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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
2 February 2017 (02.02.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/019001 A1

(51) International Patent Classification:
G06F 12/02 (2006.01)

(21) International Application Number:
PCT/US2015/042134

(22) International Filing Date:
24 July 2015 (24.07.2015)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP** [US/US]; 11445 Compaq Center Drive West, Houston, TX 77070 (US).

(72) Inventors: **VOLOS, Haris**; 1501 Page Mill Rd., Palo Alto, California 94304 (US). **LI, Jun**; 1501 Page Mill Rd., Palo Alto, California 94304 (US). **KEETON, Kimberly**; 1501 Page Mill Rd., Palo Alto, California 94304 (US). **TUCEK, Joseph**; 1501 Page Mill Rd., Palo Alto, California 94304 (US).

(74) Agents: **PAGAR, Preetam** et al.; Hewlett Packard Enterprise, 3404 E. Harmony Road, Mail Stop 79, Fort Collins, CO 80528 (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, BN, 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, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, 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, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, 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: DISTRIBUTED DATASETS IN SHARED NON-VOLATILE MEMORY

(57) Abstract: An example device includes a plurality of compute nodes; a non-volatile memory coupled to the plurality of compute nodes and storing a partitioned distributed dataset, each partition coupled to one the plurality of compute nodes; and a processor to construct a wrapper iterator object within the non-volatile memory based on a schema stored in association with the distributed dataset. The compute nodes apply a transformation to the wrapper iterator object to form a resulting distributed dataset in the non-volatile memory.

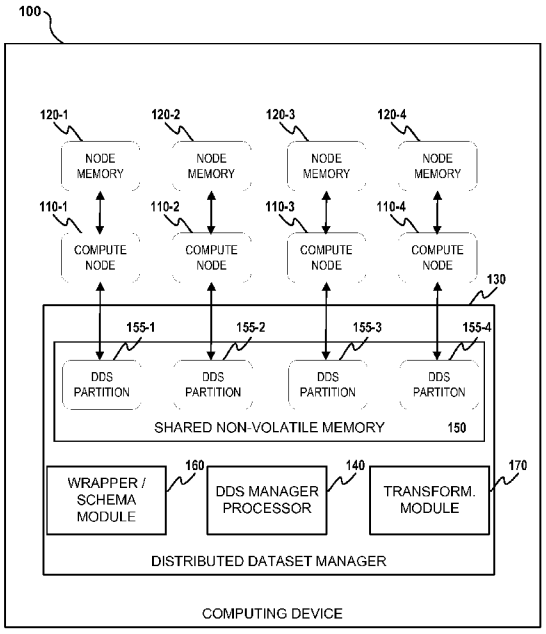

FIG. 1

WO 2017/019001 A1

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

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

Published:

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

DISTRIBUTED DATASETS IN SHARED NON-VOLATILE MEMORY

BACKGROUND

[0001] Recent work in distributed dataset computing includes a dataflow engine known as Spark (see "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," NSDI 2012, by Matei Zaharia et al.). Spark allows data transformations directly in memory (i.e., DRAM) of multiple Spark nodes. Resilient Distributed Datasets (RDDs) are collections of data partitioned across the memory of each node of a cluster.

BRIEF DESCRIPTION OF THE DRAWINGS

[0002] For a more complete understanding of various examples, reference is now made to the following description taken in connection with the accompanying drawings in which:

[0003] Figure 1 illustrates an example computing device that may utilize shared non-volatile memory for cluster computing with distributed datasets; and

[0004] Figure 2 illustrates an example flow diagram for an example process for cluster computing with distributed datasets using the computing device of Figure 1.

[0005] Figure 3 illustrates a block diagram of an example system with a computer-readable storage medium including instructions executable by a processor to forming distributed datasets in a non-volatile virtual memory.

DETAILED DESCRIPTION

[0006] Spark introduced a new distributed memory abstraction called Resilient Distributed Datasets (RDDs). RDDs are collections of data partitioned across the memory of each node of a cluster. RDDs enable programmers to perform in-memory computations in the aggregated memory of the cluster in a fault-tolerant manner. In contrast to distributed shared memory that supports fine-grain updates to shared state, RDDs support updates through coarse grain transformations such as map, reduce, and filter. Each RDD tracks its lineage, which represents the sequence of transformations used to derive the RDD. The deterministic nature of these transformations enables scalable fault tolerance. If a partition of an RDD is lost then the partition can be recovered by re-computing its parent RDD partitions based on the captured RDD transformation lineage.

[0007] Spark implements RDDs as a distributed collection of Scala/Java objects, referred to from herein as Scala objects. The Scala objects are partitioned across multiple partitions and each partition comprises multiple Scala objects. The Scala objects live in memory managed by a Java Virtual Machine (JVM) on the compute nodes in the cluster, also known as a Java Heap. RDDs are constructed by applying transformations on other RDDs (e.g., map, filter, reduceByKey, etc.). What that means is that Spark nodes iterate through each Scala object of the RDD and apply the transformations to construct the new Scala objects of the new RDD.

[0008] There are two problems with this approach that the systems and methods described herein solve. Scala objects comprising RDDs live in the Java Heap, so they have to be garbage collected (e.g., when the RDDs are out of the computing scope in the use-defined application program, or purged out of the cache managed by Spark runtime), which can cause high pressure on the garbage collector of the Spark nodes. Since the RDD lives in volatile DRAM memory, when RDDs are lost on the event of a crash, they have to be recomputed upon recovery, which can cause long execution times for recovery.

[0009] Systems and methods described herein provide for storing, processing, and analyzing large datasets. The systems and methods described herein may utilize in-situ distributed datasets (e.g., RDDs as used in Spark) that remain in a common shared non-volatile memory coupled to each of a plurality of compute nodes that implement a distributed dataset system such as, for example Spark. The use of in-situ distributed datasets enables direct in-place data manipulation of the distributed datasets stored in the non-volatile memory without first converting them into objects native to the managed language of the compute nodes (e.g., Java or Scala as used in Spark). This methodology helps avoid compute node overhead for serializing/deserializing distributed datasets, and it also reduces the number of objects managed by the compute nodes, thus reducing the pressure on the garbage collector component of the native language. It also enables the compute nodes to efficiently handle larger datasets stored in the non-volatile memory rather than datasets that fit in the smaller aggregated DRAM of the compute node cluster. Since objects are stored in unserialized byte form, transformations may be written in a different more-efficient language like C++ or even be implemented in hardware.

[0010] Referring now to the figures, Figure 1 illustrates an example computing device 100 that may utilize shared non-volatile memory for cluster computing with distributed datasets. The computing device 100 includes a plurality of compute nodes 110, four compute nodes, in this

example, including first, second, third and fourth compute nodes 110-1, 110-2, 110-3 and 110-4, respectively. The compute nodes 110 may each include a set of processing cores, caches and various ancillary peripherals. The compute nodes 110 are each respectively coupled to local node memory 120-1, 120-2, 120-3 and 120-4. The local memories 120 may be DRAM, RAM, ROM etc. The local memories may include API functions in a native language of the compute nodes, e.g., Scala and Java in the case of Spark nodes as described above.

[0011] The compute nodes 110 are communicatively coupled to a distributed dataset manager device 130. The distributed dataset manager 130 includes a distributed dataset (DDS) manager processor 140 such as a central processing unit (CPU). The distributed dataset manager 130 also includes a shared non-volatile memory 150 that stores partitions of distributed datasets or DDS 155, e.g., RDDs as used in Spark. The DDS partitions 155 include first, second, third and fourth DDS partitions 155-1, 155-2, 155-3 and 155-4, each coupled respectively to the first, second, third and fourth compute nodes 110-1, 110-2, 110-3 and 110-4.

[0012] The shared non-volatile memory 150 may be a byte addressable persistent memory and may be accessed directly, by the compute nodes 110, via normal CPU load/store instructions in units as small as one byte. The shared non-volatile memory 150 may be a resistive type of memory. The shared non-volatile memory 150 may serve as both memory and persistent storage.

[0013] Storing the DDS partitions 155 in the shared non-volatile memory 150 provides many advantages in contrast to the RDDs of Spark, which are stored in DRAM of individual Spark nodes when RDDs are applied with transformation. The RDDs of Spark rely heavily on serialization when re-distributing partitions between cluster nodes to support operations that require shuffling data between nodes, operations such as reduceBy, groupBy, and join. In contrast to the RDDs of Spark, where serialization is necessary in a cluster of machines communicating using TCP/IP, this becomes unnecessary in the context where the compute nodes 110 can exchange the DDS partitions 155 through the shared non-volatile memory 150.

[0014] The in-situ DDS partitions 155 provide for direct in-place data manipulation by the compute nodes 110 without first converting them into objects native to the managed language of the compute nodes 110. This helps avoid the CPU overhead of serializing/deserializing the DDS partitions 155 and also reduces the number of objects managed by the JVM heap, thus reducing the pressure on the JVM garbage collector.

[0015] The in-situ DDS partitions 155 enable faster recovery of RDDs. Non-volatile memory has better recovery behavior than DRAM. Once a DDS including all the DDS partitions 155 is created and stored in the non-volatile memory 150, it is immutable. If one of the compute nodes fails with immutable DDS partitions 155 in the shared non-volatile memory 150, then the DDS partitions 155 do not need to be recomputed using the transformation lineage.

[0016] In response to receiving commands from the compute nodes 110 in the native language of the compute nodes (e.g., Scala / Java), the DDS manager processor 140 provides various stub functions using a wrapper / schema module 160 and a transform module 170. The wrapper / schema module 160 and the transform module 170 may be implemented in software, firmware and/or hardware. The in-situ DDS partitions 155 are stored in shared non-volatile memory 150 following a schema provided by the wrapper / schema module 160. The schema enables representing the DDS partitions using a language independent layout, thus enabling components written in different (managed or/and unmanaged) languages to create, access, and process the in-situ DDS partitions 155 shared between them. This enables an engine like Spark to offload computation from the Scala native language to more efficient languages like C++ for performing critical steps, including, for example, shuffling operations or entire transformations such as filtering, sampling, and indexing implemented entirely in C++.

[0017] The wrapper/schema module 160 and the transform module 170 may use unsafe Java APIs to access the underlying objects stored in the in-situ DDS partitions 155 all at once. These objects and their associated schema may be automatically generated using a stub compiler that takes as input the native language objects of the compute nodes 110 (e.g., Scala objects of Spark nodes). The wrapper objects are then used by the wrapper iterators of the wrapper / schema module 160 to iterate over the elements of the in-situ DDS partitions 155 directly without generating a native language representation of the DDS in the JVM.

[0018] The DDS manager processor 140 manages DDSs stored in shared non-volatile memory 150. An entire in-situ DDS stored in the shared non-volatile memory 150 includes the DDS partitions 155, dependencies on previous in-situ DDSs, and Scala function closures implementing transformations on the dependent DDSs. The Scala function closures may be in the form of stub functions provided by the transform module 170. The stub functions interpret the unserialized bytes of the DDS partitions 155 as Native Java Objects using a schema provided by the wrapper / schema module 160.

[0019] In addition, an in-situ DDS stores a schema, as provided by the wrapper / schema module 160 that defines the fields and layout of the in-situ DDS, and potentially the function closure expressed in different languages such as C++.

[0020] An iterator is an abstraction over a collection of objects that share the same object type and undergo the same transformation. In Spark, each processing step performs a transformation over an iterator exposed by the underlying RDD. In computing device 100, the compute nodes 110 access the in-situ DDS partitions 155 through wrapper iterators provided by the wrapper / schema module 160. Wrapper iterators provided by the wrapper / schema module 160 may implement RDD APIs and enable seamless integration of in-situ DDS partitions 155 with all of the compute nodes 110. The wrapper iterators rely on the associated DDS schema to access the fields of the underlying in-situ DDS partitions 155 using, in one example, Java Unsafe APIs.

[0021] Compared to regular RDDs stored in DRAM, the in-situ DDSs stored in the non-volatile memory 150 enable direct in-place data manipulation without first converting them into objects native to the managed language (e.g. Scala, Java). In-place data manipulation has several benefits. First, it can reduce the overhead of serialization/deserialization, and it also reduces the number of objects managed by the JVM runtime, thus reducing the pressure on the JAVA garbage collector. Second, it enables Spark to efficiently handle larger datasets stored in the non-volatile memory 150 rather than datasets that fit in the smaller aggregated DRAM of the Spark nodes.

[0022] Figure 2 illustrates an example flow diagram for an example process 200 for cluster computing with distributed datasets using the computing device 100 of Figure 1. The process 200 is exemplary only and may be modified. The example process 200 of Figure 2 will now be described with further references to Figure 1.

[0023] The process 200 may begin with the DDS manager processor 140 receiving a command to apply a transformation to a partitioned distributed dataset stored in the shared non-volatile memory 150. Each partition of the distributed dataset is coupled to one of a plurality of the compute nodes 110 coupled to the shared non-volatile memory. The command is in a first language native to the compute nodes 110, block 210. The received command may be received from a programmer terminal communicatively coupled to the computing device 100. The received command may be, in various examples, in Scala language.

[0024] Upon receiving the command at block 210, the DDS manager processor 140 causes the wrapper / schema module 160 to construct a wrapper iterator object within the shared non-volatile memory 150 using a second language different than the native language and based on a schema stored in association with the distributed dataset, block 220. In various examples, the native language of the compute nodes 110 is based on Java and a JVM. The second language or languages used by the wrapper / schema module may include C++.

[0025] Rather than being described in any of several programming languages such as Java, Scala or C++, the distributed datasets are defined with one of several selected schemas. In various examples, the select schema are described in a domain-specific language that is independent of programming languages such as Java, Scala or C++ and provide basic types such as integers, strings, floating numbers, etc.

[0026] The schema language utilized at block 220 allows definition of complex types comprising an ordered list of the basic types provided by the domain-specific language of the schema. In various examples, the schema may use constructs similar to C structs such as, for example, LIST[T1, T2]. For example, LIST[X: STRING, Y: INT] defines a new complex type as a list of tuples, each of which has two named variables X and Y, where X is defined as a string and Y is defined as an integer.

[0027] For example, the schema language may provide for the definition of two kinds of in-situ DDSs of basic or complex types, as follows:

- VECTOR_IN_SITU_RDD[T], which is a vector of basic type T
- MAP_IN_SITU_RDD[(K,V)], which is an associative container storing key-value pairs

[0028] For example, the following would define an associative in-situ DDS of a string-integer key-value pairs:

MAP_IN_SITU_RDD[(STRING,INT)] (1)

[0029] This language independent declaration of types enables a schema compiler to generate a language independent binary format to represent the DDSs contained within the DDS partitions 155. In various examples, the schema employed at block 220 defines a binary byte sequence format for each basic and complex type and in-situ DDS partitions 155. The schema

may represent basic types using a binary machine representation. For example INT may be a 64-bit (8 byte) sequence. The schema may represent complex types by concatenating the byte sequence of the basic types. For example, a complex type may comprise the following schema:

BinaryFormat(LIST[X: STRING, Y: INT]) = ByteSequence(LIST[X: STRING, Y: INT]) = ByteSequence(X:STRING) + ByteSequence(Y:INT) = X.LEN + X.CHAR[0] + X.CHAR[1] + Y (2)

[0030] The complex type defined by the schema equation (2) can be depicted in table form by the following Tables 1 and 2:

Table 1

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4 …	Byte N
X.LEN	X.CHAR [0]	X.CHAR [1]	X.CHAR [2]	…	X.CHAR [N-1]

Table 2

Byte N+1	Byte N+2		Byte N+8
Y.BYTE [0]	Y.BYTE [1]	…	Y.BYTE [7]

Where N = X.LEN

[0031] The binary format of the in-situ DDS comprises all the elements of all the DDS partitions 155, with each partition being constructed by concatenating the byte sequences of each element, with the addition of a byte sequence EOF, or other known byte sequence, at the end. The EOF byte sequence marks the termination of the DDS partition. For example, the binary representation of an associative in-situ DDS partition of string-integer key-value pairs given in equation (1) above (MAP_IN_SITU_RDD[(X: STRING,Y: INT)]) with two elements would be:

Table 3

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4 …	Byte N
X.LEN	X.CHAR [0]	X.CHAR [1]	X.CHAR [2]	…	X.CHAR [N-1]

Table 4

Byte N+1	Byte N+2		Byte N+8
Y.BYTE [0]	Y.BYTE [1]	…	Y.BYTE [7]

Table 5

Byte N+9	Byte N+10	Byte N+11	Byte N+12	Byte N+13	Byte M+N+9
X.LEN	X.CHAR [0]	X.CHAR [1]	X.CHAR [2]	…	X.CHAR [M-1]

Table 6

Byte M+N+10	Byte M+N+11		Byte M+N+17
Y.BYTE [0]	Y.BYTE [1]	…	Y.BYTE [7]

Table 7

Byte M+N+18
EOF

Where N and M are the string lengths of the first and second elements respectively

[0032] Returning to block 220, when the proper schema has been generated, a stub compiler in the wrapper / schema module 160 takes as input the generated schema of the in-situ DDS selected at block 210 and generates a wrapper iterator object in a second language (e.g., C++) that is different that the native language of the compute nodes 110 (e.g., Java or Scala) for iterating through all the elements of the in-situ DDS including all the DDS partitions 155. The wrapper iterator object knows how to interpret the underlying language-independent binary format of the in-situ DDS. A single wrapper iterator object (e.g., Java or Scala) is used to access each element of the entire in-situ DDS including all the DDS partitions 155, rather than having multiple Java or Scala objects each representing one element in Spark. The wrapper iterator constructed at block 220 leverages the property that transformations on the in-situ DDSs stored in the shared non-volatile memory 150 are done on all elements of the in-situ DDS for all the

DDS partitions 155 and that all elements of the in-situ DDS have the same type. For this reason, the wrapper / schema module 160 interprets the schema for the DDS selected at block 210 once and not each time it accesses a DDS element. This in effect reduces the CPU time spent in interpreting types.

[0033] Subsequent to the wrapper iterator object being constructed at block 220, the compute nodes 110 apply the selected transformation received at block 210 to the wrapper iterator object constructed at block 220 to form a resulting distributed dataset in the shared non-volatile memory 150. The resulting distributed dataset is formed and stored directly in the shared non-volatile memory 150. When the transformation has been applied to all elements of all the DDS partitions 155, the DDS manager processor 140 causes the wrapper / schema iterator module 160 to store the resulting distributed dataset in a format similar to the original DDS using the same schema or a different schema, depending on the transformation.

[0034] The transformation applied at block 230 may be a transformation object in the native language of the compute nodes 110. In various examples, the transformation applied at block 230 may be a transformation object in a language other than the native language of the compute nodes 110, as described above. In these various examples, the DDS manager processor may cause the transform module 170 to construct the selected transformation object in at least one of the second language of the wrapper iterator object or a third language different than the native language. The compute nodes 110 would then apply the constructed transformation object to the wrapper iterator object to form the resulting distributed dataset in the shared non-volatile memory 150.

[0035] The transformation applied at block 230 may include one or more of a filter transformation, a flat map transformation, a map transformation, a reduceByKey transformation, etc. A filter transformation selects elements for which a given function f evaluates to true. For example, a filter (f ➔ x=='the') searches for a specified string of characters. A reduceByKey transformation shuffles key-value pairs to reducers where pairs with the same key end up at the same reducer. For example, in reduceByKey (f ➔ x + y) each reducer sums up the values of pairs with same word key. A flatMap transformation splits words and a map transformation maps each word to a key-value pair such as (word, 1).

[0036] For example, applying a filter transformation such as, for example, filter(X= ="the") to the associative in-situ DDS of equation (1) above may be as follows:

MAP_IN_SITU_RDD[(X: STRING,Y: INT)].filter (X == “the”) (3)

[0037] The wrapper iterator object constructed at block 220 interprets the type of each element in the DDS partitions 155 as *(STRING, INT)* once at the beginning and then applies the transformation *filter (X == “the”)* on each element. Since the selected schema defines a binary format, the interpretation of the in-situ DDS partitions 155 essentially determines the beginning and end of each field in the byte sequence. So iteration through the elements of the DDS partitions 155 is essentially iteration through the byte sequence of the DDS and interpretation of each DDS field into a basic or complex type according to the schema until EOF is found. In this case, the elements would be interpreted by a wrapper iterator such as:

INTERPRET_RDD_ELEMENT(STRING, INT) (4)

[0038] The same wrapper iterator object constructed at block 220 is thus reused for interpreting each element of the DDS partitions 155 rather than reconstructed for each element. The wrapper iterator object, in this example, interprets the elements of the DDS partitions 155 as follows:

X1 = INTERPET_STRING(LEN,RDD.BYTE_SEQUENCE[0],
RDD.BYTE_SEQUENCE[0]… RDD.BYTE_SEQUENCE[N]) (5)

Y2 = INTERPRET_INTEGER(RDD.BYTE_SEQUENCE[N+1] … (6)
RDD.BYTE_SEQUENCE[N+8])

X1 = INTERPRET_STRING(LEN,RDD.BYTE_SEQUENCE[N+9],
(7)
RDD.BYTE_SEQUENCE[N+10]… RDD.BYTE_SEQUENCE[M+N+9])

Y2 = INTERPRET_INTEGER(RDD.BYTE_SEQUENCE[M+N+10] … (8)
RDD.BYTE_SEQUENCE[M+N+17])

[0039] Where INTERPRET_STRING(LEN, CHARS) interprets a byte array CHARS of length LEN as a STRING and INTERPRET_INTEGER(BYTES) interprets a byte array BYTES as a 64-bit INT.

[0040] Application of the transformation at block 230 proceeds one element at a time. At decision block 240, each of the compute nodes 110 determines if the last element of the DDS partition 155 has been reached. If the last element has not been reached, application of the transformation at block 230 continues. If the last element has been reached, the process 200 continues to block 250 where the compute nodes 110 release the wrapper iterator object. As described above, when the transformation has been applied to all elements of all the DDS partitions 155, the DDS manager processor 140 causes the wrapper / schema iterator module 160 to store the resulting distributed dataset in a format similar to the original DDS using the same schema or a different schema, depending on the transformation. When the resulting distributed dataset has been stored, the process 200 concludes.

[0041] Figure 3 illustrates a block diagram of an example system with a computer-readable storage medium including instructions executable by a processor to forming distributed datasets in a non-volatile virtual memory. The system 300 includes the processor 310 and the computer-readable storage medium 320. The computer-readable storage medium 320 includes example instructions 321-323 executable by the processor 310 to perform various functionalities described herein.

[0042] The example instructions includes receiving command to apply a transformation instructions 321. The instructions 321 cause the processor 310 to receive a command to apply a transformation to a partitioned distributed dataset stored in non-volatile memory. Each partition may be coupled to one of a plurality of compute nodes coupled to the non-volatile memory. The command may be in a first language native to the compute nodes.

[0043] The example instructions 322 cause the processor 310 to construct a wrapper iterator object within the non-volatile memory using a second language different than the native language and based on a schema stored in association with the distributed dataset. The example instructions further include applying the transformation to the wrapper instructions 323. The example instructions 323 cause the processor 310 to apply the transformation to the wrapper iterator object to form a resulting distributed dataset in the non-volatile memory.

[0044] Various examples described herein are described in the general context of method steps or processes, which may be implemented in one example by a software program product or component, embodied in a machine-readable medium, including executable instructions, such as program code, executed by entities in networked environments. Generally, program modules may include routines, programs, objects, components, data structures, etc. which may be designed to perform particular tasks or implement particular abstract data types. Executable instructions, associated data structures, and program modules represent examples of program code for executing steps of the methods disclosed herein. The particular sequence of such executable instructions or associated data structures represents examples of corresponding acts for implementing the functions described in such steps or processes.

[0045] Software implementations of various examples can be accomplished with standard programming techniques with rule-based logic and other logic to accomplish various database searching steps or processes, correlation steps or processes, comparison steps or processes and decision steps or processes.

[0046] The foregoing description of various examples has been presented for purposes of illustration and description. The foregoing description is not intended to be exhaustive or limiting to the examples disclosed, and modifications and variations are possible in light of the above teachings or may be acquired from practice of various examples. The examples discussed herein were chosen and described in order to explain the principles and the nature of various examples of the present disclosure and its practical application to enable one skilled in the art to utilize the present disclosure in various examples and with various modifications as are suited to the particular use contemplated. The features of the examples described herein may be combined in all possible combinations of methods, apparatus, modules, systems, and computer program products.

[0047] It is also noted herein that while the above describes examples, these descriptions should not be viewed in a limiting sense. Rather, there are several variations and modifications which may be made without departing from the scope as defined in the appended claims.

WHAT IS CLAIMED IS:

1. A device, comprising:

a plurality of compute nodes;

a non-volatile memory coupled to the plurality of compute nodes and storing a partitioned distributed dataset, each partition coupled to one the plurality of compute nodes;

a processor to construct a wrapper iterator object within the non-volatile memory based on a schema stored in association with the distributed dataset,

wherein the compute nodes apply a transformation to the wrapper iterator object to form a resulting distributed dataset in the non-volatile memory.

2. The device of claim 1, wherein the processor is to receive a command to apply the transformation to the partitioned dataset in a first language native to the compute nodes, and wherein the wrapper iterator object is constructed using a second language different from the first language.

3. The device of claim 2, wherein the transformation in the received command requires a shuffle operation when performed in the native language of the compute nodes and the constructed transformation requires no data serialization.

4. The device of claim 1, wherein the processor is further to store the resulting distributed dataset in a format based on at least one of the schema stored in association with the distributed dataset or a different schema.

5. The device of claim 1, wherein the schema comprises a language-independent binary byte sequence format and the wrapper iterator object is constructed to interpret the language-independent binary byte sequence format of the schema.

6. A method, comprising:

receiving a command to apply a transformation to a partitioned distributed dataset stored in non-volatile memory, each partition coupled to one a plurality of compute nodes coupled to

the non-volatile memory, the command being in a first language native to the compute nodes;

constructing a wrapper iterator object within the non-volatile memory using a second language different than the native language and based on a schema stored in association with the distributed dataset; and

applying the transformation to the wrapper iterator object to form a resulting distributed dataset in the non-volatile memory.

7. The method of claim 6, wherein applying the transformation comprises constructing the transformation in at least one of the second language or a third language different than the native language and applying the constructed transformation to the wrapper iterator object to form the resulting distributed dataset in the non-volatile memory.

8. The method of claim 7 wherein the transformation in the received command requires a shuffle operation when performed in the native language of the compute nodes and the constructed transformation requires no data serialization.

9. The method of claim 6, further comprising storing the resulting distributed dataset in a format based on at least one of the schema stored in association with the distributed dataset or a different schema.

10. The method of claim 6, wherein the schema comprises a language-independent binary byte sequence format and the wrapper iterator object is constructed to interpret the language-independent binary byte sequence format of the schema.

11. A computer program product, embodied on a non-transitory computer-readable medium, comprising:

computer code for receiving a command to apply a transformation to a partitioned distributed dataset stored in non-volatile memory, each partition coupled to one of a plurality of compute nodes coupled to the non-volatile memory, the command being in a first language native to the compute nodes;

computer code for constructing a wrapper iterator object within the non-volatile memory

using a second language different than the native language and based on a schema stored in association with the distributed dataset; and

computer code for applying the transformation to the wrapper iterator object to form a resulting distributed dataset in the non-volatile memory.

12. The computer program product of claim 11, wherein the computer code for constructing the transformation comprises computer code for constructing the transformation in at least one of the second language or a third language different than the native language and the computer code for applying the transformation comprises computer code for applying the constructed transformation to the wrapper iterator object to form the resulting distributed dataset in the non-volatile memory

13. The computer program product of claim 12, wherein the transformation in the received command would require a shuffle operation when performed in the native language of the compute nodes and the constructed transformation requires no data serialization.

14. The computer program product of claim 11 further comprising computer code for storing the resulting distributed dataset in a format based on at least one of the schema stored in association with the distributed dataset or a different schema.

15. The computer program product of claim 11, wherein the schema comprises a language-independent binary byte sequence format and the wrapper iterator object is constructed to interpret the language-independent binary byte sequence format of the schema.

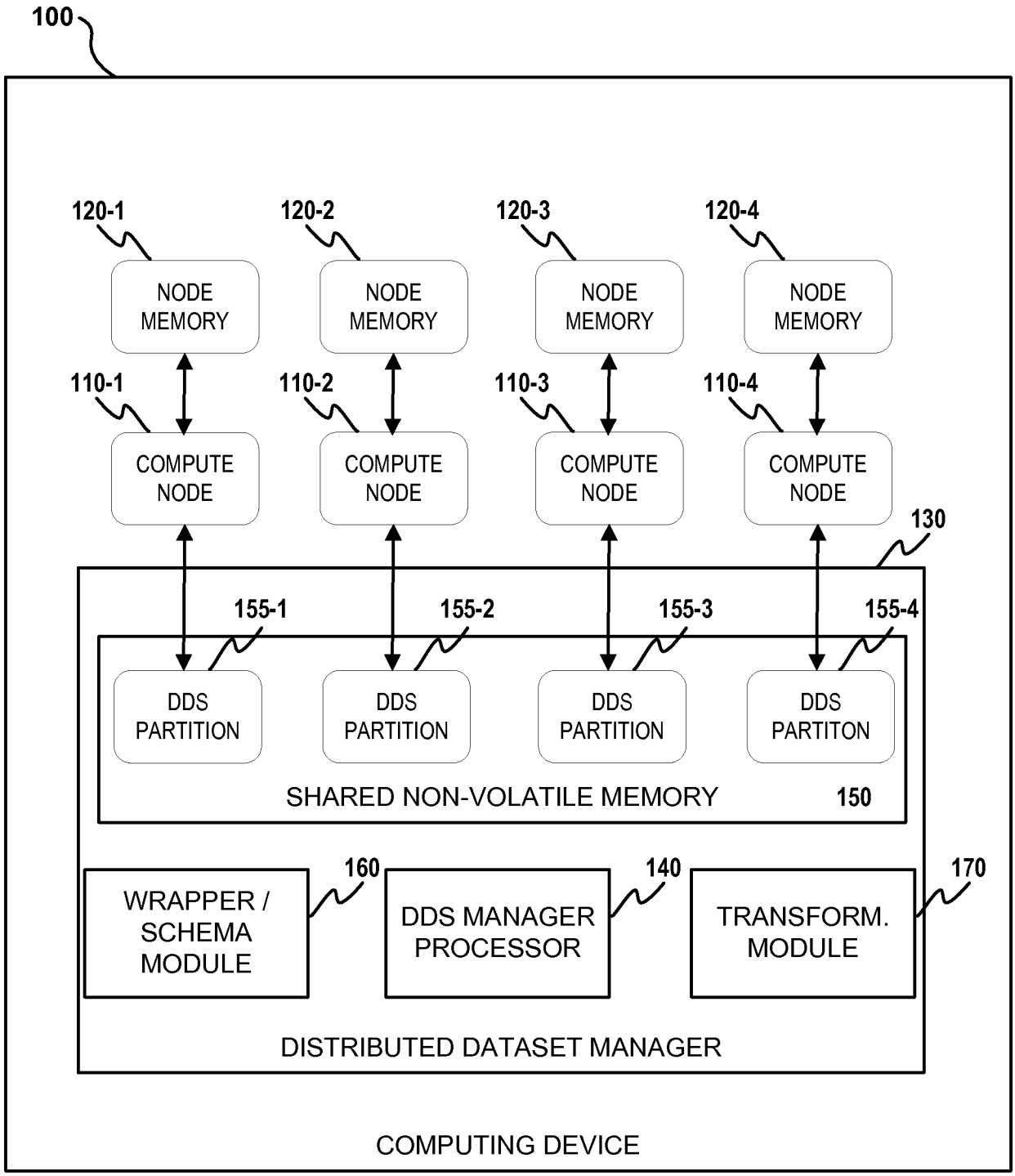
100
120-1
120-2
120-3
120-4
NODE MEMORY
NODE MEMORY
NODE MEMORY
NODE MEMORY
110-1
110-2
110-3
110-4
COMPUTE NODE
COMPUTE NODE
COMPUTE NODE
COMPUTE NODE
130
155-1
155-2
155-3
155-4
DDS PARTITION
DDS PARTITION
DDS PARTITION
DDS PARTITON
SHARED NON-VOLATILE MEMORY
150
160
140
170
WRAPPER / SCHEMA MODULE
DDS MANAGER PROCESSOR
TRANSFORM. MODULE
DISTRIBUTED DATASET MANAGER
COMPUTING DEVICE

FIG. 1

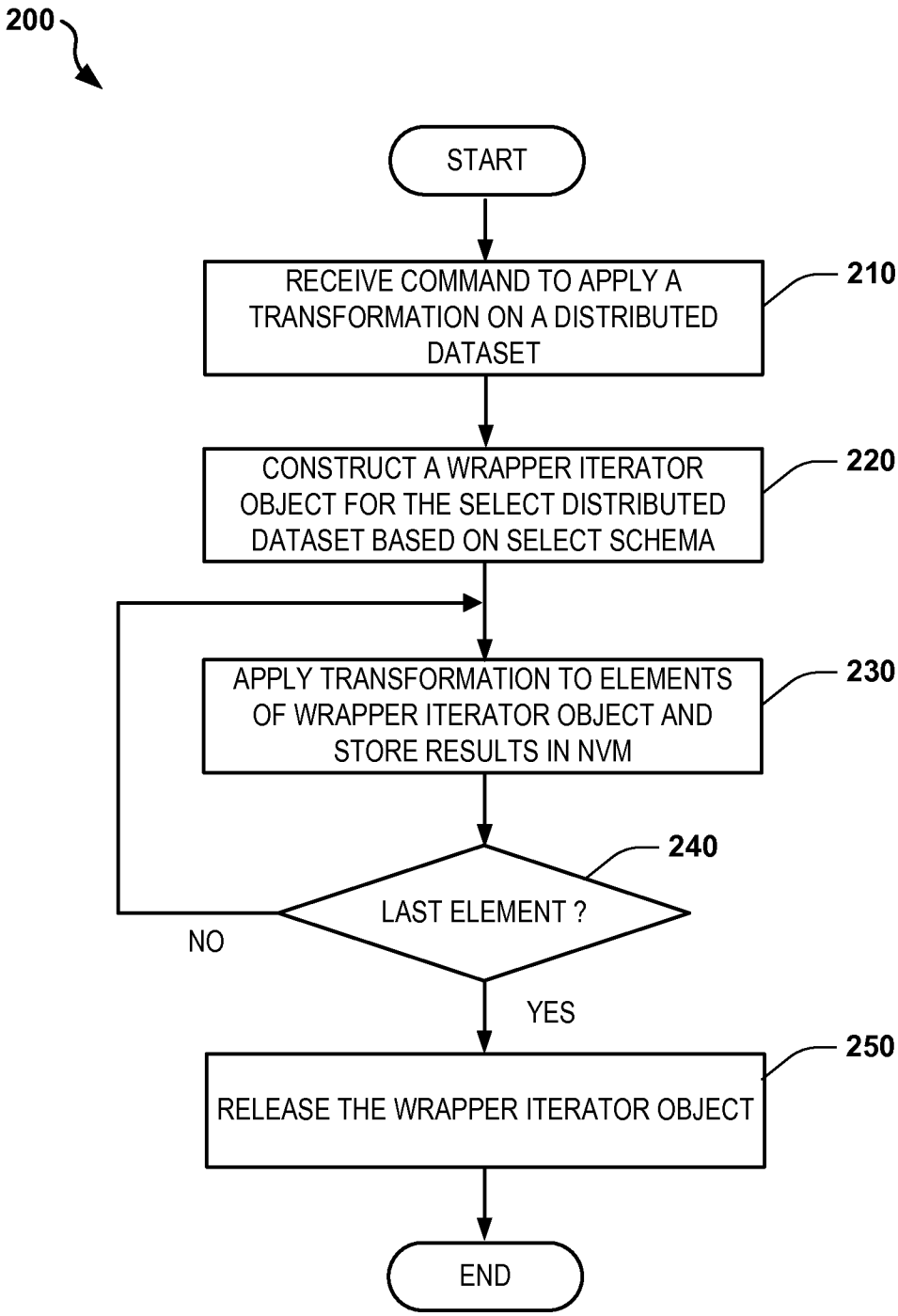
200
START
RECEIVE COMMAND TO APPLY A TRANSFORMATION ON A DISTRIBUTED DATASET
210
CONSTRUCT A WRAPPER ITERATOR OBJECT FOR THE SELECT DISTRIBUTED DATASET BASED ON SELECT SCHEMA
220
APPLY TRANSFORMATION TO ELEMENTS OF WRAPPER ITERATOR OBJECT AND STORE RESULTS IN NVM
230
LAST ELEMENT ?
240
NO
YES
RELEASE THE WRAPPER ITERATOR OBJECT
250
END

FIG. 2

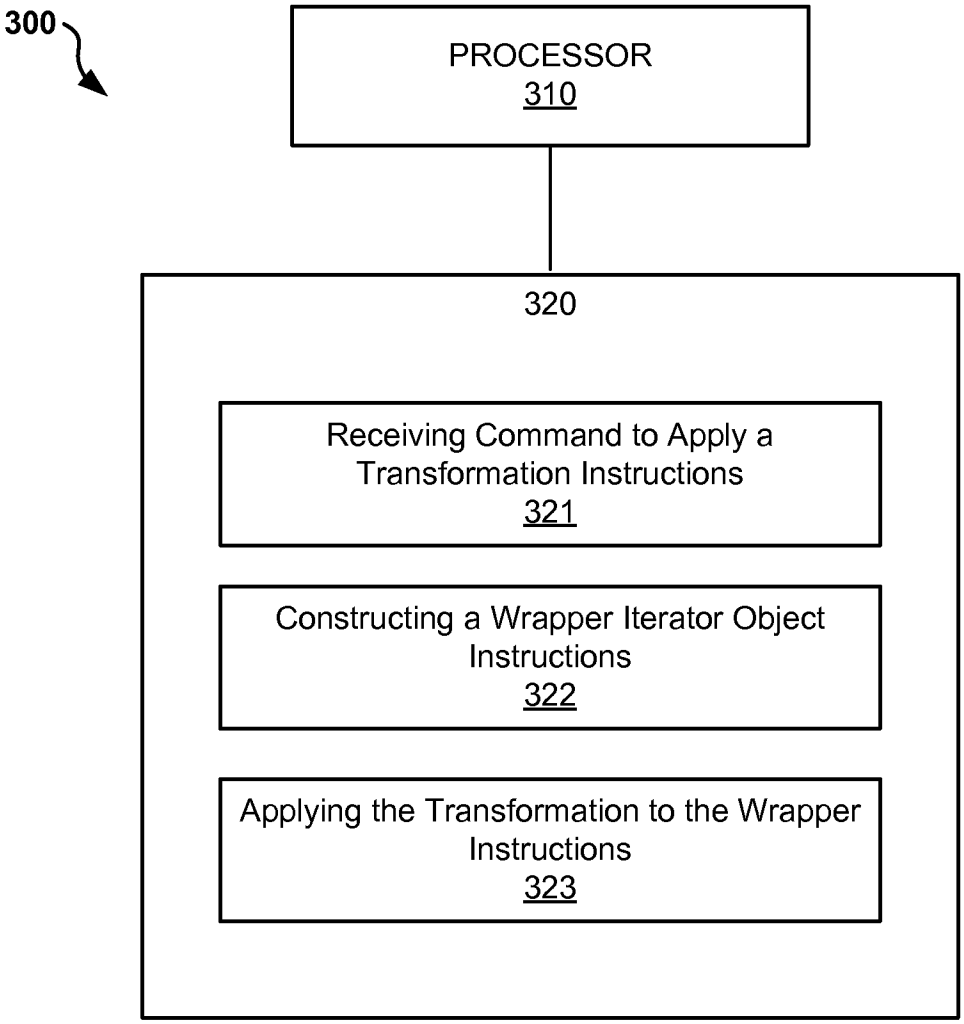
300
PROCESSOR
310
320
Receiving Command to Apply a Transformation Instructions
321
Constructing a Wrapper Iterator Object Instructions
322
Applying the Transformation to the Wrapper Instructions
323

FIGURE 3

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2015/042134

A. CLASSIFICATION OF SUBJECT MATTER

G06F 12/02(2006.01)i

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 12/02; G06F 7/00; G06F 17/30; G06F 12/14; G06F 9/455; G09G 5/00; G06F 9/45

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: nonvolatile, memory, wrapper, transform, language, format, partition, data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2012-0011500 A1 (PAOLO FARABOSCHI et al.) 12 January 2012 See paragraphs [0012]-[0016], [0023], [0034]; claim 1; and figures 1, 3B.	1-15
Y	US 2010-0161678 A1 (VIDHYASHANKARAN RAMAMOORTHY IYER et al.) 24 June 2010 See paragraphs [0004], [0018]; and figure 3.	1-15
Y	US 2014-0055496 A1 (DAVID CUNNINGHAM et al.) 27 February 2014 See paragraphs [0117], [0143]; and figure 1.	3,5,8,10,13,15
A	US 2011-0289490 A1 (JAMES A. MCATAMNEY) 24 November 2011 See paragraphs [0013], [0049]; and figure 1.	1-15
A	WO 02-35395 A2 (ENTIGEN CORPORATION) 02 May 2002 See paragraphs [0020], [0077]; and figure 3b.	1-15

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
22 April 2016 (22.04.2016)	**22 April 2016 (22.04.2016)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea
Facsimile No. +82-42-481-8578

Authorized officer
CHIN, Sang Bum
Telephone No. +82-42-481-8398

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2015/042134

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2012-0011500 A1	12/01/2012	US 08812400 B2	19/08/2014
US 2010-0161678 A1	24/06/2010	None	
US 2014-0055496 A1	27/02/2014	US 2014-059552 A1	27/02/2014
US 2011-0289490 A1	24/11/2011	US 8533690 B2	10/09/2013
WO 02-35395 A2	02/05/2002	AU 2002-228739 A8 US 2002-133504 A1 WO 02-035395 A3	06/05/2002 19/09/2002 13/02/2003