to operate on intra-row records.

[00123] The language supports nested subqueries, inter and intra-record aggregation, top-k, joins, user-defined functions, etc. Some of these features are discussed in the experimental section. As one additional example, the described query language includes an OMIT IF statement that can filter an intra-row group of values. For example, each of thousands of records may include several repeated 'Cost' fields that each includes a numerical value. A user of the query

language may want to throw out all records where a sum of the values in the fields exceeds the number '20.' Thus, the user may employ an OMIT IF statement to generate a list of the records where the summed 'Cost' in each record is twenty or less.

SECTION 6: QUERY EXECUTION

[00124] Tree Architecture. The described system uses a multi-level serving tree to execute queries (see Figure 10). A root server receives incoming queries, reads metadata from the tables, and routes the queries to the next level in the serving tree. The leaf servers communicate with the storage layer or access the data on local disk. Many of the queries that operate in the described system are single-scan aggregations; therefore, this document focuses on explaining those and uses them for experiments in the next section. Consider a simple aggregation query below:

SELECT A, COUNT(B) FROM T GROUP BY A

[00125] When the root server receives the above query, it determines all tablets, i.e., horizontal partitions of the table, that comprise the table 'T' and rewrites the query as follows:

SELECT A, SUM(c) FROM (R_1^1 UNION ALL ... R_n^1) GROUP BY A

[00126] Tables R_1^1 UNION ALL ... R_n^1 are the results of queries sent to the nodes 1, . . . , n at level 1 of the serving tree:

R_i^1) = SELECT A, COUNT(B) AS c FROM T_i^1 GROUP BY A

[00127] T_i^1 is a disjoint partition of tablets in 'T' processed by server 'I' at level '1.' Each serving level performs a similar rewriting. Ultimately, the queries reach the leaves, which scan the tablets in 'T' in parallel. On the way up,

intermediate servers perform a parallel aggregation of partial results. The execution model presented above is well suited for aggregation queries returning small and medium-sized results, which are a very common class of interactive queries.

[00128] Query Dispatcher. The described system is a multi-user system, e.g., several queries may be executed simultaneously. A query dispatcher schedules queries based on their priorities and balances the load. Another role is to provide fault tolerance when one server becomes much slower than others or a tablet replica becomes unreachable.

[00129] The amount of data processed in each query is often larger than the number of processing units available for execution, which are called slots. A slot corresponds to an execution thread on a leaf server. For example, a system of 3,000 leaf servers each using 8 threads has 24,000 slots. So, a table spanning 100,000 tablets can be processed by assigning about 5 tablets to each slot. During query execution, the query dispatcher computes a histogram of tablet processing times. If a tablet takes a disproportionately long time to process, the system reschedules the tablet on another server. Some tablets may need to be redispached multiple times.

[00130] The leaf servers read stripes of nested data in columnar representation. The blocks in each stripe are prefetched asynchronously; the read-ahead cache typically achieves hit rates of 95%. Tablets are usually three-way replicated. When a leaf server cannot access one tablet replica, it falls over to another replica.

[00131] The query dispatcher honors a parameter that specifies the minimum percentage of tablets that must be scanned before returning a result.

As described below, setting such parameter to a lower value (e.g., 98% instead of 100%) can often speed up execution significantly, especially when using smaller replication factors.

[00132] Each server may have an internal execution tree, as depicted on the right-hand side of Figure 7. The internal tree corresponds to a physical query execution plan, including evaluation of scalar expressions. Optimized, type-specific code is generated for most scalar functions. A basic execution plan consists of a set of iterators that scan input columns in lockstep and emit results of aggregates and scalar functions annotated with the correct repetition and definition levels, bypassing record assembly entirely during query execution.

[00133] Some queries by the described system, such as top-k and count-distinct, return approximate results using well-known single-scan algorithms.

SECTION 7: EXPERIMENTAL DATA

[00134] This section presents an experimental evaluation of the described system on several datasets, and examines the effectiveness of columnar storage for nested data. The properties of the datasets used in the study are summarized in Figure 11. In uncompressed, non-replicated form the datasets occupy about a petabyte of space. All tables are three-way replicated, except one two-way replicated table, and contain from 100K to 800K tablets of varying sizes. This section begins by examining the basic data access characteristics on a single machine, then show how columnar storage benefits MapReduce execution, and finally focus on the described system's performance. The experiments were conducted on system instances running in two data centers next to many other

applications, during regular business operation. Table and field names used below are anonymized.

[00135] Local Disk. In the first experiment, performance tradeoffs of columnar vs. record-oriented storage were examined by scanning a 1GB fragment of table T1 containing about 300K rows (see Figure 12). The data is stored on a local disk and takes about 375MB in compressed columnar representation. The record-oriented format uses heavier compression yet yields about the same size on disk. The experiment was done on a dual-core Intel machine with a disk providing 70MB/s read bandwidth. All reported times are cold; OS cache was flushed prior to each scan.

[00136] Figure 12 shows five graphs, illustrating the time it takes to read and uncompress the data, and assemble and parse the records, for a subset of the fields. Graphs (a)-(c) outline the results for columnar storage. Each data point in these graphs was obtained by averaging the measurements over 30 runs, in each of which a set of columns of a given cardinality was chosen at random. Graph (a) shows reading and decompression time. Graph (b) adds the time needed to assemble nested records from the columns. Graph (c) shows how long it takes to parse the records into strongly typed C++ data structures.

[00137] Graphs (d)-(e) depict the time for accessing the data on record-oriented storage. Graph (d) shows reading and decompression time. A bulk of the time is spent in decompression; in fact, the compressed data can be read from the disk in about half the time. As Graph (e) indicates, parsing adds another 50% on top of reading and decompression time. These costs are paid for all fields, including the ones that are not needed.

[00138] When few columns are read, the gains of columnar representation may be about an order of magnitude. Retrieval time for columnar nested data may grow linearly with the number of fields. Record assembly and parsing may be expensive, each potentially doubling the execution time. Similar trends were observed on other datasets. A natural question to ask is where the top and bottom graphs cross, i.e., record-wise storage starts outperforming columnar storage. In experience, the crossover point may lie at dozens of fields but varies across datasets and depends on whether or not record assembly is required.

[00139] Map Reduce and the Described System. Next an execution of MapReduce and the described system are illustrated on columnar vs. record-oriented data. In this case, a single field is accessed and the performance gains are the most pronounced. Execution times for multiple columns can be extrapolated using the results of Figure 12. In this experiment, the average number of terms in a field 'txtField' of table 'T1' is counted. MapReduce execution is done using the following Sawzall program:

```
numRecs: table sum of int;
numWords: table sum of int;
emit numRecs <- 1;
emit numWords <- CountWords(input.txtField);
```

[00140] The number of records is stored in the variable 'numRecs.' For each record, 'numWords' is incremented by the number of terms in 'input.txtField' returned by the 'CountWords' function. After the program runs, the average term frequency can be computed as numWords=numRecs. In SQL, this computation is expressed as:

```
Q1: SELECT SUM(CountWords(textile)) / COUNT(*) FROM T1
```

[00141] Figure 13 shows the execution times of two MapReduce jobs and the described system on a logarithmic scale. Both MapReduce jobs are run on 3000 workers (e.g., servers). Similarly, a 3000-node instance of the present system is used to execute Query Q₁. The described system and MapReduce-on-columns read about 0.5TB of compressed columnar data vs. 87TB read by MapReduce-on-records. As Figure 12 illustrates, MapReduce gains an order of magnitude in efficiency by switching from record-oriented to columnar storage (from hours to minutes). Another order of magnitude is achieved by using the described system (going from minutes to seconds).

[00142] Serving tree topology. In the next experiment, the impact of the serving tree depth on query execution times is illustrated. Two GROUP BY queries are performed on Table T2, each executed using a single scan over the data. Table T2 contains 24 billion nested records. Each record has a repeated field item containing a numeric amount. The field item.amount repeats about 40 billion times in the dataset. The first query sums up the item amount by country:

```
Q2: SELECT country, SUM(item.amount) FROM T2
      GROUP BY country
```

[00143] It returns a few hundred records and reads roughly 60GB of compressed data from disk. The next query performs a GROUP BY on a text field domain with a selection condition. It reads about 180GB and produces around 1.1 million distinct domains:

```
Q3: SELECT domain, SUM(item.amount) FROM T2
      WHERE domain CONTAINS '.net'
      GROUP BY domain
```

[00144] Figure 14 shows the execution times for each query as a function of the server topology. In each topology, the number of leaf servers is kept constant at 2900 so that the same cumulative scan speed may be assumed. In the 2-level topology (1:2900), a single root server communicates directly with the leaf servers. For 3 levels, a 1:100:2900 setup is used, i.e., an extra level of 100 intermediate servers. The 4-level topology is 1:10:100:2900.

[00145] Query Q_2 runs in 3 seconds when 3 levels are used in the serving tree and does not benefit much from an extra level. In contrast, the execution time of Q_3 is halved due to increased parallelism. At 2 levels, Q_3 is off the chart, as the root server needed to aggregate near-sequentially the results received from thousands of nodes. This experiment illustrates how aggregations returning many groups may benefit from multi-level serving trees.

[00146] Per-tablet Histograms. The Figure 15 shows how fast tablets get processed by the leaf servers for a specific run of Q_2 and Q_3 . The time is measured starting at the point when a tablet got scheduled for execution in an available slot, i.e., excludes the time spent waiting in the job queue. This measurement methodology factors out the effects of other queries that are executing simultaneously. The area under each histogram corresponds to 100%. As Figure 15 indicates, 99% of Q_2 (or Q_3) tablets are processed under one second (or two seconds).

[00147] Within-record Aggregation. As another experiment, the performance of Query Q_4 is examined when run on Table T3. The query illustrates within-record aggregation: it counts all records where the sum of a.b.c.d values occurring in the record are larger than the sum of a.b.p.q.r values. The fields repeat at different levels of nesting. Due to column striping only 13GB (out of

70TB) are read from disk and the query completes in 15 seconds. Without support for nesting, running this query on T3 would be expensive.

```
Q4 : SELECT COUNT(c1 > c2) FROM
      (SELECT SUM(a.b.c.d) WITHIN RECORD AS c1, SUM(a.b.p.q.r)
      WITHIN RECORD AS c2 FROM T3)
```

[00148] Scalability. The following experiment illustrates the scalability of the system on a trillion-record table. Query Q₅ shown below selects top-20 aid's and their number of occurrences in Table T4. The query scans 4.2TB of compressed data.

```
Q5: SELECT TOP(aid, 20), COUNT(*) FROM T4
      WHERE bid = {value1} AND cid = {value2}
```

[00149] The query was executed using four configurations of the system, ranging from 1000 to 4000 nodes. The execution times are in Figure 16. In each run, the total expended CPU time is nearly identical, at about 300K seconds, whereas the user-perceived time decreases near-linearly with the growing size of the system. This result suggests that a larger system can be just as effective in terms of resource usage as a smaller one, yet allows faster execution.

[00150] Stragglers. Stragglers may be tasks (e.g., processing a tablet) that are not performed, for example, because the machine performing the task has an operational problem or the machine is not being aggressive enough in handling the task given higher-priority tasks. Query Q₆ below is run on a trillion-row table T5. In contrast to the other datasets, T5 is two-way replicated. Hence, the likelihood of stragglers slowing the execution is higher since there are fewer opportunities to reschedule the work.

```
Q6: SELECT COUNT(DISTINCT a) FROM T5
```

[00151] Query Q_6 reads over 1TB of compressed data. The compression ratio for the retrieved field is about 10. As indicated in Figure 17, the processing time for 99% of the tablets is below 5 seconds per tablet per slot. However, a small fraction of the tablets take a lot longer, slowing down the query response time from less than a minute to several minutes, when executed on a 2500 node system. The next section summarizes experimental findings.

SECTION 8: OBSERVATIONS

[00152] Figure 18 shows the query response time distribution in a typical monthly workload of the described system, on a logarithmic scale. As Figure 18 indicates, most queries are processed under 10 seconds, well within the interactive range. Some queries have achieved a scan throughput close to 100 billion records per second in a busy cluster, and even higher on dedicated machines. The experimental data presented above suggests the following observations:

- Scan-based queries can be executed at interactive speeds on disk-resident datasets of numerous records;
- Near-linear scalability in the number of columns and servers may be achievable for systems containing thousands of nodes;
- MapReduce can benefit from columnar storage just like a DBMS;
- Record assembly and parsing are expensive. Software layers (beyond the query processing layer) may be optimized to directly consume column-oriented data;
- MapReduce and query processing can be used in a complementary fashion, one layer's output can feed another's input;

- In a multi-user environment, a larger system can benefit from economies of scale while offering a qualitatively better user experience;
- If trading speed against accuracy is acceptable, a query can be terminated much earlier and yet see most of the data; and
- The bulk of a web-scale dataset can be scanned fast, although getting to the last few percent may increase the amount of processing time.

[00153] Figure 19 is a block diagram of a system for generating and processing columnar storage representations of nested records. The record generator 1904 generates records of nested data from data sources 1920 and a schema 1902. The column generator 1908 receives as input the records 1906 and the schema 1902 and outputs column stripes that represent the data in the records 1906, but in a columnar format. The columnar data 1910 may be queried *in situ* by the querying system 1912 in order to produce different sets of output columns 1914. The columnar data 1910 may also be assembled back into record form by the record assembler 1916. The records 1918 that are output by the record assembler may each include a sub-set of fields from the original records in the collection 1906. The output records 1918 may be operated on by a record-based data analysis program (e.g., MapReduce).

[00154] More specifically, the data sources 1920 may include substantially unstructured data. Substantially unstructured indicates that the data may include elements that denote structure, but the entire spectrum of information may not be similarly structured. As an illustration, the data sources 1920 may include the source code for each of millions of websites. Although each website includes some degree of structure, the content of each website is not generated based on

a common schema. Standards may generally govern a format of the site, but content and placement of fields is not specified among each and every website by a single schema. In some examples, the information in data sources 1920 is not stored in the common storage layer 1922, but is pulled directly from external sources on the internet.

[00155] The schema 1902 defines a common structuring for information that may be contained in the data sources. As described earlier in this document, the schema 1902 can require certain fields of information and may permit other fields of information to be stored as optional.

[00156] The record generator 1904 receives as input the schema 1902 and information from the data sources 1920. The record generator 1904 takes the information from the data sources 1920 and structures all or portions of the information into individual instances of records that comply with the schema 1902. For example, while the data sources 1920 may include substantially unstructured data from web pages, the record generator 1904 may select pieces of information from each web page to include for particular records 1906.

[00157] Thus, each of the records 1906 may include data that is structured according to the schema 1902. The structured data may include fields, which may denote a semantics of data values and a structural relationship of the data values. Accordingly, the schema may be referenced to obtain additional definition information for the data value (e.g., what the digitally stored data value represents in the real world or on a web page and relationships to other values).

[00158] Each record 1906 may include nested fields and data values. A nested record may include more than one field of the same name or path. The fields with the same name or path, however, can be structurally located in

different locations in a particular record. For example, a single field that is defined by the schema may be able to repeat multiple times. Further, fields may have children fields (i.e., nested fields). Thus, at a top level of a record a particular field may repeat, and each repetition of the field may or may not include a particular child field. In other words, the record may include instances of the child field in some portions of the record, but not in other portions of the records.

[00159] The collection of records 1906 may be translated into columnar data 1910 to speed up processing of information in the records. For example, if the amount of records in the collection 1906 numbers in the billions, and each record could include hundreds of different fields, an analysis of the records may be time-intensive where information on a small number of fields is desired. This is because each record in the collection 1906 is stored with other information from the record. That is, each record is grouped together in a consecutive portion of memory (e.g., as illustrated in the ‘record-oriented’ depiction of nested data in Figure 1).

[00160] In contrast, columnar data 1910 includes columns that each store information for a single field in the schema 1902 (e.g., as illustrated in the ‘column-oriented’ depiction of nested data in Figure 1). Thus, if the field is a byte long, the column for the field may be on the order of billions of bytes (e.g., one byte for each record) as opposed to billions of records (e.g., where each record may be a megabyte in size). The operations of the column generator 1908 are described in more detail in Section 4.2 “Splitting Records into Columns.” The storage format for the columnar data 1910 is described in more detail in Section 4.1 “Repetition and Definition Levels.”

[00161] The columnar data 1910 may be queried directly using the querying system 1912. In other words, the columnar data 1910 may be queried without loading the data into a database. The querying system, when executing a query, may receive as an input a table of columnar data. In some examples, the querying system also receives as input the schema 1902. The columnar stripes may be stored together with the schema to make the data self-describing. The querying system allows operations to be performed on the columnar data in order to generate columns of output information 1914. The output columns 1914 may include a subset of the values represented in the columnar data 1910, as determined by a particular query. In some examples, the querying system outputs records 1918 instead of, or in addition to, the columns 1914.

[00162] For example, the querying system 1912 may receive a first query and, in response, may parse through select columns of data and generate a set of output columns that provides a title of all web pages that have one or more videos and a number of the videos for each web page. The querying system may receive a second query and in response output a second set of output columns that provides a URL of every web page that was generated within the last fifteen minutes. Other information from the columns 1910 may not be included in a set of output columns that corresponds to a particular query 1914.

[00163] Data that is stored as columnar data 1910 may need to be accessed by an analytical service that does not operate on columnar data but operates on records. Thus, the record assembler 1916 may receive as input the columnar data and assemble records from the columnar data. The process of assembling records is described in more detail in Section 4.3 “Record Assembly.”

[00164] Although the records may already be available in the collection 1906, the record assembler 1916 enables generating a set of records that includes a subset of the fields of the records in the collection 1906. For example, the records in the collection may include thousands of different fields. A user may want to run a record-oriented analysis program that only requires knowledge from two of the fields, but for all of the records. Thus, the record assembler 1916 may generate a set of records that only includes information on the requested fields. This way, multiple sets of output records 1918 can be developed for different analysis or for different analysis programs. An analysis on smaller records may be faster than an analysis that must traverse the larger records that may be found in collection 1906.

[00165] The above description of the operation of the system 1900 illustrates an example where the collection of records 1906 includes records that are formatted in accordance with the schema 1902, and where the columnar data 1910 is generated from this single set of similarly-structured data. In various examples, multiple schemas 1902 may be used to generate a collection of records that includes many sets of differently structured records 1906. Each record, however, may identify in a header the type of schema that was used in the record's generation. Similarly, a column stripe may be generated for each field in each of many sets of similarly-structured records. Each column stripe may indicate not only the name of the field, but also the schema from which the columnar data is associated (i.e., the schema used to format the records from which the columnar data was generated).

[00166] Figure 20 is a flow chart of an example process for generating columnar data. The process may be performed by components of the system 1900.

[00167] In box 2002, a set of records is generated. The generation of the records may be performed by the record generator 1904. Unstructured data (e.g., from data sources 1920) may be compiled into a standardized record format that is defined by schema 1902. The records may be stored in the collection 1906.

[00168] In box 2004, the records in the collection 1906 are accessed. For example, the column generator 1908 receives as input the data from the collection of records 1906.

[00169] In box 2006, a determination is made whether a column stripe is to be generated for an additional field. For example, a stripe is to be generated for each field in the set of records that are stored in the collection 1906 (and thus each record in the schema 1902 or a subset thereof). In this illustration, no stripes have been made so far, and thus there are fields for which a stripe is to be generated. Accordingly, the process proceeds to box 2008 in order to perform operations for a particular field. If all stripes had been generated (e.g., a stripe had been generated for every field in the collection of records 1906), the process may end.

[00170] In box 2008, a list of values for the particular is generated. For example, each of the records may be traversed and a list of values for the particular field is generated.

[00171] In box 2010, repetition levels for the particular field are generated. For example, the column generator 1908 may determine a repetition level for

each of the values in the list by determining a most recently repeated field in the path for the field.

[00172] In box 2012, definition levels for the particular field are generated. For example, the column generator 1908 may determine a definition level for each value (including values that are 'missing,' as described in more detail above).

[00173] In box 2014, a columnar stripe is assembled for the particular field. In various examples, the repetition and definition levels are placed in paired groupings in the header of the stripe. The list of values may be placed in the body of the stripe.

[00174] In box 2016, the columnar stripe is broken into blocks that may be compressed. Each block may include a set of values and their corresponding repetition and definition levels. Subsequently, a determination in box 2006 of whether columnar stripes are to be generated for additional fields is performed. If no additional columnar stripes are to be generated, the process ends.

[00175] The process depicted in Figure 20 is an example process for generating columnar stripes. Variations on the process are contemplated. For example, the operations of the boxes may not be performed sequentially as depicted in the flowchart. Stripes for multiple fields may be generated at a single time. The repetition level and definition level may be generated as each value is obtained from a record. The columnar stripe may not be generated as a whole. Instead, each block may be generated from the stripe and independently compressed. Thus, the flowchart may represent a conceptual mechanism for understanding the generation of stripes, but is not intended to be limiting. A

process for generating columnar data is depicted in the algorithm of Figure 4, which may not correspond to the operations described in relation to Figure 20.

[00176] Figure 21 is a block diagram of computing devices 2100, 2150 that may be used to implement the systems and methods described in this document, as either a client or as a server or plurality of servers. Computing device 2100 is intended to represent various forms of digital computers, such as laptops, desktops, workstations, personal digital assistants, servers, blade servers, mainframes, and other appropriate computers. Computing device 2150 is intended to represent various forms of mobile devices, such as personal digital assistants, cellular telephones, smartphones, and other similar computing devices. Additionally computing device 2100 or 2150 can include Universal Serial Bus (USB) flash drives. The USB flash drives may store operating systems and other applications. The USB flash drives can include input/output components, such as a wireless transmitter or USB connector that may be inserted into a USB port of another computing device. The components shown here, their connections and relationships, and their functions, are meant to be exemplary only, and are not meant to limit implementations described and/or claimed in this document.

[00177] Computing device 2100 includes a processor 2102, memory 2104, a storage device 2106, a high-speed interface 2108 connecting to memory 2104 and high-speed expansion ports 2110, and a low speed interface 2112 connecting to low speed bus 2114 and storage device 2106. Each of the components 2102, 2104, 2106, 2108, 2110, and 2112, are interconnected using various busses, and may be mounted on a common motherboard or in other manners as appropriate. The processor 2102 can process instructions for

execution within the computing device 2100, including instructions stored in the memory 2104 or on the storage device 2106 to display graphical information for a GUI on an external input/output device, such as display 2116 coupled to high speed interface 2108. In other implementations, multiple processors and/or multiple buses may be used, as appropriate, along with multiple memories and types of memory. Also, multiple computing devices 2100 may be connected, with each device providing portions of the necessary operations (e.g., as a server bank, a group of blade servers, or a multi-processor system).

[00178] The memory 2104 stores information within the computing device 2100. In one implementation, the memory 2104 is a volatile memory unit or units. In another implementation, the memory 2104 is a non-volatile memory unit or units. The memory 2104 may also be another form of computer-readable medium, such as a magnetic or optical disk.

[00179] The storage device 2106 is capable of providing mass storage for the computing device 2100. In one implementation, the storage device 2106 may be or contain a computer-readable medium, such as a floppy disk device, a hard disk device, an optical disk device, or a tape device, a flash memory or other similar solid state memory device, or an array of devices, including devices in a storage area network or other configurations. A computer program product can be tangibly embodied in an information carrier. The computer program product may also contain instructions that, when executed, perform one or more methods, such as those described above. The information carrier is a computer- or machine-readable medium, such as the memory 2104, the storage device 2106, or memory on processor 2102.

[00180] The high speed controller 2108 manages bandwidth-intensive operations for the computing device 2100, while the low speed controller 2112 manages lower bandwidth-intensive operations. Such allocation of functions is exemplary only. In one implementation, the high-speed controller 2108 is coupled to memory 2104, display 2116 (e.g., through a graphics processor or accelerator), and to high-speed expansion ports 2110, which may accept various expansion cards (not shown). In the implementation, low-speed controller 2112 is coupled to storage device 2106 and low-speed expansion port 2114. The low-speed expansion port, which may include various communication ports (e.g., USB, Bluetooth, Ethernet, wireless Ethernet) may be coupled to one or more input/output devices, such as a keyboard, a pointing device, a scanner, or a networking device such as a switch or router, e.g., through a network adapter.

[00181] The computing device 2100 may be implemented in a number of different forms, as shown in the figure. For example, it may be implemented as a standard server 2120, or multiple times in a group of such servers. It may also be implemented as part of a rack server system 2124. In addition, it may be implemented in a personal computer such as a laptop computer 2122.

Alternatively, components from computing device 2100 may be combined with other components in a mobile device (not shown), such as device 2150. Each of such devices may contain one or more of computing device 2100, 2150, and an entire system may be made up of multiple computing devices 2100, 2150 communicating with each other.

[00182] Computing device 2150 includes a processor 2152, memory 2164, an input/output device such as a display 2154, a communication interface 2166, and a transceiver 2168, among other components. The device 2150 may also be

provided with a storage device, such as a microdrive or other device, to provide additional storage. Each of the components 2150, 2152, 2164, 2154, 2166, and 2168, are interconnected using various buses, and several of the components may be mounted on a common motherboard or in other manners as appropriate.

[00183] The processor 2152 can execute instructions within the computing device 2150, including instructions stored in the memory 2164. The processor may be implemented as a chipset of chips that include separate and multiple analog and digital processors. Additionally, the processor may be implemented using any of a number of architectures. For example, the processor 410 may be a CISC (Complex Instruction Set Computers) processor, a RISC (Reduced Instruction Set Computer) processor, or a MISC (Minimal Instruction Set Computer) processor. The processor may provide, for example, for coordination of the other components of the device 2150, such as control of user interfaces, applications run by device 2150, and wireless communication by device 2150.

[00184] Processor 2152 may communicate with a user through control interface 2158 and display interface 2156 coupled to a display 2154. The display 2154 may be, for example, a TFT (Thin-Film-Transistor Liquid Crystal Display) display or an OLED (Organic Light Emitting Diode) display, or other appropriate display technology. The display interface 2156 may comprise appropriate circuitry for driving the display 2154 to present graphical and other information to a user. The control interface 2158 may receive commands from a user and convert them for submission to the processor 2152. In addition, an external interface 2162 may be provide in communication with processor 2152, so as to enable near area communication of device 2150 with other devices. External interface 2162 may provide, for example, for wired communication in some implementations, or for

wireless communication in other implementations, and multiple interfaces may also be used.

[00185] The memory 2164 stores information within the computing device 2150. The memory 2164 can be implemented as one or more of a computer-readable medium or media, a volatile memory unit or units, or a non-volatile memory unit or units. Expansion memory 2174 may also be provided and connected to device 2150 through expansion interface 2172, which may include, for example, a SIMM (Single In Line Memory Module) card interface. Such expansion memory 2174 may provide extra storage space for device 2150, or may also store applications or other information for device 2150. Specifically, expansion memory 2174 may include instructions to carry out or supplement the processes described above, and may include secure information also. Thus, for example, expansion memory 2174 may be provide as a security module for device 2150, and may be programmed with instructions that permit secure use of device 2150. In addition, secure applications may be provided via the SIMM cards, along with additional information, such as placing identifying information on the SIMM card in a non-hackable manner.

[00186] The memory may include, for example, flash memory and/or NVRAM memory, as discussed below. In one implementation, a computer program product is tangibly embodied in an information carrier. The computer program product contains instructions that, when executed, perform one or more methods, such as those described above. The information carrier is a computer- or machine-readable medium, such as the memory 2164, expansion memory 2174, or memory on processor 2152 that may be received, for example, over transceiver 2168 or external interface 2162..

[00187] Device 2150 may communicate wirelessly through communication interface 2166, which may include digital signal processing circuitry where necessary. Communication interface 2166 may provide for communications under various modes or protocols, such as GSM voice calls, SMS, EMS, or MMS messaging, CDMA, TDMA, PDC, WCDMA, CDMA2000, or GPRS, among others. Such communication may occur, for example, through radio-frequency transceiver 2168. In addition, short-range communication may occur, such as using a Bluetooth, WiFi, or other such transceiver (not shown). In addition, GPS (Global Positioning System) receiver module 2170 may provide additional navigation- and location-related wireless data to device 2150, which may be used as appropriate by applications running on device 2150.

[00188] Device 2150 may also communicate audibly using audio codec 2160, which may receive spoken information from a user and convert it to usable digital information. Audio codec 2160 may likewise generate audible sound for a user, such as through a speaker, e.g., in a handset of device 2150. Such sound may include sound from voice telephone calls, may include recorded sound (e.g., voice messages, music files, etc.) and may also include sound generated by applications operating on device 2150.

[00189] The computing device 2150 may be implemented in a number of different forms, as shown in the figure. For example, it may be implemented as a cellular telephone 2180. It may also be implemented as part of a smartphone 2182, personal digital assistant, or other similar mobile device.

[00190] Various implementations of the systems and techniques described here can be realized in digital electronic circuitry, integrated circuitry, specially designed ASICs (application specific integrated circuits), computer hardware,

firmware, software, and/or combinations thereof. These various implementations can include implementation in one or more computer programs that are executable and/or interpretable on a programmable system including at least one programmable processor, which may be special or general purpose, coupled to receive data and instructions from, and to transmit data and instructions to, a storage system, at least one input device, and at least one output device.

[00191] These computer programs (also known as programs, software, software applications or code) include machine instructions for a programmable processor, and can be implemented in a high-level procedural and/or object-oriented programming language, and/or in assembly/machine language. As used herein, the terms “machine-readable medium” “computer-readable medium” refers to any computer program product, apparatus and/or device (e.g., magnetic discs, optical disks, memory, Programmable Logic Devices (PLDs)) used to provide machine instructions and/or data to a programmable processor, including a machine-readable medium that receives machine instructions as a machine-readable signal. The term “machine-readable signal” refers to any signal used to provide machine instructions and/or data to a programmable processor.

[00192] To provide for interaction with a user, the systems and techniques described here can be implemented on a computer having a display device (e.g., a CRT (cathode ray tube) or LCD (liquid crystal display) monitor) for displaying information to the user and a keyboard and a pointing device (e.g., a mouse or a trackball) by which the user can provide input to the computer. Other kinds of devices can be used to provide for interaction with a user as well; for example, feedback provided to the user can be any form of sensory feedback (e.g., visual

feedback, auditory feedback, or tactile feedback); and input from the user can be received in any form, including acoustic, speech, or tactile input.

[00193] The systems and techniques described here can be implemented in a computing system that includes a back end component (e.g., as a data server), or that includes a middleware component (e.g., an application server), or that includes a front end component (e.g., a client computer having a graphical user interface or a Web browser through which a user can interact with an implementation of the systems and techniques described here), or any combination of such back end, middleware, or front end components. The components of the system can be interconnected by any form or medium of digital data communication (e.g., a communication network). Examples of communication networks include a local area network ("LAN"), a wide area network ("WAN"), peer-to-peer networks (having ad-hoc or static members), grid computing infrastructures, and the Internet.

[00194] The computing system can include clients and servers. A client and server are generally remote from each other and typically interact through a communication network. The relationship of client and server arises by virtue of computer programs running on the respective computers and having a client-server relationship to each other.

[00195] Although a few implementations have been described in detail above, other modifications are possible. Moreover, other mechanisms for generating and processing columnar storage representations of nested records may be used. In addition, the logic flows depicted in the figures do not require the particular order shown, or sequential order, to achieve desirable results. Other steps may be provided, or steps may be eliminated, from the described flows,

and other components may be added to, or removed from, the described systems. Accordingly, other implementations are within the scope of the following claims.

WHAT IS CLAIMED IS:

1. A computer-implemented method comprising:
 - accessing, by a computing system, a collection of data records that are stored in computer memory, each record in the collection of data records including a plurality of data values and a plurality of data elements that identify semantics of corresponding data values from the plurality of data values, one or more of the data records in the collection of data records each including multiple instances of a same data element and including data values that correspond to the multiple instances of the same data element, wherein at least one data record in the collection of data records includes data elements and corresponding data values that are stored according to a nested data model;
 - generating, by the computing system, a collection of columnar stripes, the collection of columnar stripes including the data values from each data record in the collection of data records, each columnar stripe in the collection of columnar stripes including all data values that correspond to a specific data element from each of the records in the collection of records, and
 - generating, by the computer system, for each data value in each columnar stripe in the collection of columnar stripes and for storage with the data values in the collection of columnar strips, data that identifies a location of the respective data value in a respective data record from the collection of data records.
2. The method of claim 1, wherein the data consists of a repetition value and a definition value.
3. The method of any one of claims 1 through 2, further comprising:
 - upon storing the repetition value and the definition value for each data value in the collection of columnar stripes, reconstructing, from (i) the columnar stripes in the collection of columnar stripes, and (ii) the data, a set of records that contains just a subset of data elements from the records in the collection of data records.
4. The method of any one of claims 1 through 3, further comprising generating, for storage with the data values in the collection of columnar stripes, a repetition value for each particular data value in the collection of columnar stripes,

wherein a path for each particular data element includes any one or more parent data elements to the particular data element;

wherein the repetition value for each particular data value identifies a most-recently repeated data element in the path of the particular data element that corresponds to the particular data value;

wherein the most recently-repeated data element in the path of the particular data element is the data element in the path of the particular data element that is encountered for a second time during an analysis, of a particular data record that includes the particular data value, that works upwards towards a beginning of the particular data record from a position of the particular data value in the particular data record.

5. The method of any one of claims 1 through 4, wherein each particular data element, of the data elements that are included in the collection of data elements, are associated with a respective path that includes any one or more of the parent data elements to the particular data element;

further comprising generating, for storage with the data values in the columnar stripes, a definition value for each particular path or portion of the particular path in the collection of data records;

wherein the definition level for the particular path or portion of the particular path identifies a quantity of data elements included in the particular path or portion of path.

6. The method of any one of claims 1 through 5, further comprising:

receiving, by the computing system, information from a collection of data sources, each data source including information that is unstructured according to a schema; and

generating, by the computing system, each data record in the collection of data records by structuring the information in each data source according to the schema.

7. The method of any one of claims 1 through 6, further comprising:

performing, by the computing system, a query within the collection of columnar stripes; and

outputting, by the computing system and in response to performance of the query, a new columnar stripe that includes a subset of the values from a columnar stripe from the collection of columnar stripes that are identified by the query.

8. The method of claim 7, wherein the query of the collection of columnar stripes is performed without loading the data values included in the collection of columnar stripes into a database.
9. The method of any one of claims 1 through 8, wherein at least a first columnar stripe of the collection of columnar stripes includes multiple blocks of data, at least some of the multiple blocks of data each including an assertion value that defines a kind of values that are found in each block's values, so that the computing system, when performing a query of the first columnar stripe, avoids querying one or more blocks of data that do not include a data value that is specified by the query.
10. The method of any one of claims 1 through 9, wherein:
 - a first data record includes a first data element and a second data element, the first data element being a parent data element to the second data element, and the second data element being a child data element to the first data element;
 - a first columnar stripe includes a data value that corresponds to the first data element in the first data record and all other data values that correspond to instances of the first data element in the collection of data records; and
 - a second columnar stripe includes a data value that corresponds to the second data element in the first data record and all other data values that correspond to instances of the second data element in the collection of data records.
11. The method of any one of claims 2 through 10, wherein some data elements of the data elements stored according to the nested data model are children of other data elements of the data elements stored according to the nested data model, which are referenced as parent data elements.

12. A computer program product, tangibly stored on a computer-readable storage device, the product comprising computer program instructions operable to cause one or more processors to perform operations according to any one of claims 1 to 11.
13. A system comprising:
 - one or more programmable processors; and
 - a computer-readable storage device coupled to the processor and having instructions stored therein, which when executed by the one or more programmable processors causes the one or more programmable processors to perform operations according to any one of claims 1 to 11.

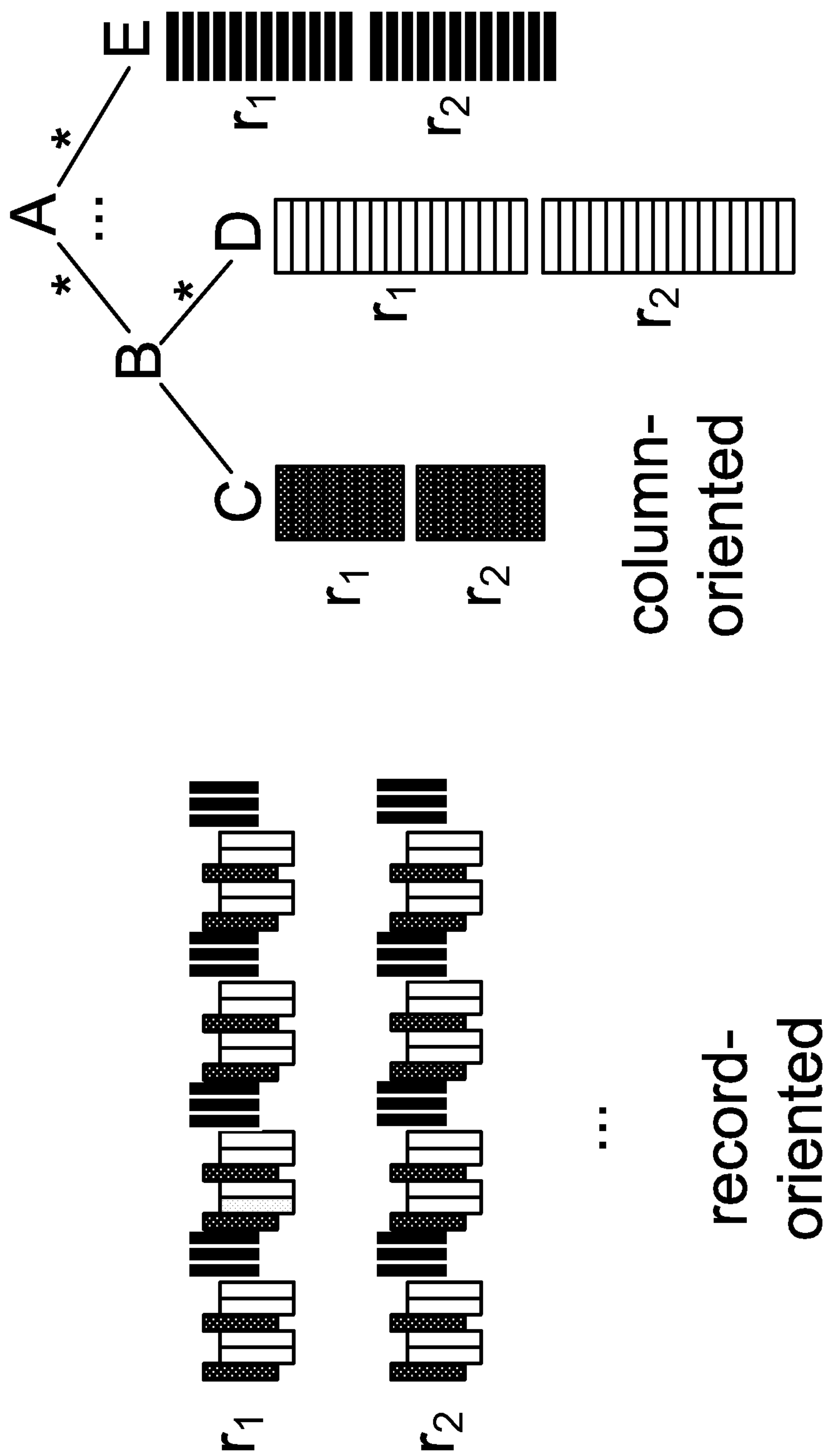
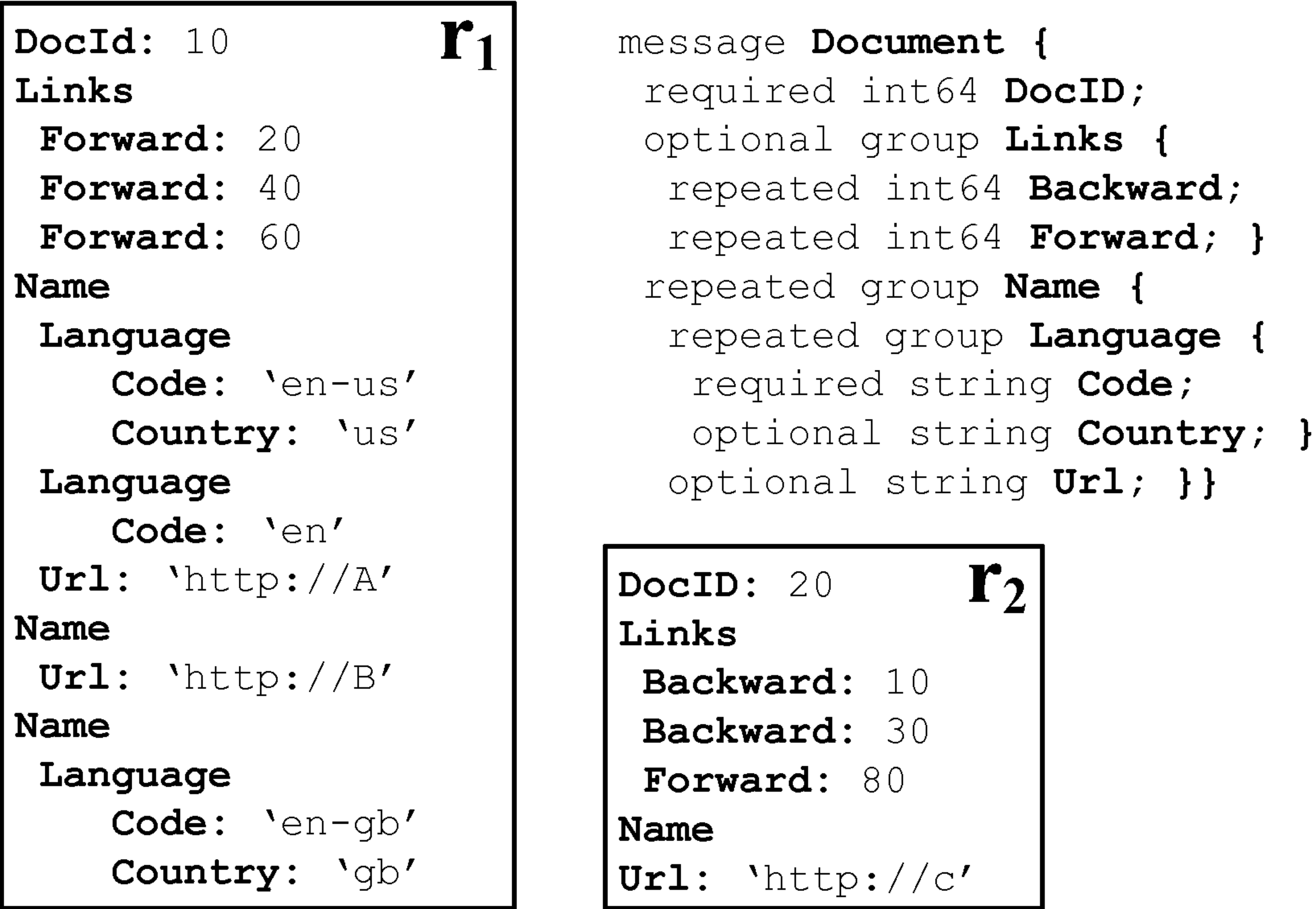


FIG. 1

2/14



3/14

```
1 procedure DissectRecord (RecordDecoder decoder,  
2     FieldWriter writer, int repetitionLevel) :  
3   Add current repetitionLevel and definition level to writer  
4   seenFields = {} // empty set of integers  
5   while decoder has more field values  
6     FieldWriter chWriter =  
7       child of writer for field read by decoder  
8     int chRepetitionLevel = repetitionLevel  
9     if set seenFields contains field ID of chWriter  
10      chRepetitionLevel = tree depth of chWriter  
11    else  
12      Add field ID of chWriter to seenFields  
13    end if  
14    if chWriter corresponds to an atomic field  
15      Write value of current field read by decoder  
16      using chWriter at chRepetitionLevel  
17    else  
18      DissectRecord (new RecordDecoder for nested record  
19        read by decoder, chWriter, chRepetitionLevel)  
20    end if  
21  end while  
22 end procedure
```

FIG. 4

4/14

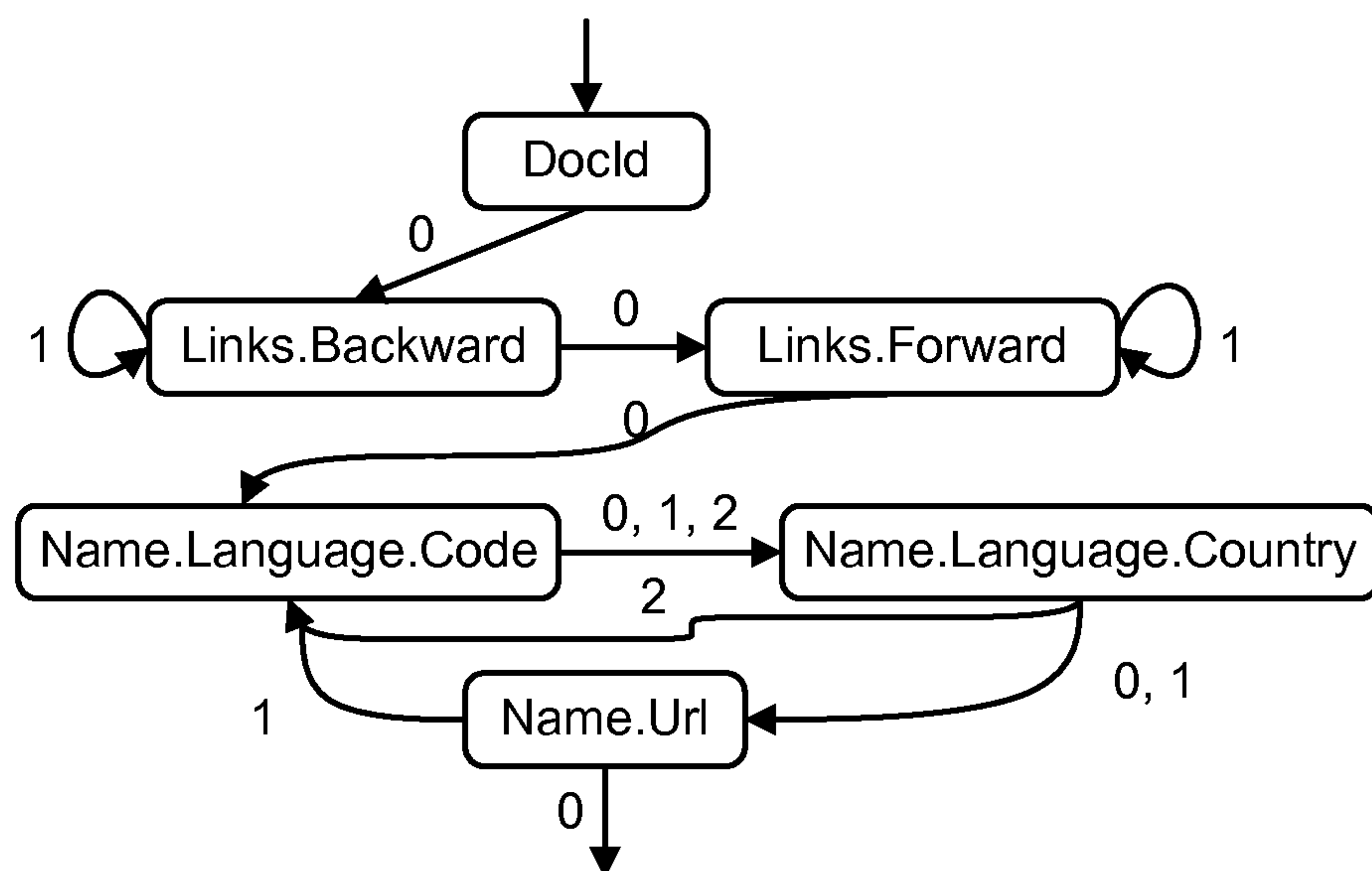


FIG. 5

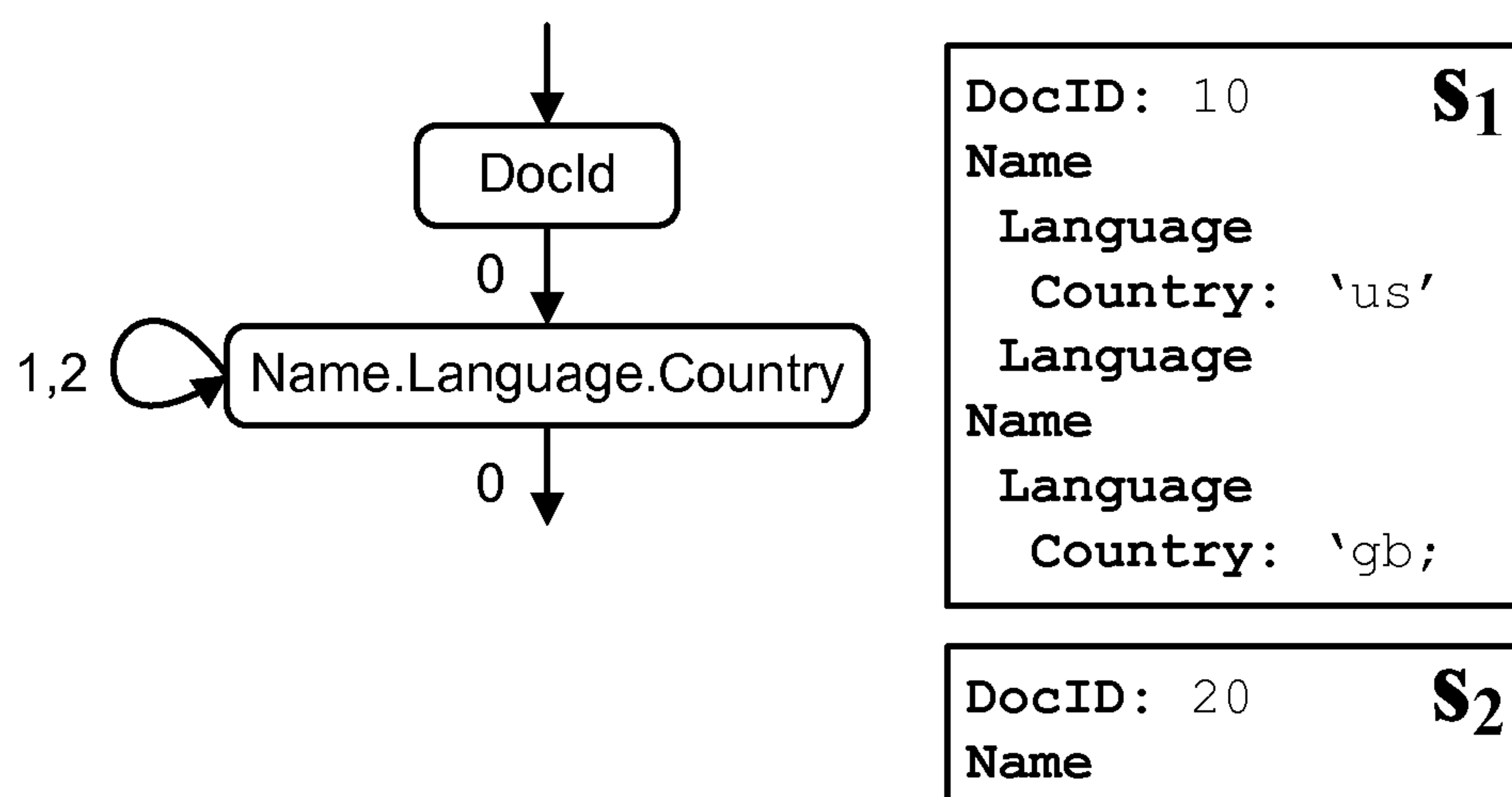


FIG. 6

5/14

```
1 procedure ConstructFSM(Field [] fields) :  
2 for each field in fields:  
3   maxLevel = maximal repetition level of field  
4   barrier = next field after field or final FSM state otherwise  
5   barrierLevel = common repletion level of field and barrier  
6   for each preField before field whose  
7     repetition level is larger than barrierLevel:  
8     backLevel = common repetition level of preField and field  
9     Set transition (field, backLevel) -> preField  
10  end for  
11  for each level in [barrierLevel+1..maxLevel]  
12    that lacks transition from field:  
13    Copy transition's destination from that of level-1  
14  end for  
15  for each level in [0..barrierLevel]:  
16    Set transition (field, level) -> barrier  
17  end for  
18 end for  
19 end procedure
```

FIG. 7

6/14

```

1 Record AssembleRecord(FieldReaders[] readers) :
2   record = create a new record
3   lastReader = select the root field reader in readers
4   reader = readers [0]
5   while reader has data
6     Fetch next value from reader
7     if current value is not NULL
8       MoveToLevel (tree level of reader, reader)
9       Append reader's value to record
10    else
11      MoveToLevel (full definition level of reader, reader)
12    end if
13    reader = reader that FSM transitions to
14      when reading next repetition level from reader
15    ReturnToLevel (tree level of reader)
16  end while
17  ReturnToLevel (0)
18  End all nested records
19  return record
20 end procedure
21
22 MoveToLevel (int newLevel, FieldReader nextReader):
23   End nested records up to the level of the lowest common ancestor
24   of lastReader and nextReader
25   Start nested records from the level of the lowest common ancestor
26   up to newLevel
27   Set lastReader to the one at newLevel
28 end procedure
29
30 ReturnToLevel (int newLevel) {
31   End nested records up to newLevel
32   Set lastReader to the one at newLevel
33 end procedure

```

FIG. 8

7/14

```

SELECT DocId AS Id,
       COUNT (Name.Language.Code) WRITTEN Name AS Cnt,
       Name.Url + ',' + Name.Language.Code AS Str
FROM t
WHERE REGEXP (Name.Url, '^http') AND DocId <20;

```

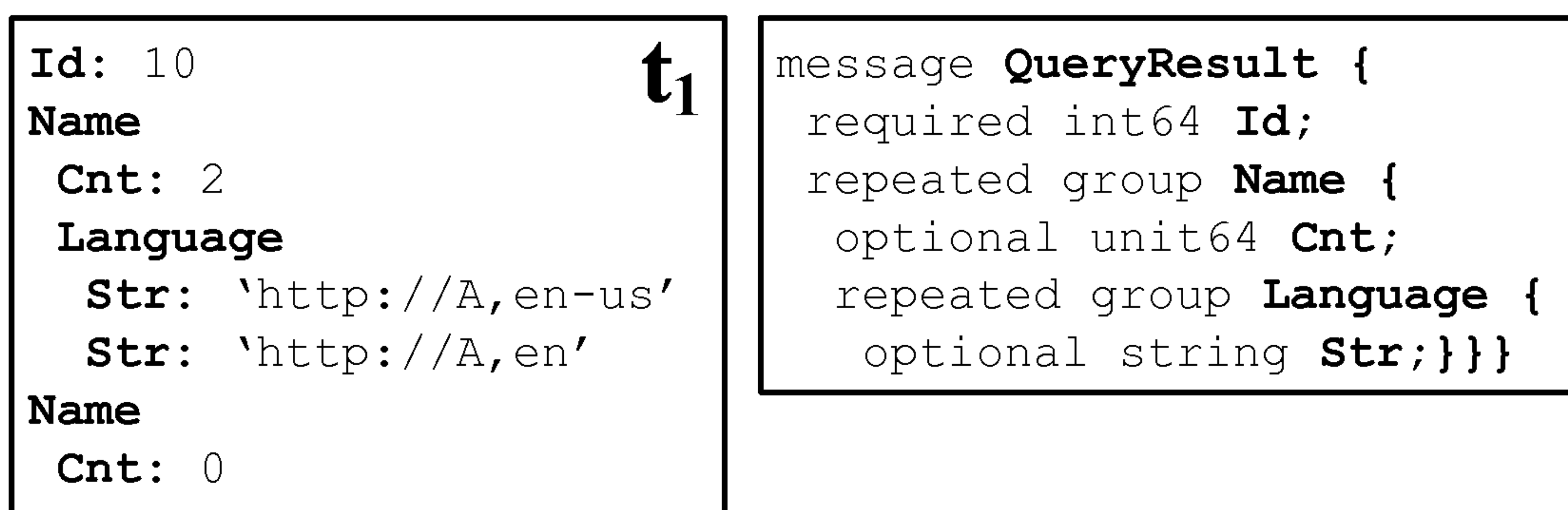


FIG. 9

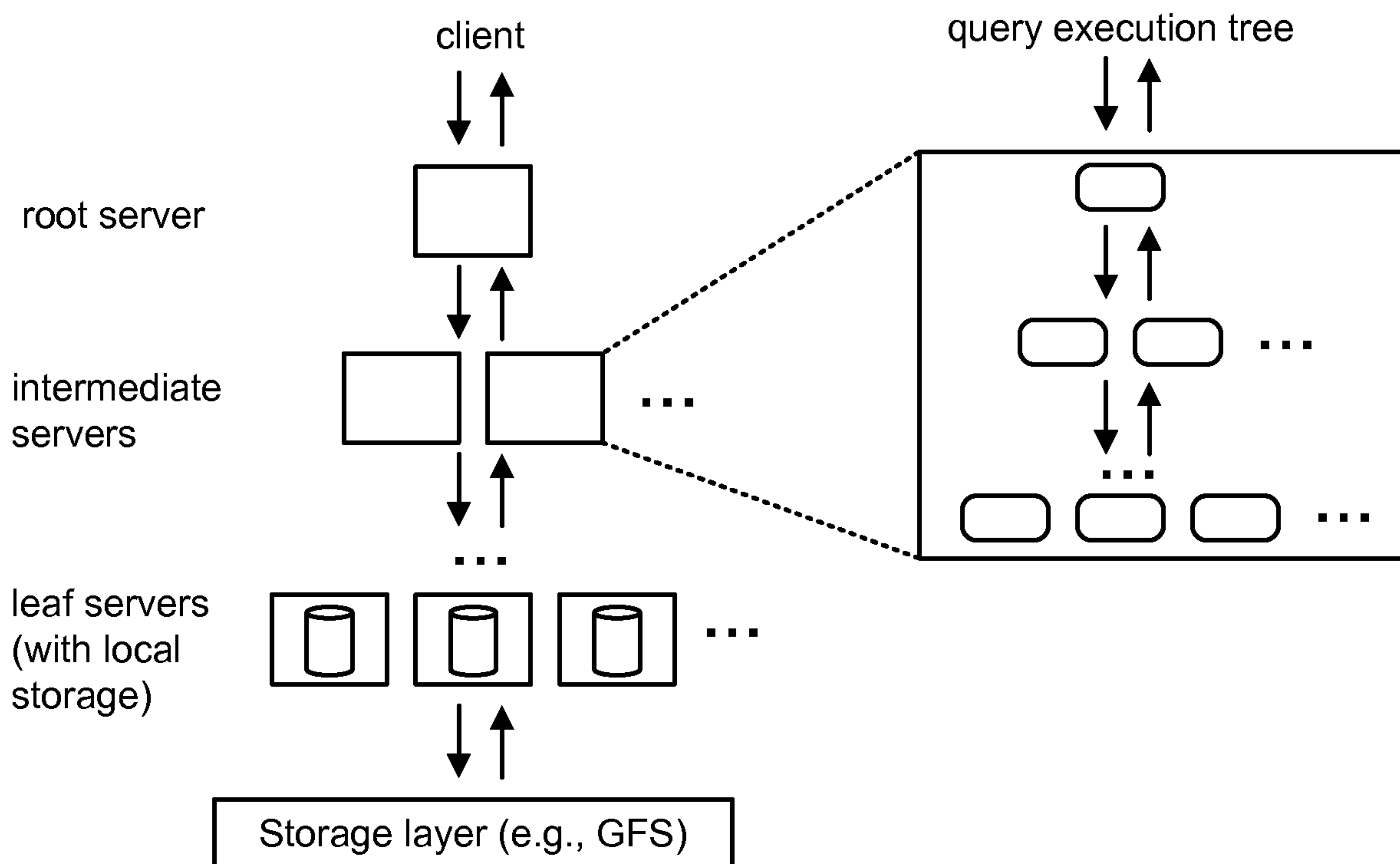


FIG. 10

8/14

Table Name	Number of records	Size (unrepl. compressed)	Number of fields	Data center	Repl. factor
T1	85 billion	87 TB	270	A	3x
T2	24 billion	13 TB	530	A	3x
T3	4 billion	70 TB	1200	A	3x
T4	1+ trillion	105 TB	50	B	3x
T5	1+ trillion	20 TB	30	B	2x

FIG. 11

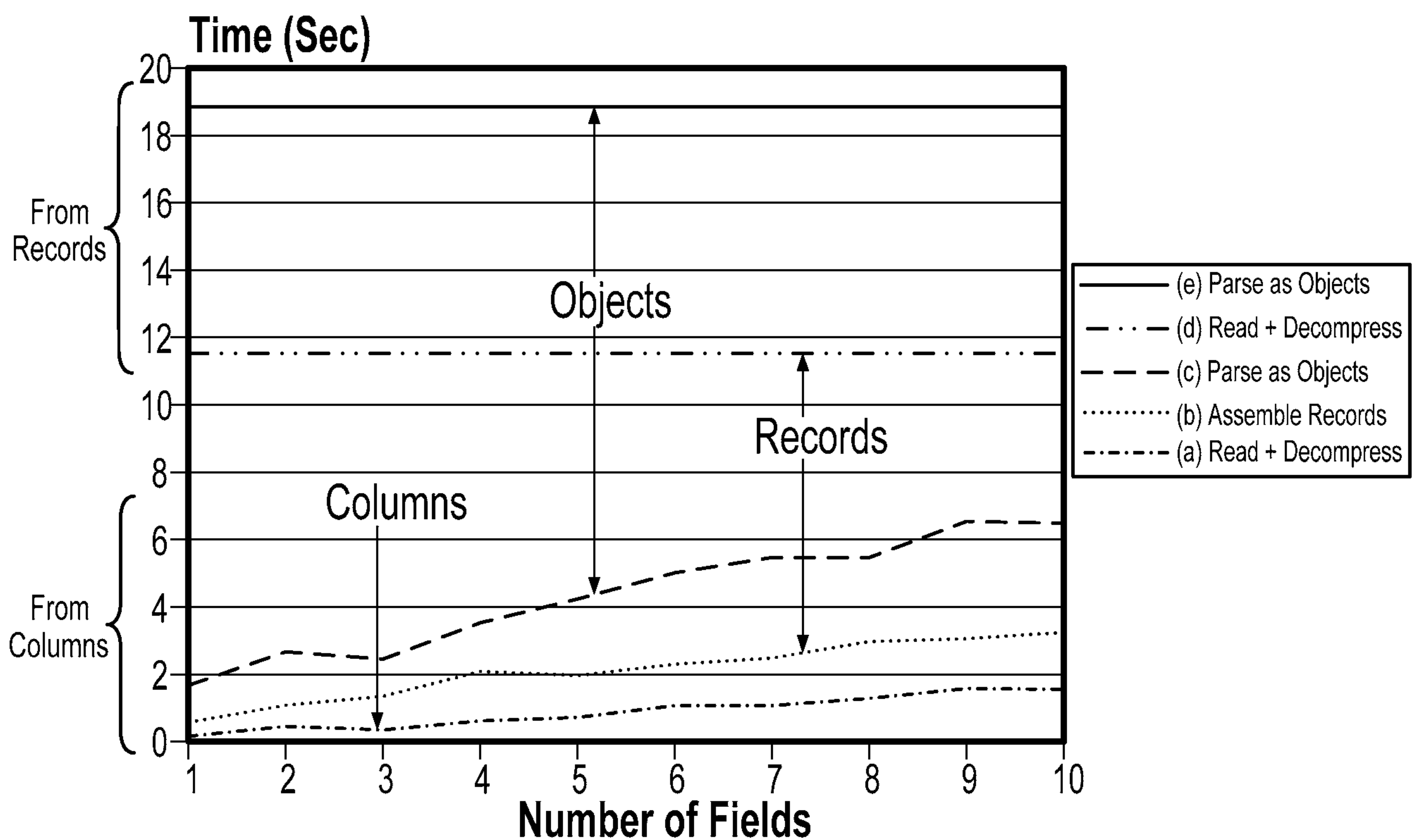


FIG. 12

9/14

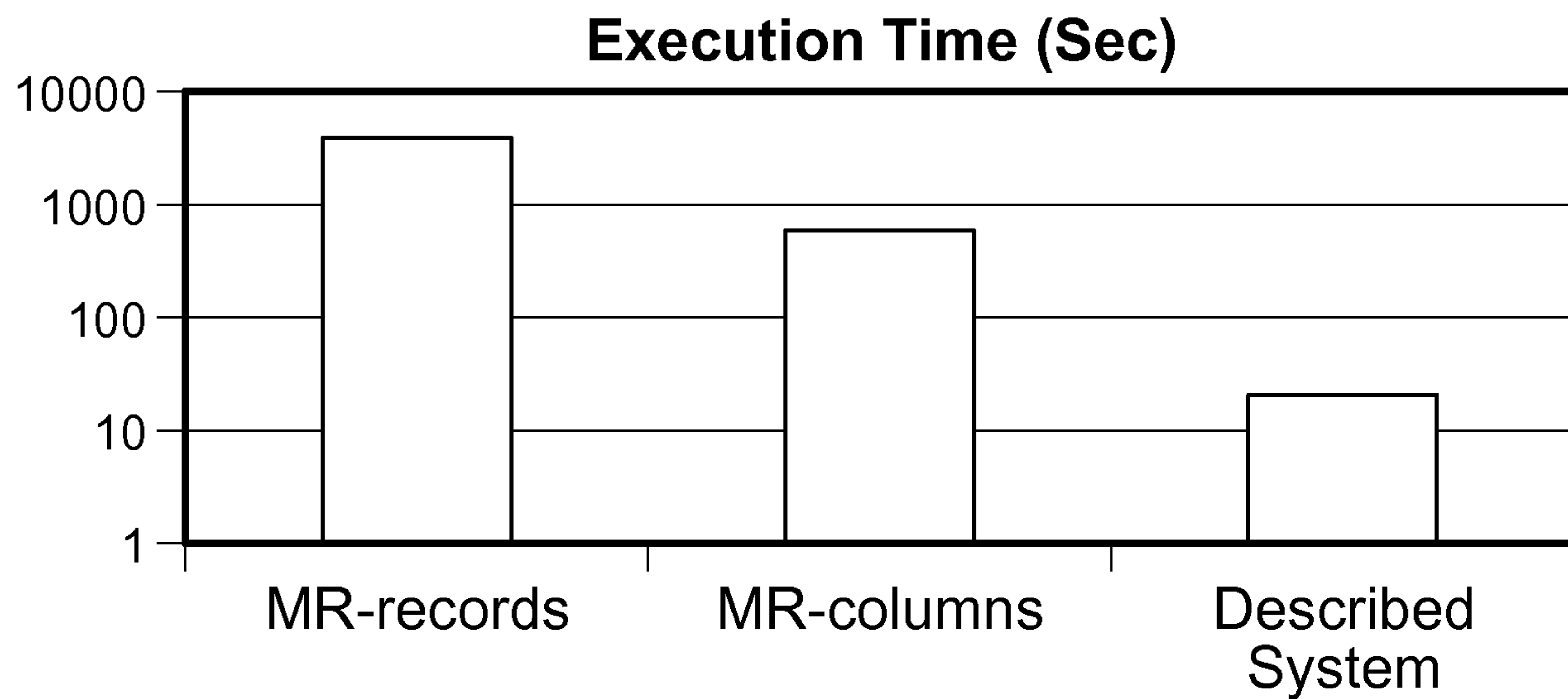


FIG. 13

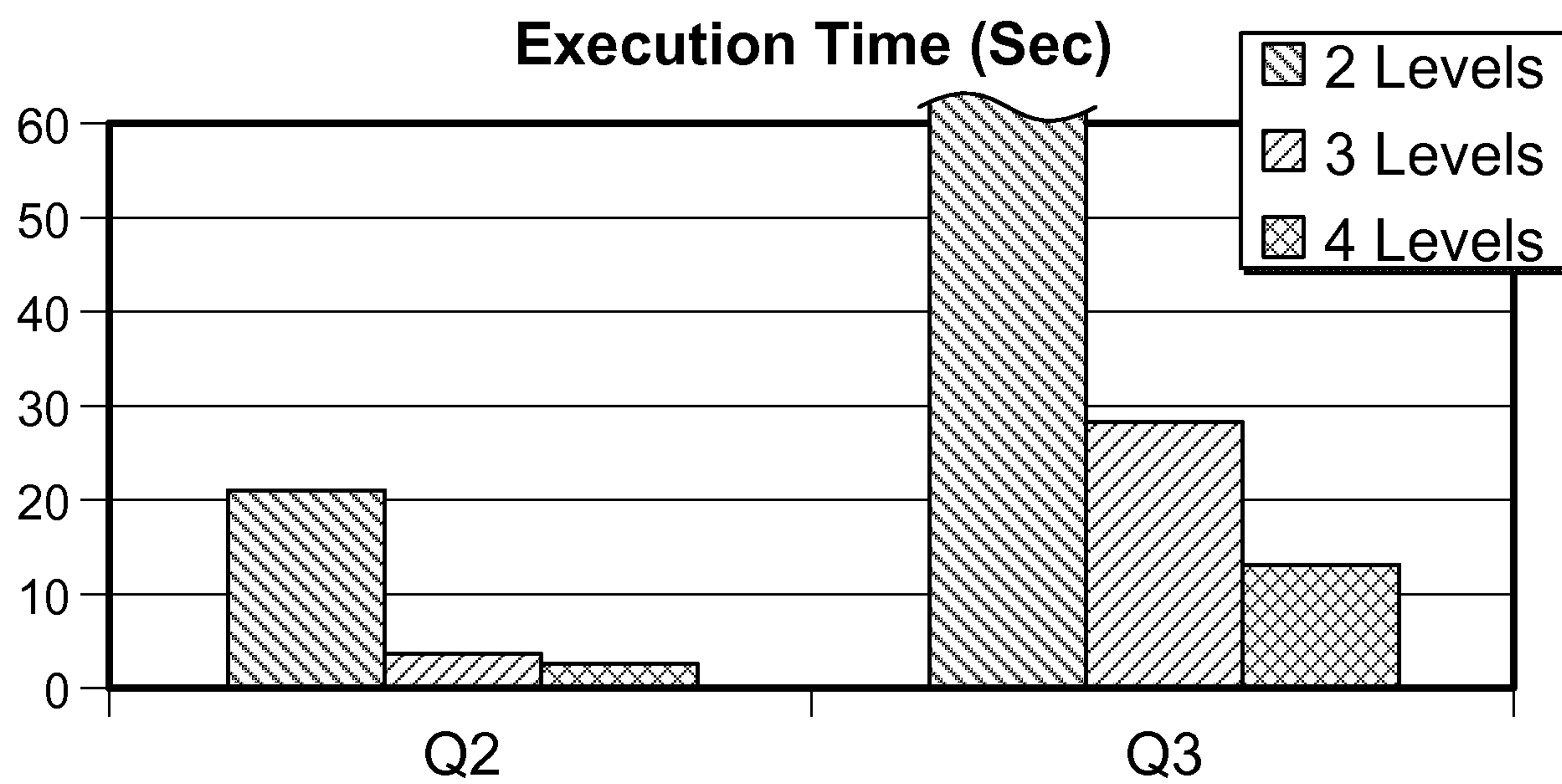


FIG. 14

10/14

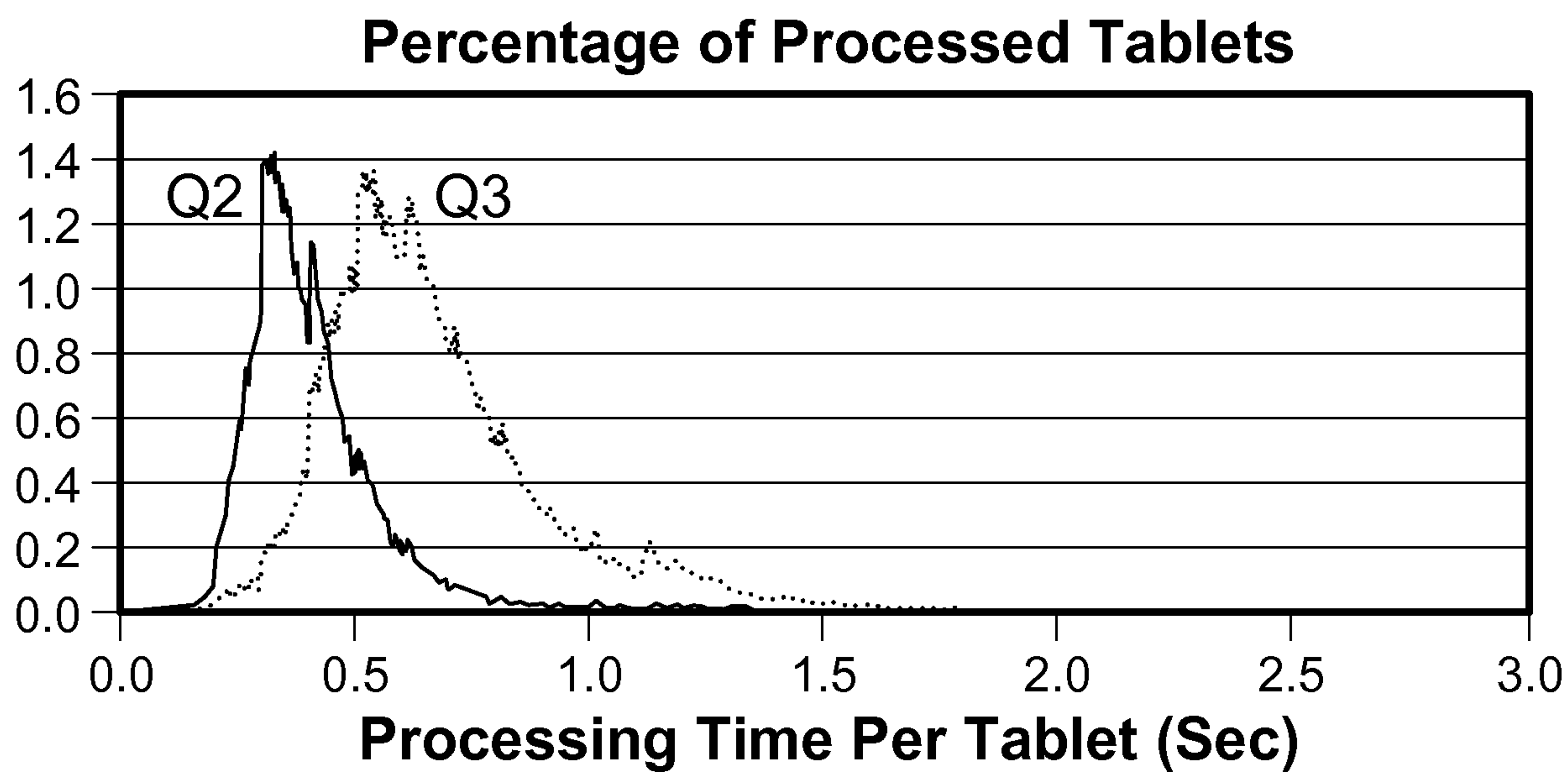


FIG. 15

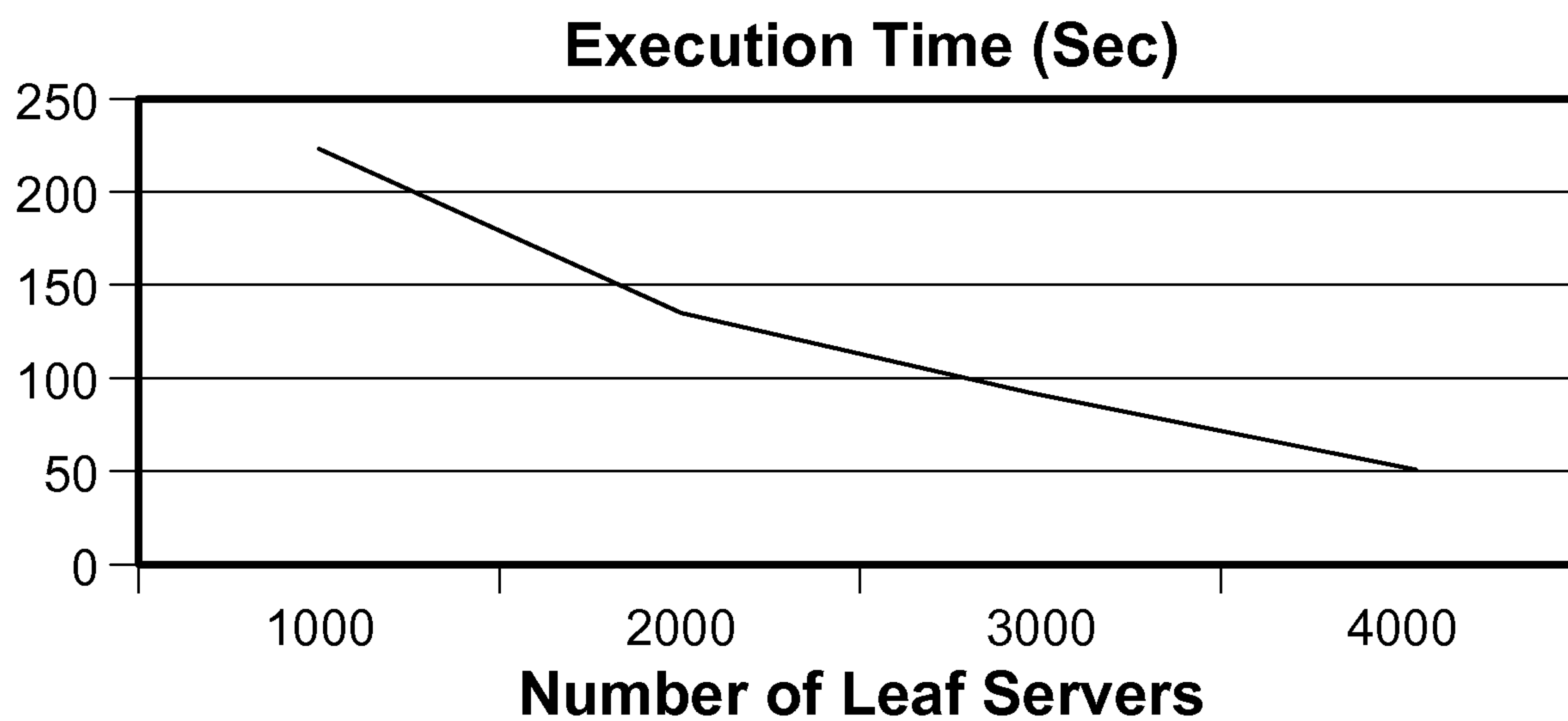


FIG. 16

11/14

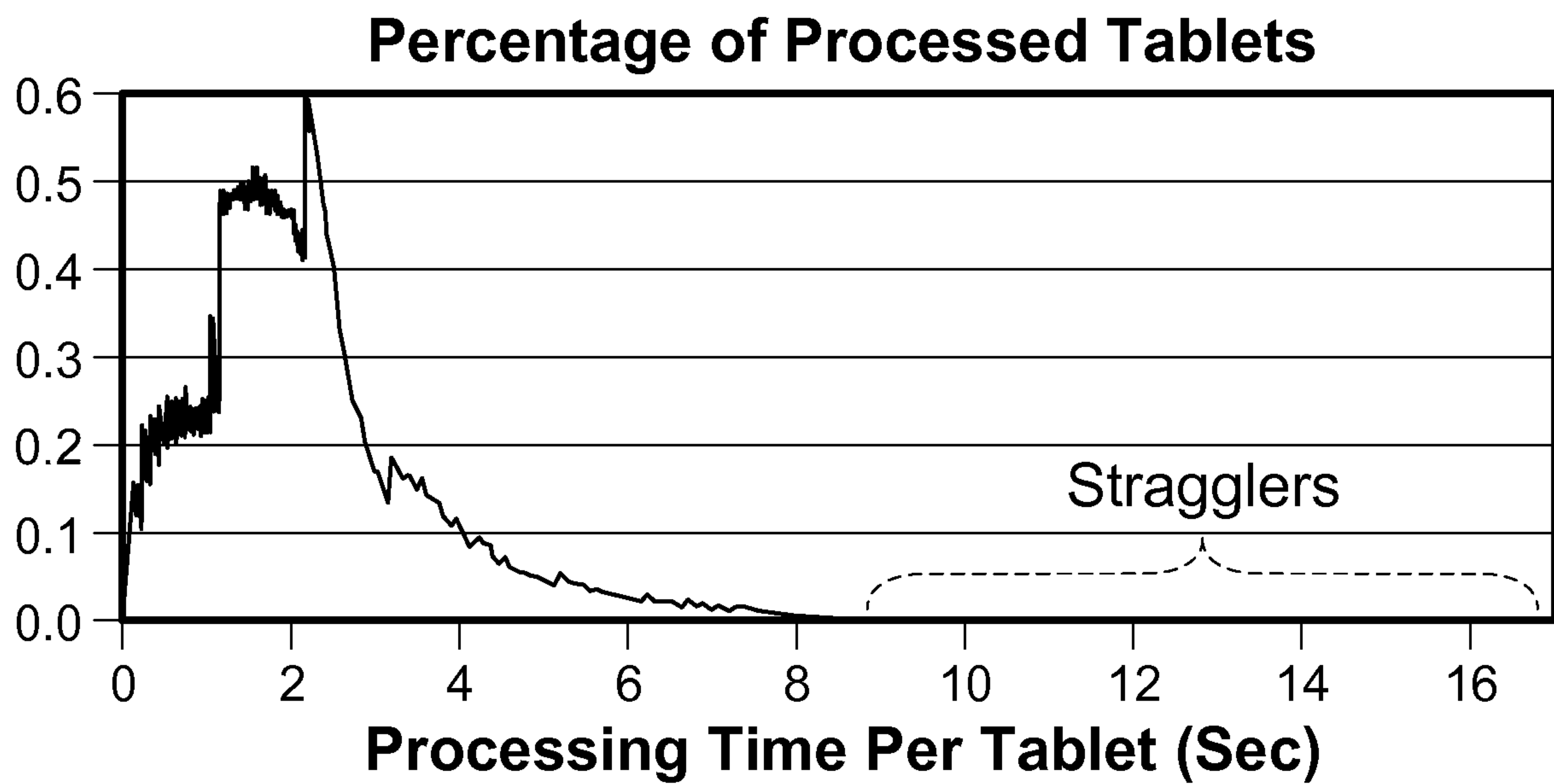


FIG. 17

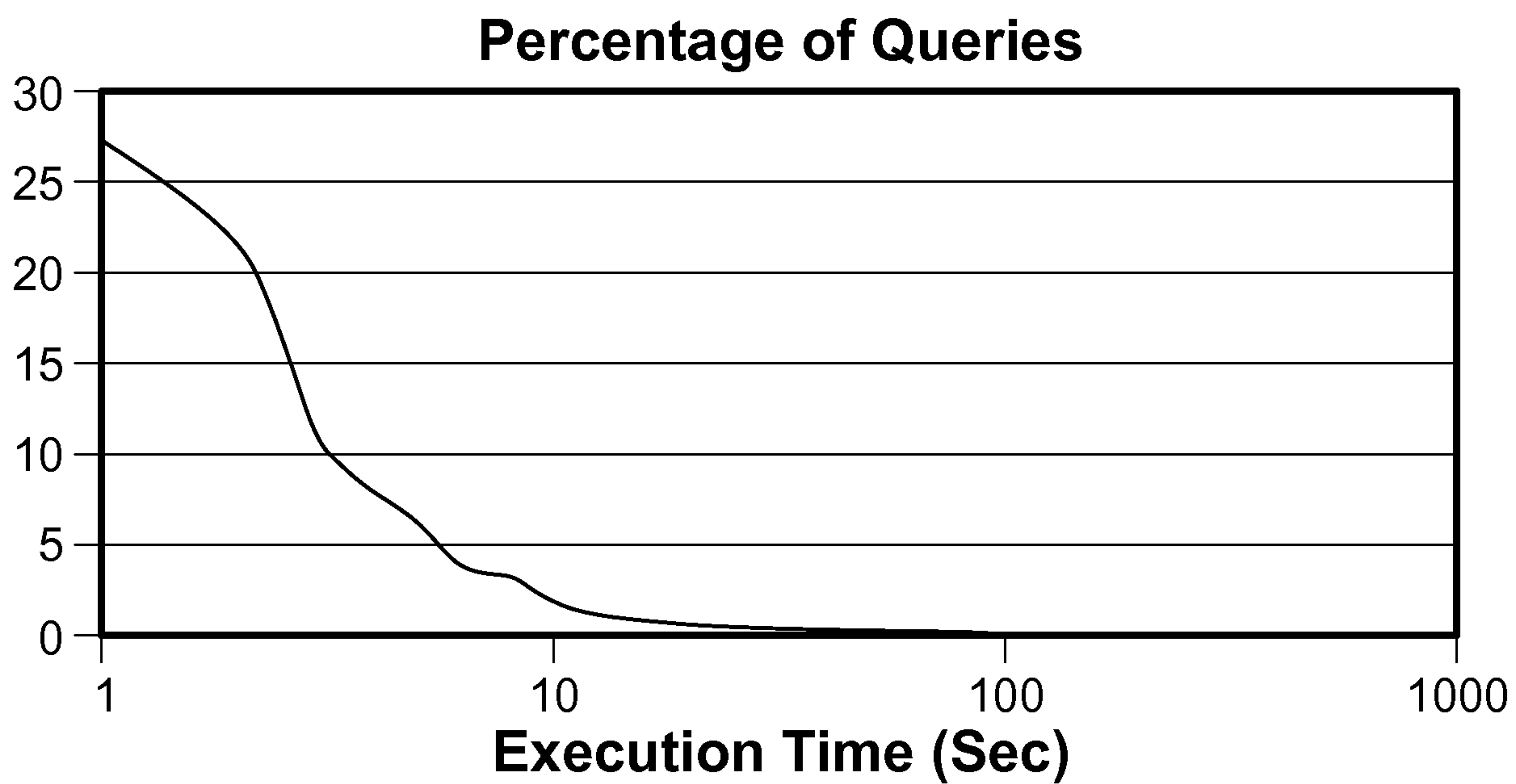
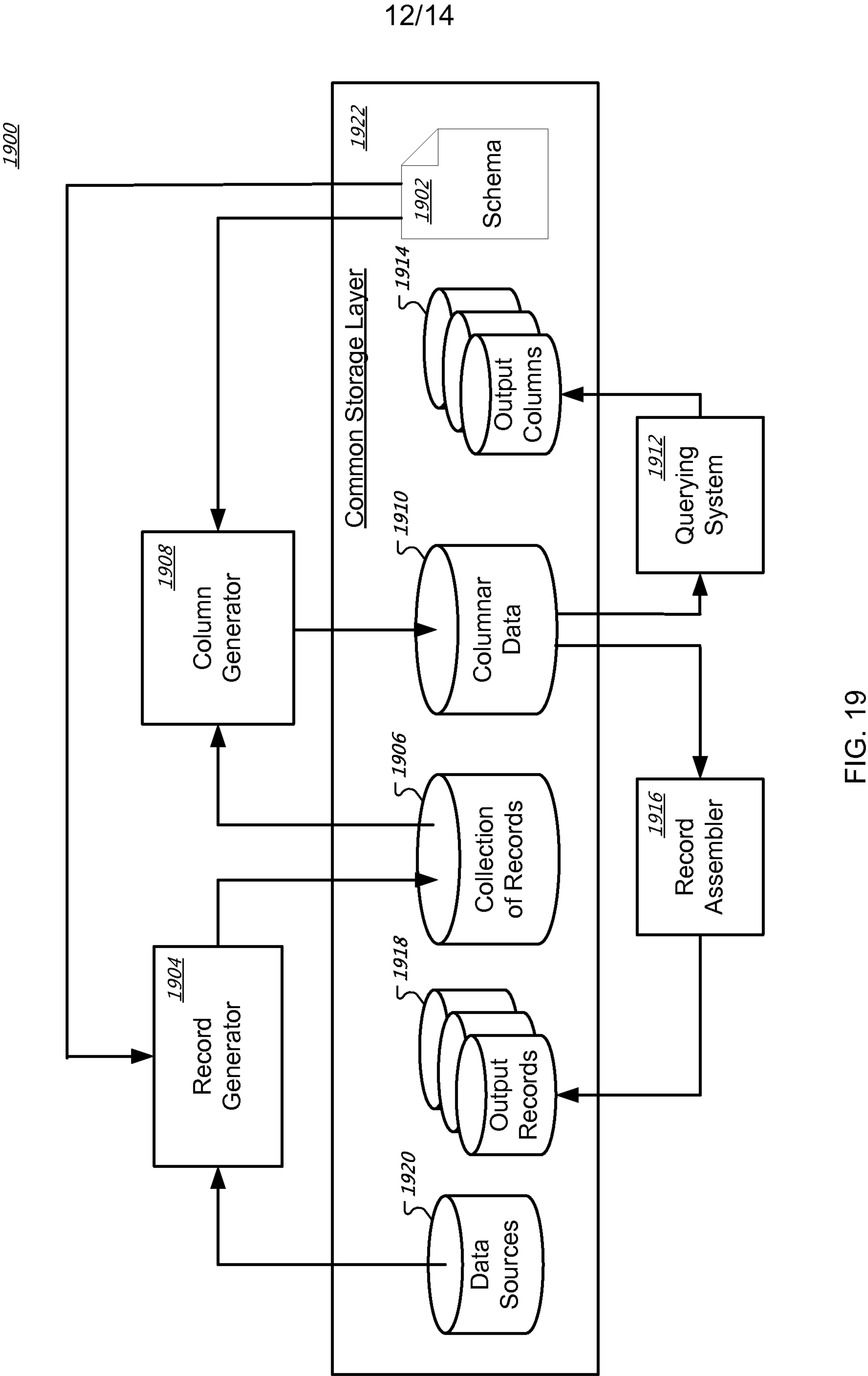


FIG. 18



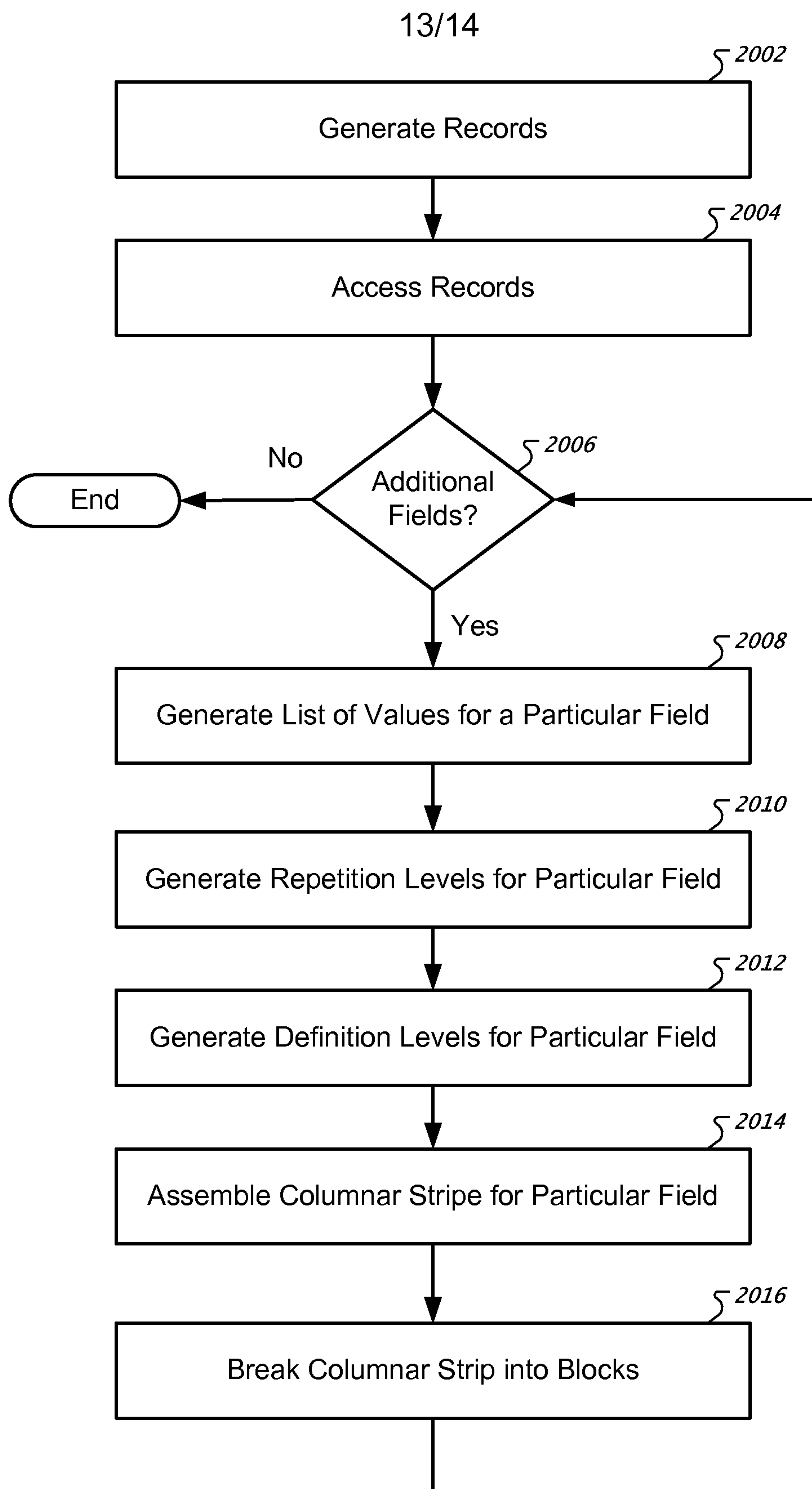


FIG. 20

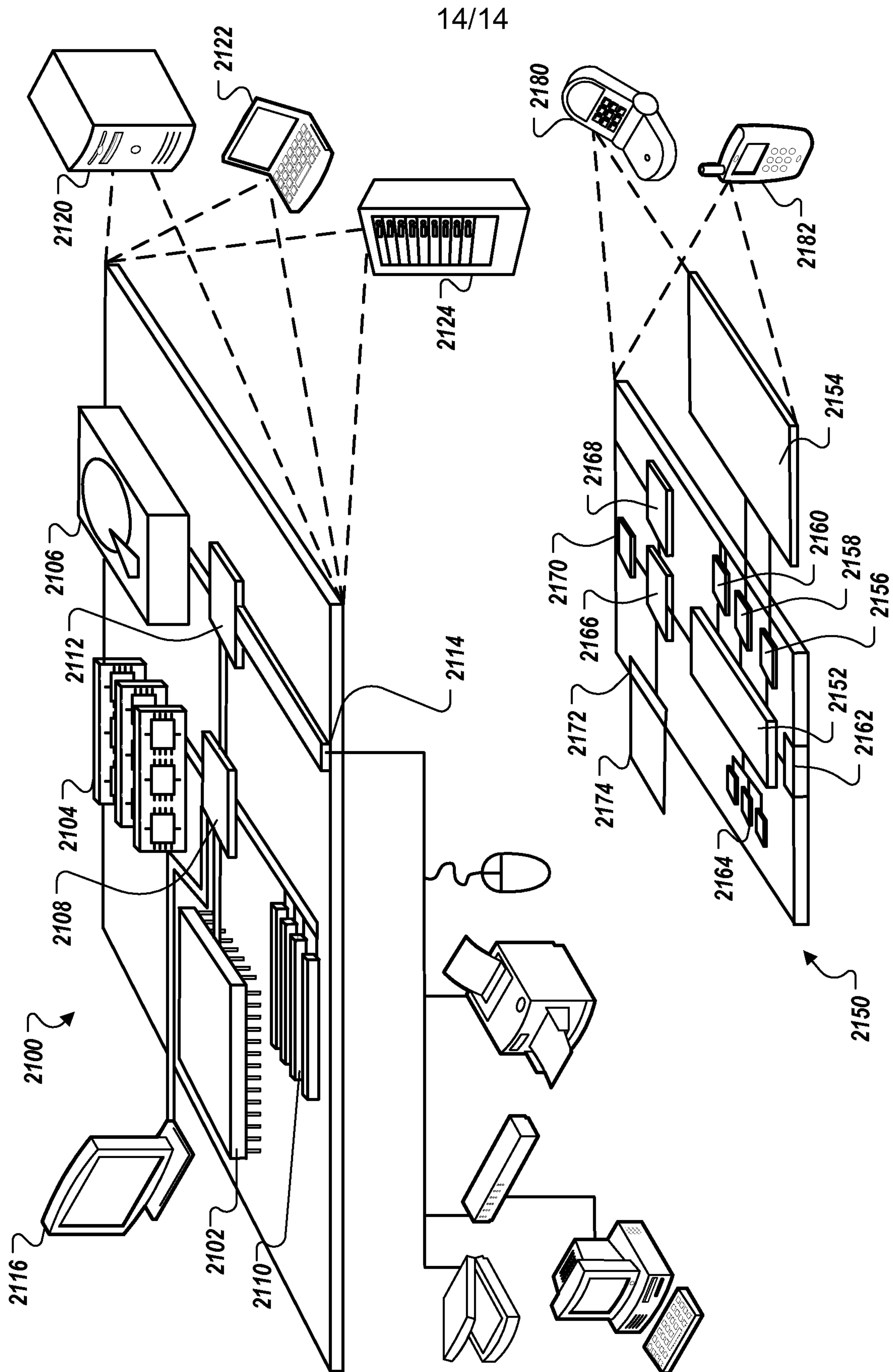
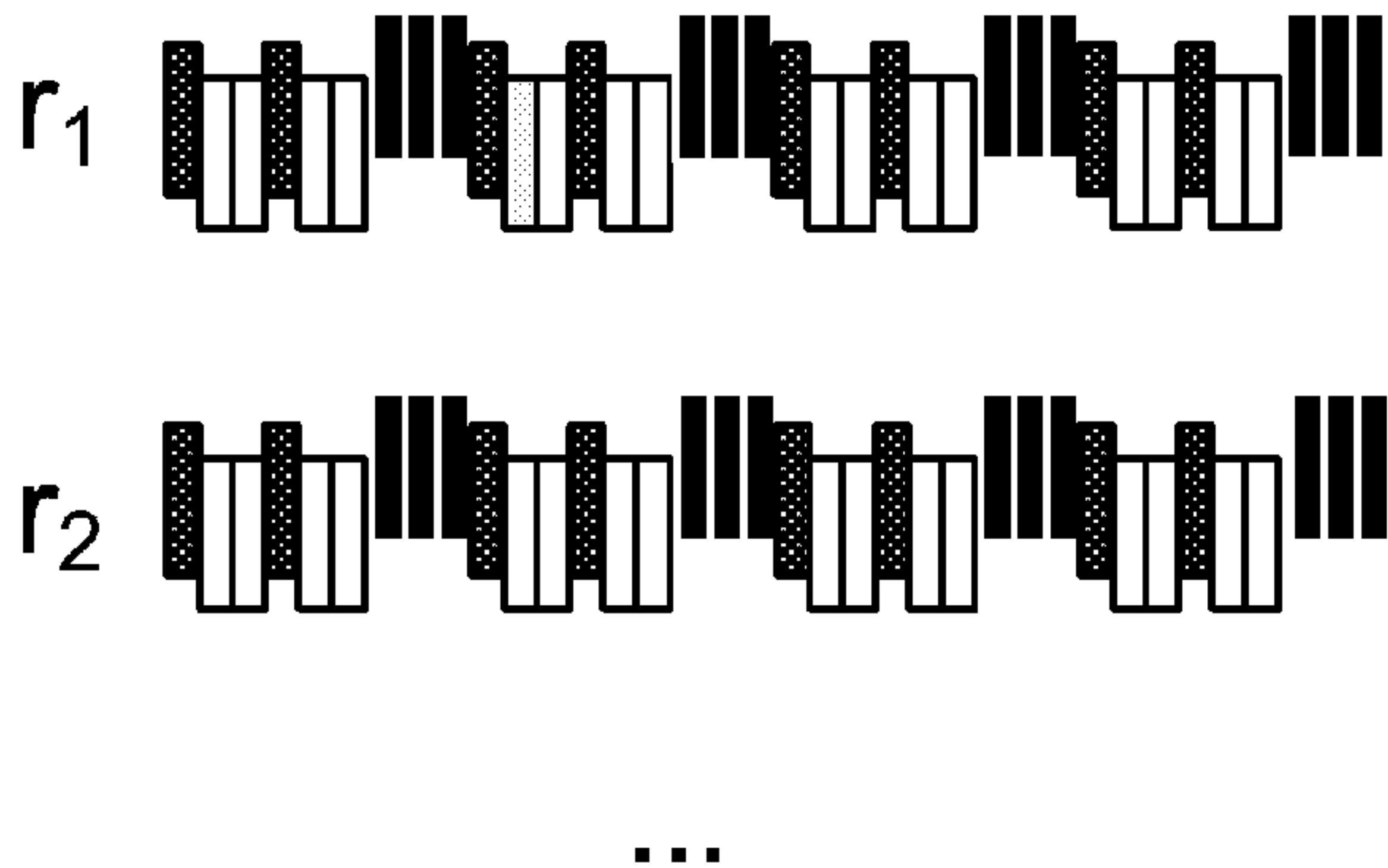


FIG. 21



record-
oriented

