US 20180067869A1

(19) **United States**

(12) **Patent Application Publication**

Yang et al.

(10) **Pub. No.: US 2018/0067869 A1**

(43) **Pub. Date: Mar. 8, 2018**

(54) **IN-MEMORY SHARED DATA REUSE REPLACEMENT AND CACHING**

(71) Applicant: **Samsung Electronics Co., Ltd.**, Suwon-si (KR)

(72) Inventors: **Zhengyu Yang**, Boston, MA (US); **Jiayin Wang**, Dorchester, MA (US); **Thomas David Evans**, San Marcos, CA (US)

(21) Appl. No.: **15/404,121**

(22) Filed: **Jan. 11, 2017**

Related U.S. Application Data

(60) Provisional application No. 62/384,078, filed on Sep. 6, 2016.

Publication Classification

(51) **Int. Cl.**

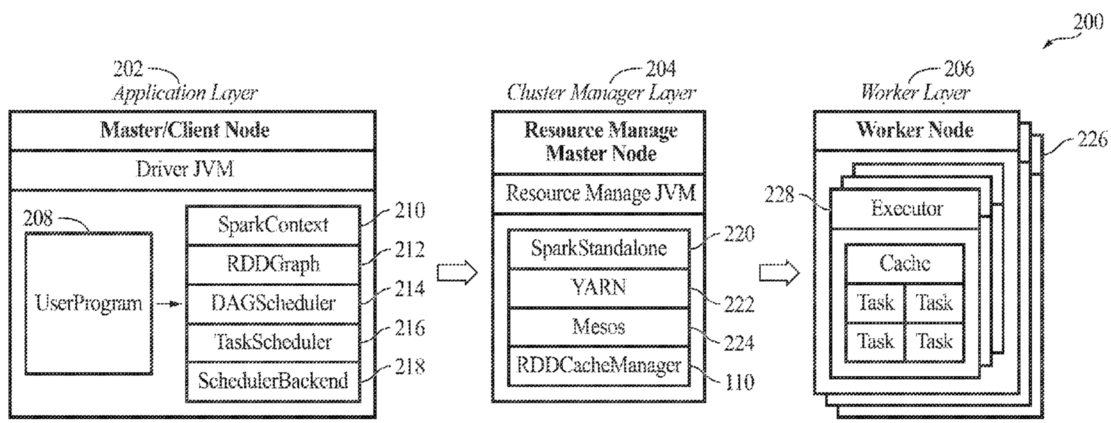
G06F 12/126	(2006.01)
G06F 12/0808	(2006.01)
G06F 12/0842	(2006.01)

(52) **U.S. Cl.**

CPC ***G06F 12/126*** (2013.01); ***G06F 12/0808*** (2013.01); *G06F 2212/69* (2013.01); *G06F 2212/1021* (2013.01); ***G06F 12/0842*** (2013.01)

(57) **ABSTRACT**

A cache management system for managing a plurality of intermediate data includes a processor and a memory having stored thereon instructions that cause the processor to perform identifying a new intermediate data to be accessed, loading the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data, in response to not identifying the new intermediate data as one of the plurality of intermediate data, selecting a set of victim intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data, the scores being based on a score table, evicting the set of victim intermediate data from the memory, updating the score table based on the set of victim intermediate data, and adding the new intermediate data to the plurality of intermediate data stored in the memory.



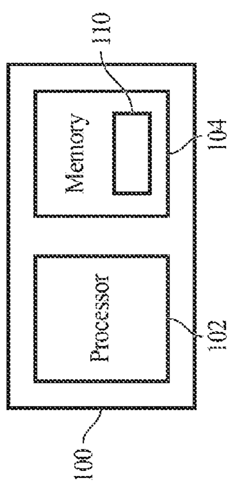

100
Processor
102
Memory
110
104

FIG. 1

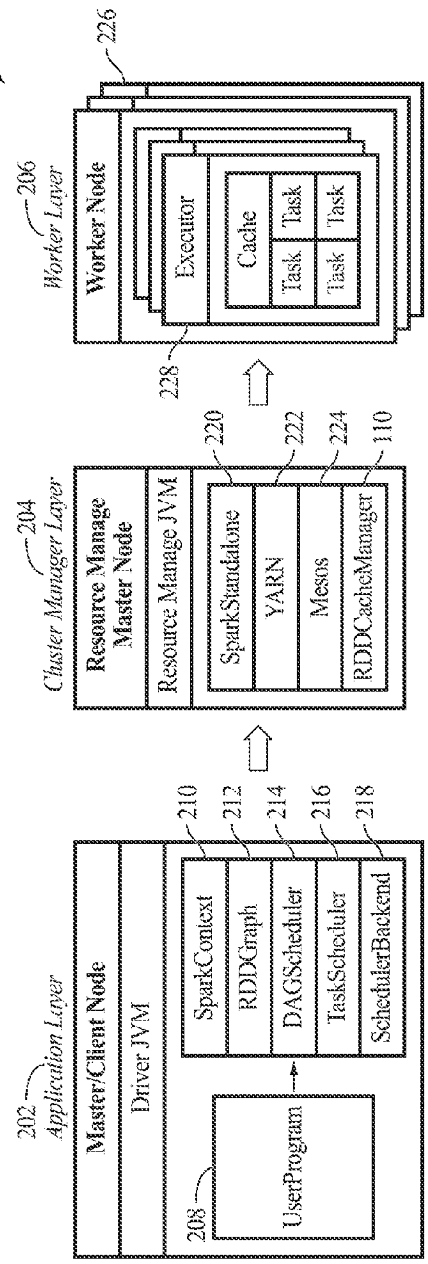

200
202
Application Layer
Master/Client Node
Driver JVM
208
UserProgram
SparkContext
RDDGraph
DAGScheduler
TaskScheduler
SchedulerBackend
210
212
214
216
218
204
Cluster Manager Layer
Resource Manage Master Node
Resource Manage JVM
SparkStandalone
YARN
Mesos
RDDCacheManager
220
222
224
110
206
Worker Layer
Worker Node
Executor
Cache
Task
Task
Task
Task
226
228

FIG. 2

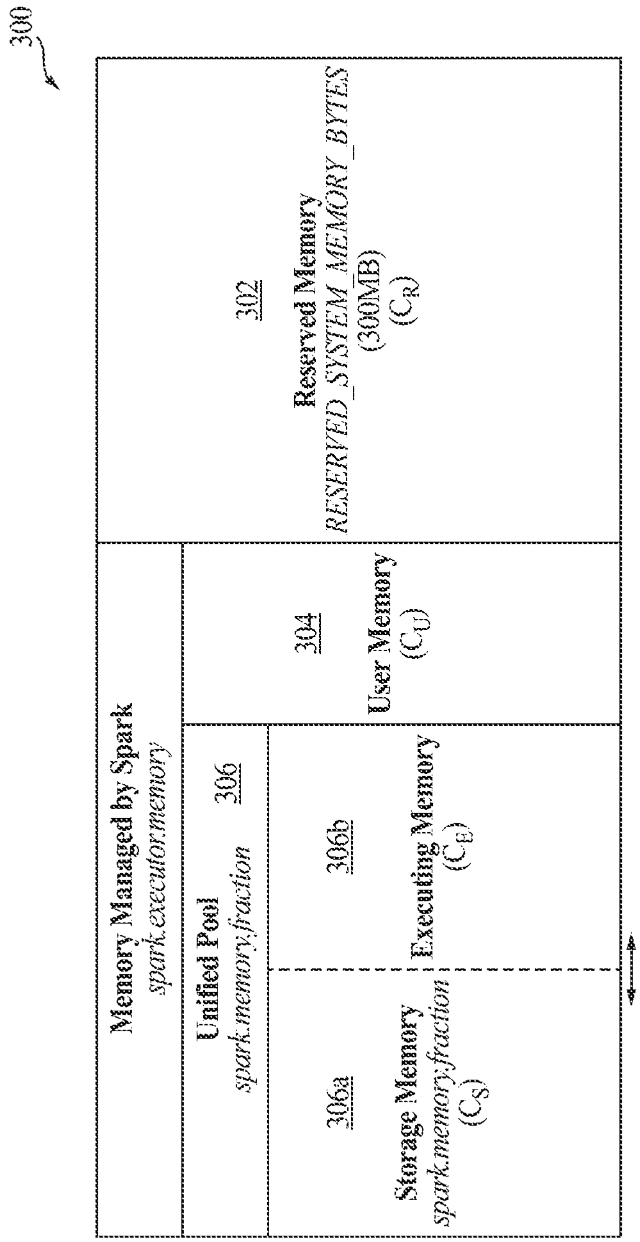

300
302
Reserved Memory
RESERVED_SYSTEM_MEMORY_BYTES
(300MB)
(C_R)
Memory Managed by Spark
spark.executor.memory
304
User Memory
(C_U)
Unified Pool
spark.memory.fraction
306
306a
Storage Memory
spark.memory.fraction
(C_S)
306b
Executing Memory
(C_E)

FIG. 3

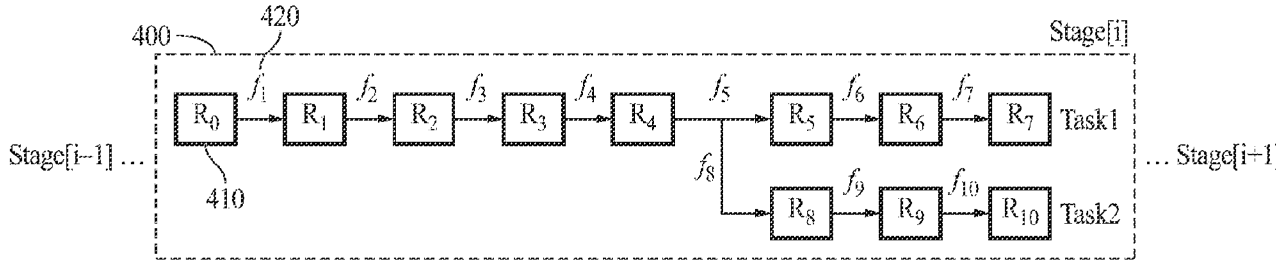


FIG. 4A

430

RDD	GenChain	Child# (N_C)	Size(|R|)	GenTime (T_G)
R0	R0	1	|R0|	0
R1	f1(R0)	1	|R1|	T[0→1]
R2	f2(f1(R0))	1	|R2|	T[1→2]
R3	f3(f2(f1(R0)))	1	|R3|	T[2→3]
R4	f4(f3(f2(f1(R0))))	2	|R4|	T[3→4]
R5	f5(f4(f3(f2(f1(R0)))))	1	|R5|	T[4→5]
…	……	…	…	
R10	f10(f9(f8(…)))	0	|R10|	T[9→10]

FIG. 4B

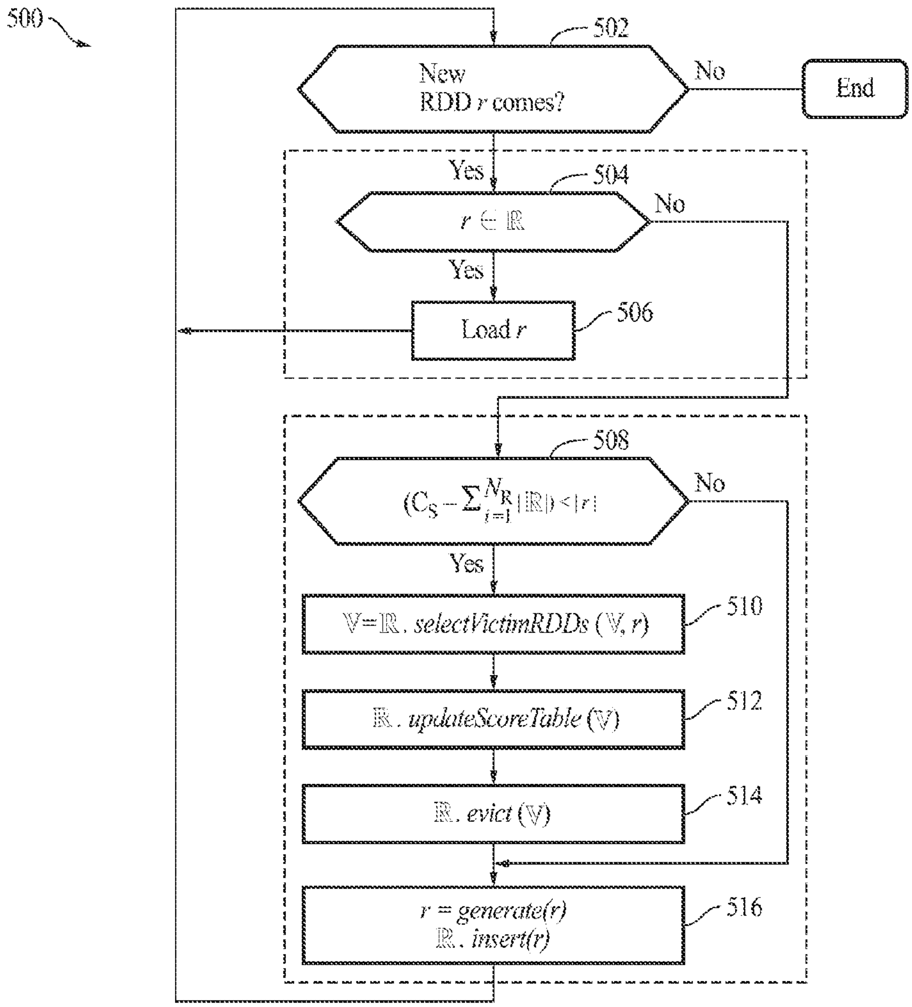

500
502
New
RDD r comes?
No
End
Yes
504
$r \in \mathbb{R}$
No
Yes
Load r
506
508
$(C_S - \sum_{i=1}^{N_R} |\mathbb{R}|) < |r|$
No
Yes
$\mathbb{V}=\mathbb{R}.selectVictimRDDs(\mathbb{V}, r)$
510
$\mathbb{R}.updateScoreTable(\mathbb{V})$
512
$\mathbb{R}.evict(\mathbb{V})$
514
$r = generate(r)$
$\mathbb{R}.insert(r)$
516

FIG. 5

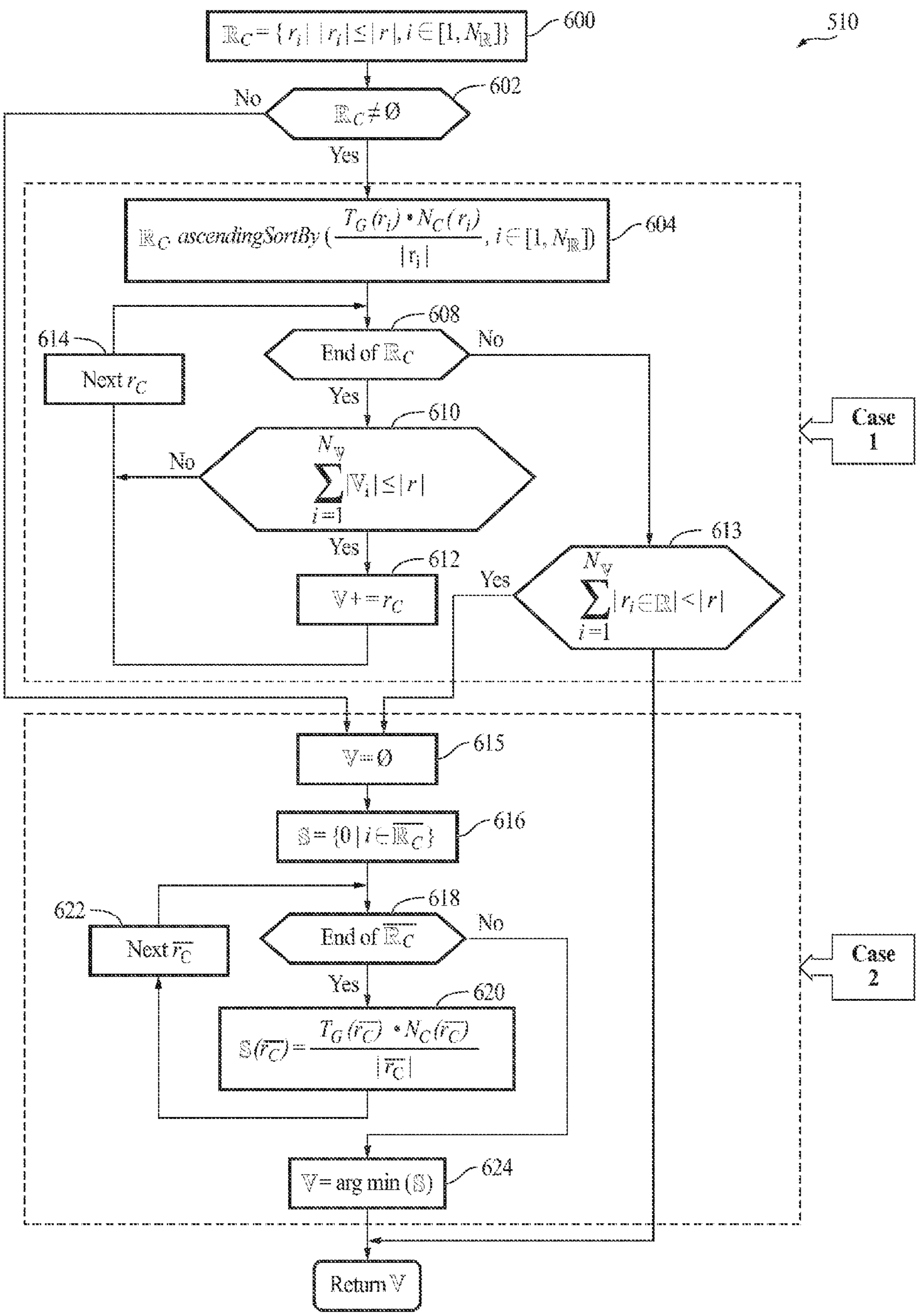

510
$\mathbb{R}_C = \{r_i \mid |r_i| \leq |r|, i \in [1, N_{\mathbb{R}}]\}$
600
$\mathbb{R}_C \neq \emptyset$
602
No
Yes
$\mathbb{R}_C$ ascendingSortBy $(\frac{T_G(r_i) \cdot N_C(r_i)}{|r_i|}, i \in [1, N_{\mathbb{R}}])$
604
608
End of $\mathbb{R}_C$
614
Next r_C
No
Yes
610
$\sum_{i=1}^{N_{\mathbb{V}}} |\mathbb{V}_i| \leq |r|$
No
Yes
612
$\mathbb{V} += r_C$
613
$\sum_{i=1}^{N_{\mathbb{V}}} |r_i \in \mathbb{R}| < |r|$
Yes
Case 1
615
$\mathbb{V} = \emptyset$
616
$\mathbb{S} = \{0 \mid i \in \overline{\mathbb{R}_C}\}$
618
End of $\overline{\mathbb{R}_C}$
622
Next $\overline{r_C}$
No
Yes
620
$\mathbb{S}(\overline{r_C}) = \frac{T_G(\overline{r_C}) \cdot N_C(\overline{r_C})}{|\overline{r_C}|}$
624
$\mathbb{V} = \arg\min(\mathbb{S})$
Case 2
Return $\mathbb{V}$

FIG. 6

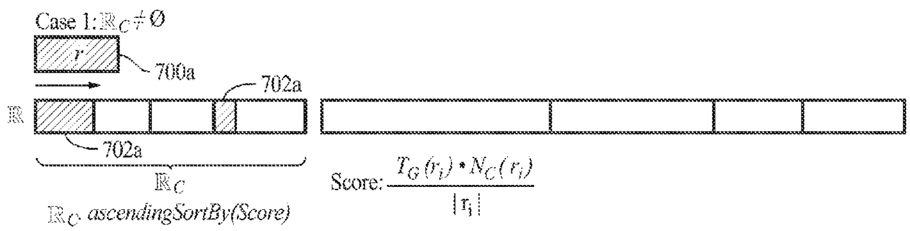


FIG. 7A

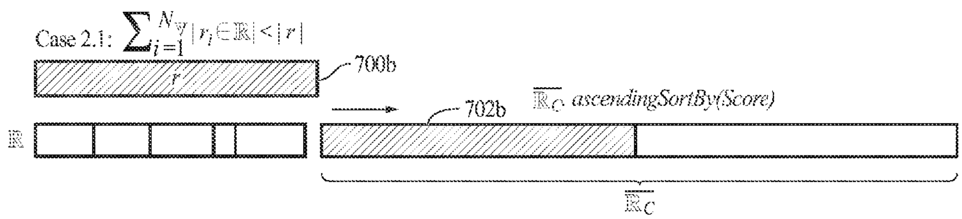


FIG. 7B

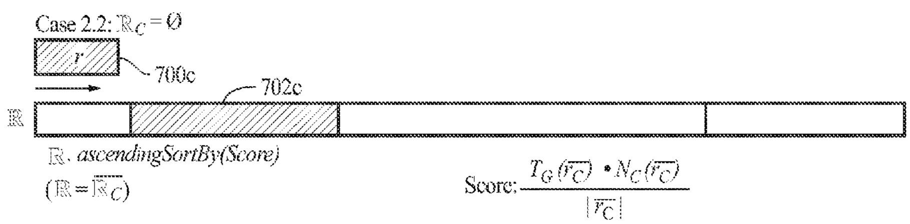


FIG. 7C

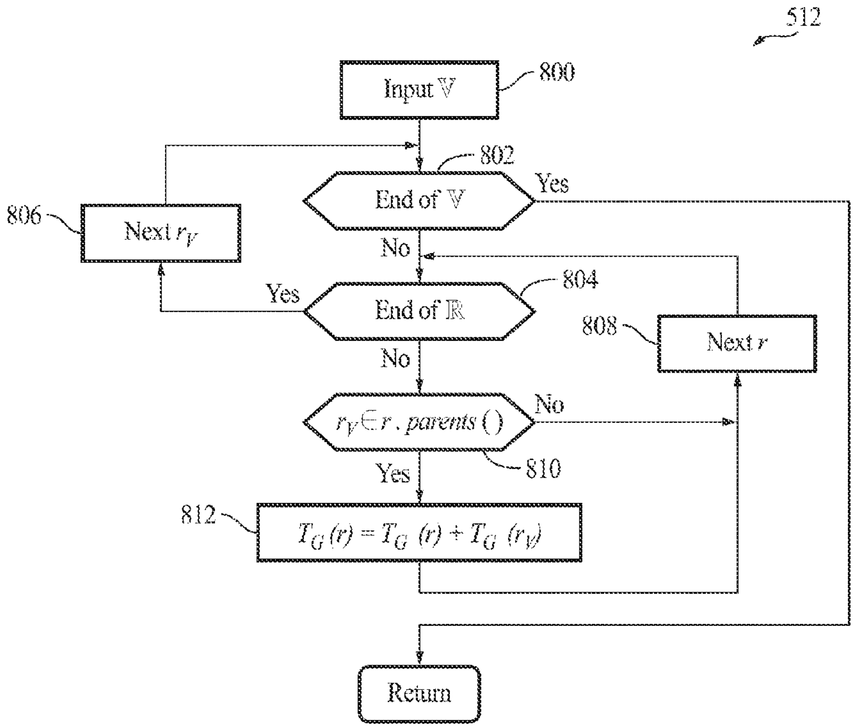

512
Input 𝕍
800
End of 𝕍
802
Yes
No
Next r_V
806
End of ℝ
804
Yes
No
Next r
808
$r_V \in r$. parents ()
810
No
Yes
$T_G(r) = T_G(r) + T_G(r_V)$
812
Return

FIG. 8

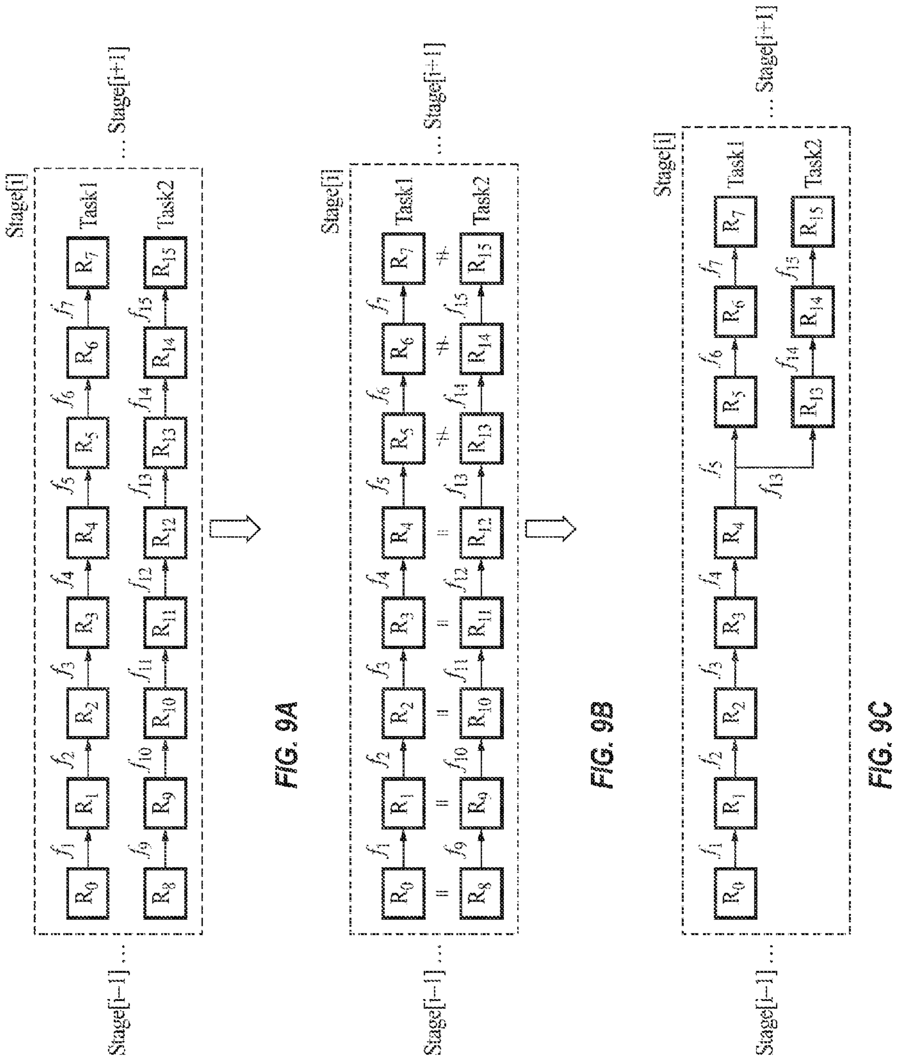
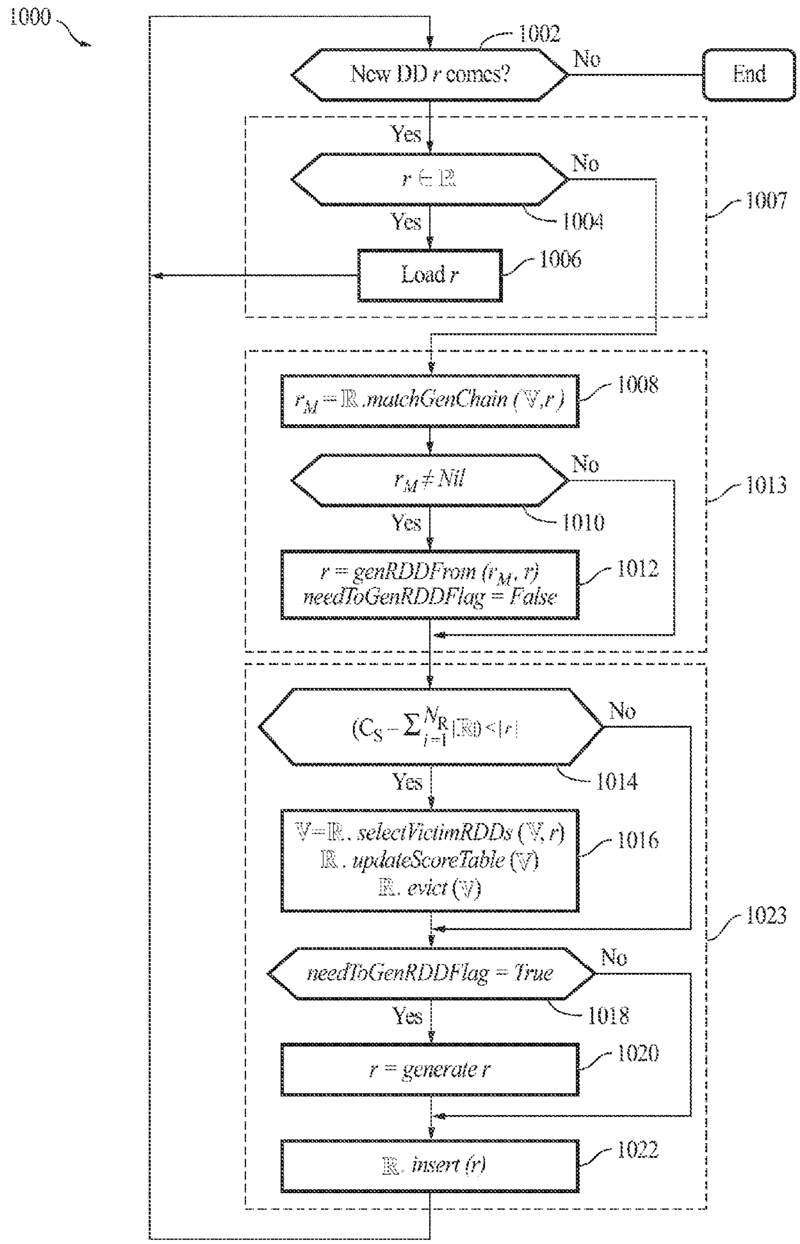


FIG. 9A

FIG. 9B

FIG. 9C



1000
1002
New DD r comes?
No
End
Yes
$r \in \mathbb{R}$
No
1007
1004
Yes
Load r
1006
1008
$r_M = \mathbb{R}.matchGenChain(\mathbb{V}, r)$
No
$r_M \neq Nil$
1013
1010
Yes
$r = genRDDFrom(r_M, r)$
$needToGenRDDFlag = False$
1012
$(C_S - \sum_{i=1}^{N_R} |\mathbb{R}|) < |r|$
No
1014
Yes
$\mathbb{V} = \mathbb{R}.selectVictimRDDs(\mathbb{V}, r)$
$\mathbb{R}.updateScoreTable(\mathbb{V})$
$\mathbb{R}.evict(\mathbb{V})$
1016
1023
$needToGenRDDFlag = True$
No
1018
Yes
$r = generate\ r$
1020
$\mathbb{R}.insert(r)$
1022

FIG. 10

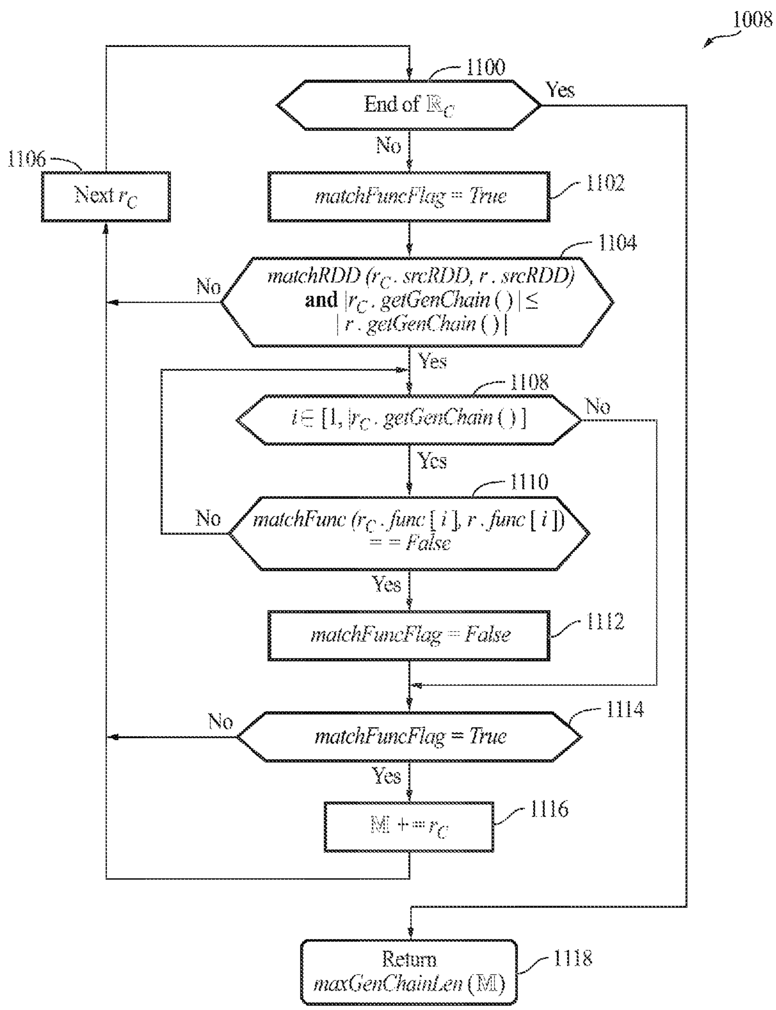

1008
1100
End of $\mathbb{R}_C$
Yes
No
1106
Next r_C
$matchFuncFlag = True$
1102
1104
No
$matchRDD(r_C.srcRDD, r.srcRDD)$ **and** $|r_C.getGenChain()| \leq |r.getGenChain()|$
Yes
1108
$i \in [1, |r_C.getGenChain()|]$
No
Yes
1110
No
$matchFunc(r_C.func[i], r.func[i]) == False$
Yes
$matchFuncFlag = False$
1112
No
$matchFuncFlag = True$
1114
Yes
$\mathbb{M} += r_C$
1116
Return $maxGenChainLen(\mathbb{M})$
1118

FIG. 11

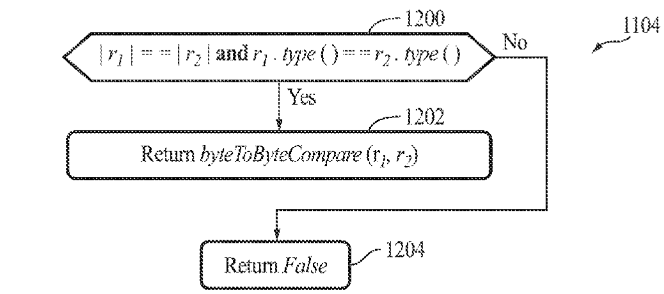


FIG. 12

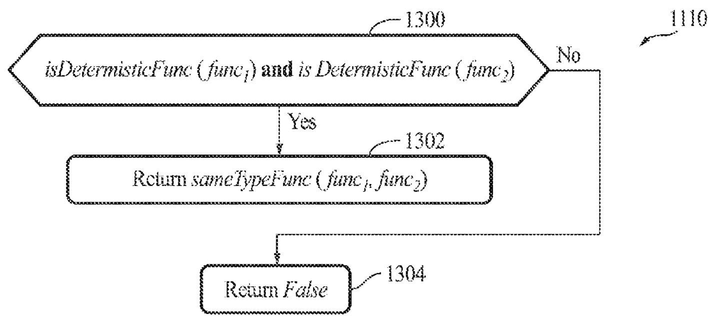


FIG. 13

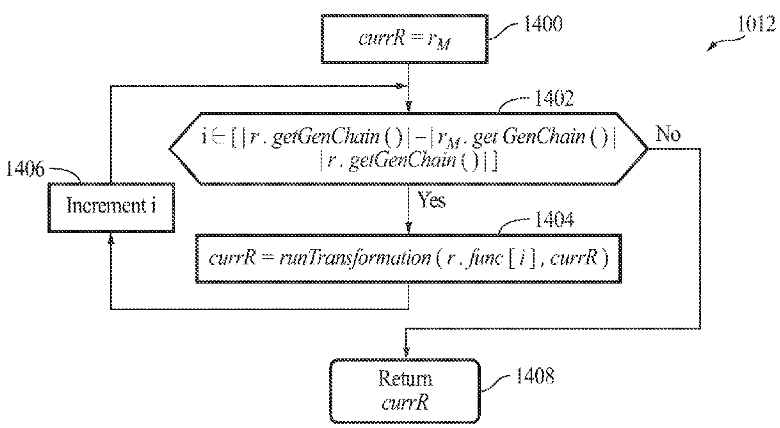


FIG. 14

IN-MEMORY SHARED DATA REUSE REPLACEMENT AND CACHING

CROSS-REFERENCE TO RELATED APPLICATION(S)

[0001] This application claims the benefit of and priority to U.S. Provisional Patent Application No. 62/384,078 filed Sep. 6, 2016, the entire content of which is incorporated herein by reference.

FIELD

[0002] Aspects of the present invention relate to the field of memory management in memory systems.

BACKGROUND

[0003] The Apache Spark framework is a generalized framework for distributed data processing providing functional APIs for manipulating data at scale, in-memory data caching, and reuse across computations. It applies a set of coarse-grained transformations over partitioned data and relies on a dataset's lineage to re-compute tasks in case of failures. Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, which is maintained in a fault-tolerant way. From a caching point of view, RDD represents distributed immutable data (partitioned data and iterator) and lazily evaluated operations (transformations). RDD has the following properties: different RDDs may have different sizes (in bytes); different RDDs' re-generating times may be different; a child RDD's re-generating time may be changed if its parent RDD(s) is (are) removed from the memory; and operations on RDD have different types, such as transformations, actions, and persistence. Spark improves the system performance by storing as many intermediate results (such as resilient distributed datasets (RDDs)) into the volatile memory instead of long-term data storage (such as a hard disk drive (HDD)) as possible.

[0004] However, this all-in-memory caching management scheme is limited by the finite size of memory, and cached RDDs may be evicted from the random access memory (RAM) by the Spark's default least recently used (LRU) policy, which may be not optimal for many use cases.

SUMMARY

[0005] Aspects of embodiments of the present invention are directed to a cache manager, which selects a victim intermediate data (e.g., a victim resilient distributed dataset (RDD)) with the consideration of both the importance degrees and sizes of cached intermediate data (e.g., RDDs). According to some embodiments, for each cache miss, the cache manager comprehensively makes eviction decisions based on a "score table" which monitors each intermediate data's (e.g., RDD's) re-generate time cost, children/dependence number, occupancy size, and/or the like. Thus, embodiments of the present invention may increase the dataset caching hit ratio in memory and improve the overall system performance.

[0006] According to some embodiments of the present invention, there is provided a cache management system for managing a plurality of intermediate data, the cache management system including: a processor; and a memory having stored thereon the plurality of intermediate data and instructions that when executed by the processor, cause the processor to perform: identifying a new intermediate data to be accessed; loading the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data; in response to not identifying the new intermediate data as one of the plurality of intermediate data: selecting a set of victim intermediate data from the plurality of intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data, the plurality of scores being based on a score table stored in the memory; evicting the set of victim intermediate data from the memory; updating the score table based on the set of victim intermediate data; and adding the new intermediate data to the plurality of intermediate data stored in the memory.

[0007] In some embodiments, the score table includes for each intermediate data of the plurality of intermediate data a number of children of the intermediate data, a size of the intermediate data, a re-compute time of the intermediate data, and a generating logic chain of the intermediate data.

[0008] In some embodiments, each one of the plurality of intermediate data is a resilient distributed data (RDD), and the cache management system operates under a Spark framework.

[0009] In some embodiments, the selecting of the set of victim intermediate data includes: identifying a first set of intermediate data of the plurality of intermediate data, each one of the first set of intermediate data having a size less than that of the new intermediate data.

[0010] In some embodiments, the selecting of the set of victim intermediate data includes: in response to the first set of intermediate data not being empty: sorting the first set of intermediate data in a sorted order based on first scores associated with respective ones of the first set of intermediate data; and adding ones of the first set of intermediate data to the set of victim intermediate data to be evicted, based on the sorted order, while a cumulative size of the set of victim intermediate data is less than the new intermediate data.

[0011] In some embodiments, the order is an ascending order.

[0012] In some embodiments, the selecting of the set of victim intermediate data further includes: generating each one of the first scores based on a re-compute time of, a number of children of, and a size of an associated one of the first set of intermediate data stored at the score table.

[0013] In some embodiments, the generating of each one of the first scores includes: calculating a product of a re-compute time and a number of children of an associated one of the first set of intermediate data; and dividing the calculated product by a size of the associated one of the first set of intermediate data to calculate the one of the first scores.

[0014] In some embodiments, the selecting of the set of victim intermediate data includes: in response to the first set of intermediate data having a size smaller than the new intermediate data: emptying the set of victim intermediate data; identifying a second set of intermediate data of the plurality of intermediate data, each one of the second set of intermediate data having a size greater than or equal to that of the new intermediate data; generating second scores associated with respective ones of the plurality of intermediate data based on the score table; and selecting one intermediate data, as the set of victim intermediate data, from a second set of intermediate data having a lowest associated score among the second scores.

[0015] In some embodiments, the generating of the second scores includes: generating each one of the second scores based on a re-compute time of, a number of children of, and a size of an associated one of the second set of intermediate data stored at the score table.

[0016] In some embodiments, the updating of the score table includes: for each one of the set of victim intermediate data: identifying a child intermediate data associated with the one of the set of victim intermediate data; calculating a re-compute time of the child intermediate data based on a re-compute time of the one of the set of victim intermediate data; and updating the score table based on the calculated re-compute time of the child intermediate data.

[0017] According to some embodiments of the present invention, there is provided a cache management system for managing a plurality of intermediate data, the cache management system including: a processor; and a memory having stored thereon a plurality of intermediate data and instructions that when executed by the processor, cause the processor to perform: identifying a new intermediate data to be accessed; loading the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data; in response to not identifying the new intermediate data as one of the plurality of intermediate data: generating a plurality of scores associated with respective ones of the plurality of intermediate data, each of the plurality of scores being based on a re-compute time of, a number of children of, and a size of an associated one of the plurality of intermediate data stored at a score table in the memory; selecting a set of victim intermediate data from the plurality of intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data; evicting the set of victim intermediate data from the memory; updating the score table based on the set of victim intermediate data; and adding the new intermediate data to the plurality of intermediate data stored in the memory.

[0018] According to some embodiments of the present invention, there is provided a method of managing a plurality of intermediate data stored in a memory, the method including: identifying, by a processor, a new intermediate data to be accessed; loading, by the processor, the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data; in response to not identifying the new intermediate data as one of the plurality of intermediate data: selecting, by the processor, a set of victim intermediate data from the plurality of intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data, the plurality of scores being based on a score table stored in the memory; evicting, by the processor, the set of victim intermediate data from the memory; updating, by the processor, the score table based on the set of victim intermediate data; and adding, by the processor, the new intermediate data to the plurality of intermediate data stored in the memory.

[0019] In some embodiments, the score table includes for each intermediate data of the plurality of intermediate data a number of children of the intermediate data, a size of the intermediate data, a re-compute time of the intermediate data, and a generating logic chain of the intermediate data.

[0020] In some embodiments, the selecting of the set of victim intermediate data includes: identifying a first set of intermediate data of the plurality of intermediate data, each one of the first set of intermediate data having a size less than that of the new intermediate data.

[0021] In some embodiments, the selecting of the set of victim intermediate data includes: in response to the first set of intermediate data not being empty: sorting the first set of intermediate data in a sorted order based on first scores associated with respective ones of the first set of intermediate data; and adding ones of the first set of intermediate data to the set of victim intermediate data to be evicted, based on the sorted order, while a cumulative size of the set of victim intermediate data is less than or equal to the new intermediate data, wherein the order is an ascending order.

[0022] In some embodiments, the selecting of the set of victim intermediate data further includes: generating each one of the first scores based on a re-compute time of, a number of children of, and a size of an associated one of the first set of intermediate data stored at the score table.

[0023] In some embodiments, the selecting of the set of victim intermediate data includes: in response to the first set of intermediate data having a size smaller than the new intermediate data: emptying the set of victim intermediate data; identifying a second set of intermediate data of the plurality of intermediate data, each one of the second set of intermediate data having a size greater than or equal to that of the new intermediate data; generating second scores associated with respective ones of the plurality of intermediate data based on the score table; and selecting one intermediate data, as the set of victim intermediate data, from a second set of intermediate data having a lowest associated score among the second scores.

[0024] In some embodiments, the generating of the second scores includes: generating each one of the second scores based on a re-compute time of, a number of children of, and a size of an associated one of the second set of intermediate data stored at the score table.

[0025] In some embodiments, the updating of the score table includes: for each one of the set of victim intermediate data: identifying a child intermediate data associated with the one of the set of victim intermediate data; calculating a re-compute time of the child intermediate data based on a re-compute time of the one of the set of victim intermediate data; and updating the score table based on the calculated re-compute time of the child intermediate data.

BRIEF DESCRIPTION OF THE DRAWINGS

[0026] The accompanying drawings, together with the specification, illustrate example embodiments of the present invention, and, together with the description, serve to explain the principles of the present invention.

[0027] FIG. **1** illustrates a cache management system utilizing a cache manager, according to some example embodiments of the present invention.

[0028] FIG. **2** illustrates a Spark architecture that utilizes the cache manager, according to some example embodiments of the present invention.

[0029] FIG. **3** illustrates an example of memory partitions under the Spark framework.

[0030] FIGS. **4**A and **4**B illustrate an example of a resilient distributed dataset (RDD) lookup procedure, according to some example embodiments of the present invention.

[0031] FIG. **5** is a flow diagram illustrating a process of performing an RDD caching policy, according to some example embodiments of the present invention.

[0032] FIG. **6** illustrates a process of victim selection performed by the cache manager, according to some example embodiments of the present invention.

[0033] FIGS. **7**A-**7**C illustrate various cases provided in FIG. **6**, according to some example embodiments of the present invention.

[0034] FIG. **8** illustrates a process of updating an RDD score table as performed by the cache manager, according to some example embodiments of the present invention.

[0035] FIGS. **9**A-**9**C illustrate the process of deduplication performed by the cache manager, according to some example embodiments of the present invention.

[0036] FIG. **10** is a flow diagram illustrating a process of performing an RDD caching policy, according to some example embodiments of the present invention.

[0037] FIG. **11** is a flow diagram illustrating a process of matching generation chains performed by the cache manager, according to some example embodiments of the present invention.

[0038] FIG. **12** is a flow diagram illustrating a process of matching RDDs performed by the cache manager, according to some example embodiments of the present invention.

[0039] FIG. **13** is a flow diagram illustrating a process of matching functions performed by the cache manager, according to some example embodiments of the present invention.

[0040] FIG. **14** is a flow diagram illustrating a process of obtaining the longest reusable chain as performed by the cache manager, according to some example embodiments of the present invention.

DETAILED DESCRIPTION

[0041] Features of the inventive concept and methods of accomplishing the same may be understood more readily by reference to the following detailed description of embodiments and the accompanying drawings. Hereinafter, example embodiments will be described in more detail with reference to the accompanying drawings, in which like reference numbers refer to like elements throughout. The present invention, however, may be embodied in various different forms, and should not be construed as being limited to only the illustrated embodiments herein. Rather, these embodiments are provided as examples so that this disclosure will be thorough and complete, and will fully convey the aspects and features of the present invention to those skilled in the art. Accordingly, processes, elements, and techniques that are not necessary to those having ordinary skill in the art for a complete understanding of the aspects and features of the present invention may not be described. Unless otherwise noted, like reference numerals denote like elements throughout the attached drawings and the written description, and thus, descriptions thereof will not be repeated. In the drawings, the relative sizes of elements, layers, and regions may be exaggerated for clarity.

[0042] As will be understood by a person of ordinary skill in the art, while some embodiments of the present invention may be described in the context of the Apache Spark system, embodiments of the present invention are not limited thereto. Rather, embodiments of the present invention, and the term "Spark" as used herein, may be applied to any suitable cluster compute engine for scalable data processing, having the same or similar characteristics described herein.

[0043] FIG. **1** illustrates a cache management system **100** utilizing the cache manager **110**, according to some example embodiments of the present invention.

[0044] The processing system **100** may include a processor **102** and a memory **104**. The memory **104** may include volatile memory (e.g., random access memory (RAM), such as dynamic RAM) and non-volatile memory (e.g., a hard disk drive (HDD) or a solid state drive (SSD)), which may serve as long-term storage. The cache manager **110**, according to some embodiments, may reside at the memory **104** and its operations may be performed by the processor **102**.

[0045] According to some embodiments, the cache manager **110** may operate within the Spark framework. Spark is a generalized framework for distributed data processing providing functional application programming interface (API) for manipulating data at scale, in-memory data caching and reuse across computations. Spark applies a set of coarse-grained transformations over partitioned data and relies on a dataset's lineage to re-compute tasks in case of failures. Spark supports a majority of data formats, has integrations with various storage systems, and may be executed on a cluster manager, such as Apache Mesos or Apache Hadoop YARN (Yet Another Resource Negotiator).

[0046] FIG. **2** illustrates an architecture (e.g., a Spark architecture) **200** that utilizes the cache manager **110** according to some example embodiments of the present invention.

[0047] Referring to FIG. **2**, the spark architecture **200** includes an application layer (e.g., a Spark Application Layer) **202**, a cluster manager layer **204**, and a worker layer **206**.

[0048] The spark application layer includes user program **208** and spark context **210** and creates RDDs and performing a series of transformations to achieve a final result. The RDD may represent a minimum unit for caching. These transformations of RDDs are then translated into a directed Acyclic graph (DAG) structure (also referred to as an RDD architecture, an RDD graph, or a generating logic chain) and submitted to the DAG scheduler **214** to be executed on a set of worker nodes **220**. The spark context **210** represents the connection to a spark cluster, and may be used to create RDDs, accumulators, and broadcast variables on that cluster. For example, the spark context **210** includes RDD graph **212**, DAG scheduler **214**, task scheduler **216**, and scheduler backend modules **218**.

[0049] The cluster manager layer **204** may run different implementations (e.g., spark standalone **220**, YARN **222**, Mesos **224**). According to some embodiments, the cache manager (e.g., of the resilient distributed dataset, RDD) **110** resides at (e.g., is configured to be incorporated or plugged into) the cluster manager layer **204**.

[0050] The worker Layer **206** includes multiple work nodes **226**, each of which is a java virtual machine (JVM) running multiple executors **228**. The executors **228** run tasks scheduled by a driver, store computation results in memory, and then conduct on-disk or off-heap interactions with storage systems. The executors **228** may run as Java processes, so that the available memory is equal to the heap size. Internally available memory is split into several regions with specific functions.

[0051] FIG. **3** illustrates an example of memory partitions under the Spark framework.

[0052] Referring to FIG. **3**, the volatile portion **300** of memory **104** may be partitioned into reserved memory **302**, user memory **304**, and the unified pool **306**. The reserved memory (C_R) **302** is reserved by the system and has a hardcoded size (e.g., 300 MB in Spark version 1.6). This partition is for running executor **228** and may not be strictly related to Spark. The user memory (C_U) **304** may store user data structures, which may be used in RDD transformations, and internal metadata used by Spark. For example, users may rewrite Spark aggregation by using the mapPartitions transformation maintaining hash table for this aggregation to run, which would consume user memory **304**. The unified pool may be split into two non-static sub-partitions: storage memory **306***a* and executing memory **306***b*, and the boundary therebetween may be set by the spark.memory.storageFraction parameter (defaults to 50%). The storage memory (C_S) **306***a* is used for storing Spark cached data and for temporary space serialized data unroll. Also, all the broadcast variables are stored there as cached blocks. The cache manager **110**, according to some embodiments, is focused on this partition, as user "preserved" RDDs are stored in the storage memory **306***a*. The executing memory (C_E) **306***b* is used for storing the objects required during the execution of Spark tasks. The size of said memory partitions may be related according to the following two Equations:

$$C = C_R + C_U + C_E + C_S \tag{1}$$

$$C_S = (C - C_R - C_U) \times \epsilon \tag{2}$$

[0053] where C represents total memory size, and ϵ (i.e., spark.memory.fraction) adjusts the sizes of storage and executing memory **306***a* and **306***b*. The cache manager **110** is incorporated into the cluster manager layer **204** and operates on the RDDs stored ("preserved") by the user in the storage memory **306***a*. Embodiments of the present invention primarily focus on the scenario where the amount of total (or accumulated) preserved RDDs requested by the user is larger than the total available space of storage memory (C_S), and the cache manager **110**, rather than using the related art least recently used (LRU) eviction policy, intelligently and efficiently evicts RDD victims before inserting a new RDD.

[0054] According to embodiments of the present invention, the cache manager **110** performs a caching replacement process, whereby victim datasets (e.g., victim RDD(s)) are selected with the consideration of both the importance and sizes of these cached RDDs. For example, for each cache miss, the cache manager **110** may comprehensively make decisions based on a "score table" which monitors each RDD's re-generate time cost, children/dependence number, occupancy size, and/or the like. Embodiments of the present invention may be configured to increase the caching hit ratio in memory **104** and improve the overall system performance (e.g., performance of Spark framework) by reusing intermediate data (e.g., RDDs) instead of re-computing them. According to some embodiments of the present invention, intermediate data is reused by optimizing the cache that stores user preserved intermediate data sets (e.g., user preserved RDDs).

[0055] FIGS. **4**A and **4**B illustrate an example of an RDD lookup procedure, according to some example embodiments of the present invention. FIG. **4**A illustrates an example of a DAG structure (e.g., an RDD graph or generating logic chain) **400** of an example RDD transformation, where there are 11 RDDs (intermediate data) **410** and 10 transformations **420** in stage[i]. FIG. **4**B illustrates an example score table **430** generated and maintained by the cache manager **110**, according to some embodiments of the present invention, to track certain information about each accessed RDD. According to some embodiments, the cache manager **110** records, in the score table **430**, spatial (i.e., RDD size in bytes) and temporal (i.e., re-computation) costs, as well as generating operation chain (i.e., GenChain, source RDD, child#) of each RDD. Based on these tabulated records, the cache manager **110** can calculate the score (e.g., "importance degree") of each RDD when making decisions on victim selection and eviction. In some embodiments, the score table contains more parameters than those shown in Table 1, such as time costs of prefetching from memory/SSD/HDD if a multi-tier storage system is utilized. In some embodiments, the score table is stored in the storage memory **306***a*; however, embodiments of the present invention are not limited thereto, and the score table may be at least partially (e.g., fully) stored in non-volatile memory (e.g., NVMe disk, SSD or HDD).

[0056] FIG. **5** is a flow diagram illustrating a process **500** of the cache manager **110** for performing an RDD caching policy, according to some example embodiments of the present invention.

[0057] According to some examples, for every new RDD to be accessed (act **502**), when the new RDD r is found in the cache $\mathbb{R}$ in act **504** (i.e., there is cache hit), the cache manager **110** loads the cached RDD r in act **506**. Otherwise, if the RDD r is not found in the cache $\mathbb{R}$ in act **504**, the cache manager **110** first checks, in act **508**, whether the cache (i.e., storage memory (C_S) **306***a*) has sufficient capacity to store the new RDD. If there is sufficient space for the RDD, the cache manager **110** proceeds to act **516**. If there is insufficient space for the RDD, in act **510**, the cache manager **110** selects victim RDD(s) to evict (e.g., delete) from the cache (the selectVictimRDDs function); in act **512**, updates the score table (the updateScoreTable function) accordingly; and, in act **514**, evicts the victim RDD(s). Thereafter, in act **516**, the cache manager **110** generates (e.g., computes) the new RDD r and inserts it into the memory (e.g., storage memory (C_S) **306***a*). The process **500** is also shown as the pseudo code below:

Process 500

```
Procedure RDDCache (r)
    for each new RDD r do
        if r ∈ℝ then
            load r
        else
            if (C_S − Σ_{i=1}^N ℝ |ℝ |) < |r| then
                𝕍 = Ø
                𝕍 = ℝ .selectVictimRDDs( 𝕍 ,r)
                updateScoreTable(ℝ , 𝕍 )
                ℝ .evict( 𝕍 )
            r = generate(r)
            ℝ .insert(r)
    Return
```

[0058] where $\mathbb{R}$ represents the set of RDDs stored in the storage memory **306***a* (e.g., RAM), $\mathbb{V}$ represents the set of victim RDDs to be evicted from the storage memory **306***a*, and $N_{\mathbb{R}}$ represents the total number of RDDs in the storage memory **306***a*, C_S represents the size (e.g., in bytes) of the storage memory **306***a*, $\sum_{i=1}^{N_{\mathbb{R}}} \mathbb{R}$ represents the sum total of the size of the set of RDDs stored in the storage memory **306***a*, and $|r|$ represents the size of the new RDD.

[0059] According to some embodiments, the RDD victim selection process (i.e., the selectVictimRDDs function) is converted to a multi-objective optimization process, as expressed in equations 3 to 6 below:

$$\text{Minimize} \left(\left(\sum_{i=1}^{N_{\mathbb{V}}} |\mathbb{V}_i| \right) - |r|, \sum_{i=1}^{N_{\mathbb{V}}} S(\mathbb{V}_i) \right) \tag{3}$$

$$\text{Subject to: } |r| \le C_S \tag{4}$$

$$\mathbb{V} \subseteq \mathbb{R} \tag{5}$$

$$\left(\sum_{i=1}^{N_{\mathbb{V}}} |\mathbb{V}_i| \right) - |r| \ge 0 \tag{6}$$

[0060] The first objective function in Equation 3 aims to reduce (e.g., minimize) the size of the evicted RDD(s) $\mathbb{V}$ ($\sum_{i=1}^{N_{\mathbb{V}}} |\mathbb{V}_i|$, where $N_{\mathbb{V}}$ represents the number of victim RDD(s)) based on the size of the new RDD ($|r|$). The aim of the second objective function in Equation 3 is to minimize the total of the scores of evicted RDDs ($\sum_{i=1}^{N_{\mathbb{V}}} S(\mathbb{V})$). The score of an RDD may represent the importance of the RDD and may be determined based on the score table **430**, which tracks each accessed RDD's generating logic chain (transformation functions and dependent RDDs), children number, RDD size in bytes, and generating time cost (re-computing time). In some embodiments of the present invention, the cache manager **110** evicts as few stored RDDs as possible, and selects for eviction those RDD with as low a score as possible.

[0061] Equation 4 expresses the assumption upon which the process **500** is based, which is that the new RDD (r) should not be larger than the entire available memory size (i.e., C_S, the entire size of the memory storage **306***a*). Equation 5 expresses that the set of victim RDD(s) ($\mathbb{V}$) is a subset of the entire RDD set ($\mathbb{R}$) stored in the RAM. Equation 6 expresses the aim that the total space of victims should cover the new RDD's working set space.

[0062] According to embodiments of the present invention, when selecting victim RDDs, the cache manager **110** aims to achieve the objective of Equation 3 as much as practicable, while also satisfying the conditions of Equations 4 to 6. Example embodiments of the present invention operate to set different importance score functions and victim candidate sets for different cases.

[0063] FIG. **6** illustrates the process **510** of victim selection (i.e., the selectVictimRDDs function) performed by the cache manager **110**, according to some example embodiments of the present invention. The process **510** is also shown in the pseudo code below:

Procedure 510

Procedure selectVictimRDDs ($\mathbb{V}$, r)
$\mathbb{R}_C = \{r_i \mid |r_i| \le |r|, i \in [1, N_{\mathbb{R}}]\}$
if $\mathbb{R}_C \ne \emptyset$ do
$\mathbb{R}_C \cdot \text{ascendingSortBy}\left(\frac{T_G(r_i) \cdot N_C(r_i)}{|r_i|}, i \in [1, N_{\mathbb{R}}]\right)$
for r_C in $\mathbb{R}_C$ do
if $\sum_{i=1}^{N_{\mathbb{V}}} |\mathbb{V}_i| \le |r|$ then
$\mathbb{V} \mathrel{+}= r_C$
else if $\mathbb{R}_C = \emptyset$ or $\sum_{i=1}^{N_{\mathbb{V}}} |r_i$ in $\mathbb{R}_C| < |r|$ then
$\mathbb{V} = \emptyset$
$\mathbb{S} = \{0 \mid i \in \overline{\mathbb{R}_C}\}$
for $\overline{r_C} \in \overline{\mathbb{R}_C}$ do
$\mathbb{S}(\overline{r_C}) = \frac{T_G(\overline{r_C}) \cdot N_C(\overline{r_C})}{|\overline{r_C}|}$
$\mathbb{V} = \arg\min (\mathbb{S})$
return $\mathbb{V}$

[0064] where $\mathbb{R}_C$ represents the set of cached (stored) RDDs (a first set of intermediate data) whose sizes are less than or equal to the new RDD (r), and $\mathbb{S}$ represents the record of scores of all RDDs stored in the memory (e.g., storage memory **306***a*).

[0065] In act **600**, a subset of cached RDDs ($\mathbb{R}_C$) to record all RDDs whose sizes are smaller than the new RDD (r) is maintained. In act **602**, based on whether $\mathbb{R}_C$ is an empty set or not, the problem is categorized into two cases:

[0066] Case **1**: If $\mathbb{R}_C$ is not an empty set, in act **604**, the cache manager **110** sets those cached RDDs whose sizes are smaller than r as victim candidates. To select one or more victims from $\mathbb{R}_C$, the cache manager **110** defines a first score (e.g., an importance degree score) as:

$$\text{Score-}v1\text{: } \frac{T_G(i) \cdot N_C(i)}{|R_i|}, \tag{7}$$

[0067] where $T_G(i)$, $N_C(i)$ and $|R_i|$ represent the re-computing (generation) time cost, the number of children RDDs, and the size (in bytes) of RDD i, respectively. As expressed in Equation 7, the cache manager **110** may assign a higher score (a higher importance degree) to an RDD that has a higher re-computing time $T_G(i)$, a greater number of children, and/or a smaller size, as the eviction of such RDDs may have a higher impact on the remaining RDDs and/or consume more system resources to recreate.

[0068] In act **608**, the cache manager **110** sorts the victim candidates based on (e.g., in ascending or descending order of) the first score (Score-v1). In acts **608-614**, the cache manager **110** adds the sorted RDDs to the victim RDDs subset $\mathbb{V}$ until the size of $\mathbb{V}$ is equal to or greater than the new RDD r.

[0069] Case **2**: This may be presented as two sub-cases: in the first sub-case, $\mathbb{R}_C$ is not empty, but even if all RDDs in $\mathbb{R}_C$ are selected as victim RDDs, there is still not enough space for the new RDD r (i.e., $\sum_{i=1}^{N_{\mathbb{V}}} |r_i$ in $\mathbb{R}_C| < |r|$; act **613**). In the second sub-case, $\mathbb{R}_C$ is empty, indicating that the new

RDD r is smaller than all cached RDDs, in which case the cache manager **110** selects just one RDD to evict.

[0070] In either sub-case, the cache manager **110** empties, in act **615**, the victim RDD set $\mathbb{V}$ and selects and adds, in acts **616-622**, only one RDD from $\overline{\mathbb{R}_C}$ ($\overline{\mathbb{R}_C} = \mathbb{R} - \mathbb{R}_C$), the complementary set of $\mathbb{R}_C$ (the second set of intermediate data), to $\mathbb{V}$. In the second sub-case, as $\mathbb{R}_C$ is empty, $\overline{\mathbb{R}_C}$ =R. In either sub-case, the cache manager **110** may not simply select the smallest RDD in $\overline{\mathbb{R}_C}$, as it may have a high importance degree. Therefore, in some embodiments, the cache manager **110** computes the scores (second scores) of the RDD in $\overline{\mathbb{R}_C}$ and selects the RDD with the least score (e.g., the least importance) as the victim $\mathbb{V}$ (act **624**). In some embodiments, the cache manager **110** defines the score of each RDD in $\overline{\mathbb{R}_C}$ using Equation 7.

[0071] As will be understood by a person of ordinary skill in the art, the first and second sub-cases may be collapsed into the second sub-case as an empty $\mathbb{R}_C$ has a size that is less than that of the new RDD r, however, these two sub-cases are separately discussed here for clarity of illustration.

[0072] FIG. **7**A illustrates case **1** described above, in which $\mathbb{R}_C$ is not empty. FIGS. **7**B and **7**C respectively illustrate the first sub-case and the second sub-case of Case **2** described above. In FIGS. **7**A to **7**C, cell **700***a/b/c* indicates the new RDD r and cell(s) **702***a/b/c* indicates the victim RDD(s) selected for eviction.

[0073] According to some embodiments, after the victim RDD set $\mathbb{V}$ is determined, and before the constituent RDD (s) is/are evicted from the storage memory **306***a* (e.g., the cache $\mathbb{R}$, the cache manager **110** updates the score table. The reason for this is that once an RDD is evicted, the re-computing time cost of all other RDDs that are related to said RDD (i.e., "children" of said RDD) will be changed. For example, if $RDD_3=f_3(RDD_2)$, and if the parent, RDD_2, is cached in the memory, then R_3's time cost is T[R2→R3]. However, if RDD_2 is selected as a victim RDD to be removed, the cache manager **110** calculates RDD_3 from RDD_1 if $RDD_2=f_2(RDD_1)$, and thus, the cache manager **110** updates the time cost of RDD_3 as T[R1→R2]+T[R2→R3].

[0074] FIG. **8** illustrates the process **512** of updating the score table **430** (i.e., the updateScoreTable function) performed by the cache manager **110**, according to some example embodiments of the present invention. The process **512** is also shown in the pseudo code below:

Process 512
Note: parent( ) : Get the set of a certain RDD's parents.

```
Procedure updateScoreTable (ℝ, 𝕍)
    for r_V in 𝕍 do
        for r in ℝ do
            if r_V ∈ r.parents( ) then
                T_G(r) = T_G(r) + T_G(r_V)
    return
```

[0075] In acts **800-812**, for each selected victim RDD r_V in the victim subset $\mathbb{V}$, the cache manager **110** updates re-computing times of those existing RDDs in $\mathbb{R}$ that depend on said r_V. In act **810**, the cache manager **110** identifies a parent (e.g., a direct parent) of the victim RDD r_V, and in act **812**, increments the re-compute time of the parent RDD by the re-compute time of the victim RDD r_V, and updates the score table **430** accordingly. In some embodiments, the cache manager **110** identifies a parent (e.g., direct parent) of an RDD based on the generating logic chain recorded in the score table **430** (see, e.g., FIG. **4**B). In other words, the cache manager **110** identifies a child of each of the selected victim RDD r_V in the victim subset $\mathbb{V}$, and re-computes the generation (re-compute) time based on the generation time of the respective victim RDD r_V.

[0076] According to embodiments of the present invention, the cache manager **110** performs deduplication of intermediate data (e.g., RDDs) by detecting and reducing duplicated intermediate data (e.g., duplicate RDDs) by tracking and comparing their content and the generating logic chain (i.e., operation path). Instead of solely following the user declared topologic path (e.g., hardcoded "preserved" RDDs), the cache manager **110** further actively refines the intermediate data graph (a directed multigraph with properties attached to each vertex and edge, e.g., an RDD graph), in order to improve performance by saving both temporal (e.g., re-computation time) and spatial (e.g., memory space) costs. According to some embodiments of the present invention, the cache manager **110** is configured to be plugged into any caching subsystem that has knowledge of the generating chain of each dataset (even by disassembling operations or more advanced reverse engineering techniques). Additionally, the cache manager **110** according to some example embodiments of the present invention is configured to be integrated with (e.g., adopted into) existing techniques (e.g., the techniques described in "Analyzing Compute vs. Storage Tradeoff for Video-aware Storage Efficiency" by Kathpal et al., HotStorage (2012); and "The Constrained Ski-Rental Problem and its Application to Online Cloud Cost Optimization" by Khanafer et al., INFOCOM, 2013 Proceedings IEEE (2013), the entire contents of which are incorporated herein by reference). Thus, the cache manager **110** is configured to merge duplicated datasets and increase the hit ratio in memory. The hit ratio, which is a measurement of cache memory performance, may be defined as the ratio of the number of times data requested by CPU are found in the cache (number of "hits") to the number of total CPU data requests (which includes requests for data that are not found in the cache memory, i.e., "misses").

[0077] According to some example embodiments of the present invention, the cache manager **110** is configured to detect duplicated RDDs and deterministic RDD operations, to reduce or avoid re-computation and duplicated RDD storage. While related-art deduplication algorithms may primarily focus on the dataset content itself, embodiments of the present invention also consider the generating chain of each dataset. Therefore, utilizing a "two-dimensional comparison" (i.e., RDDs and operations) according to embodiments of the present invention efficiently improves the dataset (e.g., RDD) reusability.

[0078] According to some embodiments of the present invention, the cache manager **110** differentiates between deterministic and non-deterministic operations and automatically tracks changes of the stored RDDs and their generating chains (e.g., time cost of re-computing if a cached RDD's parent RDD is evicted).

[0079] FIGS. **9**A-**9**C illustrate the process of deduplication performed by the cache manager **110**, according to some example embodiments of the present invention.

[0080] In the example shown in FIG. **9**A, the RDD DAG structure includes two tasks and each task has 8 RDDs in stage[i]. The illustrated RDD graph may be based on what the user (e.g., the human programmer) hardcoded. FIGS. **9**B-**9**C illustrate the process of comparing the RDDs and operations between RDD tasks according to some example embodiments. As shown in FIG. **9**B, the cache manager **110** first compares the RDDs and the operations of stage [i] from the source (i.e., left hand side) to the end (i.e., right hand side). For example, the cache manager **110** identifies RDDs R_0 to R_4 as being respectively identical to RDDs R_8 to R_{12}, and the functions f_1 to f_4 as being respectively identical to functions f_9 to f_{12}. FIG. **9**C illustrates the refined RDD graph after detection and deduplication (e.g., reuse). Instead of 16 RDDs, only 11 RDDs need to be stored in the memory without any loss in terms of cache hit ratio.

[0081] Table 1 and Table 2 illustrate the score tables respectively before and after the detection and reuse process.

TABLE 1

<table>
<tr><th>RDD</th><th>GenChain</th><th>SrcRDD</th><th>Child# (N_c)</th><th>Size (|R|)</th><th>GenTime (TG)</th></tr>
<tr><td>R0</td><td>—</td><td>R0</td><td>1</td><td>|R0|</td><td>0</td></tr>
<tr><td>R1</td><td>f1</td><td>R0</td><td>1</td><td>|R1|</td><td>T[0→1]</td></tr>
<tr><td>R2</td><td>f2-f1</td><td>R0</td><td>1</td><td>|R2|</td><td>T[1→2]</td></tr>
<tr><td>R3</td><td>f3-f2-f1</td><td>R0</td><td>1</td><td>|R3|</td><td>T[2→3]</td></tr>
<tr><td>R4</td><td>f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R4|</td><td>T[3→4]</td></tr>
<tr><td>R5</td><td>f5-f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R5|</td><td>T[4→5]</td></tr>
<tr><td>R6</td><td>f6-f5-f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R6|</td><td>T[5→6]</td></tr>
<tr><td>R7</td><td>f7-f6-f5-f4-f3-f2-f1</td><td>R0</td><td>0</td><td>|R7|</td><td>T[6→7]</td></tr>
<tr><td>R8</td><td>—</td><td>R8</td><td>1</td><td>|R5|</td><td>0</td></tr>
<tr><td>R9</td><td>f9</td><td>R8</td><td>1</td><td>|R9|</td><td>T[8→9]</td></tr>
<tr><td>R10</td><td>f10-f9</td><td>R8</td><td>1</td><td>|R10|</td><td>T[9→10]</td></tr>
<tr><td>R11</td><td>f11-f10-f9</td><td>R8</td><td>1</td><td>|R11|</td><td>T[10→11]</td></tr>
<tr><td>R12</td><td>f12-f11-f10-f9</td><td>R8</td><td>1</td><td>|R12|</td><td>T[11→12]</td></tr>
<tr><td>R13</td><td>f13-f12-f11-f10-f9</td><td>R8</td><td>1</td><td>|R13|</td><td>T[12→13]</td></tr>
<tr><td>R14</td><td>f14-f13-f12-f11-f10-f9</td><td>R8</td><td>1</td><td>|R14|</td><td>T[13→14]</td></tr>
<tr><td>R15</td><td>f15-f14-f13-f12-f11-f10-f9</td><td>R8</td><td>0</td><td>|R15|</td><td>T[14→15]</td></tr>
</table>

TABLE 2

<table>
<tr><th>RDD</th><th>GenChain</th><th>SrcRDD</th><th>Child# (N_c)</th><th>Size (|R|)</th><th>GenTime (TG)</th></tr>
<tr><td>R0</td><td>—</td><td>R0</td><td>1</td><td>|R0|</td><td>0</td></tr>
<tr><td>R1</td><td>f1</td><td>R0</td><td>1</td><td>|R1|</td><td>T[0→1]</td></tr>
<tr><td>R2</td><td>f2-f1</td><td>R0</td><td>1</td><td>|R2|</td><td>T[1→2]</td></tr>
<tr><td>R3</td><td>f3-f2-f1</td><td>R0</td><td>1</td><td>|R3|</td><td>T[2→3]</td></tr>
<tr><td>R4</td><td>f4-f3-f2-f1</td><td>R0</td><td>2</td><td>|R4|</td><td>T[3→4]</td></tr>
<tr><td>R5</td><td>f5-f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R5|</td><td>T[4→5]</td></tr>
<tr><td>R6</td><td>f6-f5-f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R6|</td><td>T[5→6]</td></tr>
<tr><td>R7</td><td>f7-f6-f5-f4-f3-f2-f1</td><td>R0</td><td>0</td><td>|R7|</td><td>T[6→7]</td></tr>
<tr><td>R13</td><td>f13-f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R13|</td><td>T[4→13]</td></tr>
<tr><td>R14</td><td>f14-f13-f4-f3-f2-f1</td><td>R0</td><td>1</td><td>|R14|</td><td>T[13→14]</td></tr>
<tr><td>R15</td><td>f15-f14-f13-f4-f3-f2-f1</td><td>R0</td><td>0</td><td>|R15|</td><td>T[14→15]</td></tr>
</table>

[0082] FIG. **10** is a flow diagram illustrating a process **1000** of the cache manager **110** for performing an RDD caching policy, according to some example embodiments of the present invention.

[0083] Process **100** illustrates an example process of the RDD caching policy inside the cluster manager subsystem **204** within which the cache manager **110** is plugged in, according to some example embodiments. In some examples, for every new RDD to be accessed (act **1002**), when the new RDD r is found in the cache $\mathbb{R}$ in act **1004** (i.e., there is cache hit), the cache manager **110** loads the cached RDD r in act **1006**. Otherwise, if the RDD r is not found in the cache $\mathbb{R}$ in act **1004**, the cache manager **110** finds any matched duplicate RDDs r_M sharing all or a part of the new RDD's generation chain (GenChain in Table 1) in act **1008** (the matchGenChain function). In act **1010**, if the search returns a non-nil value, r_M represents the longest RDD chains that may be reused (e.g., in the example of FIGS. **9**A-**9**C, if the new RDD is R_{15}, then R_4 may be returned as r_M). In act **1012**, the cache manager **110** then generates the remaining chain from the detected reusable RDD r_M (e.g., in the example of FIGS. **9**A-**9**C, the cache manager **110** generates R_{13}, R_{14}, and then R_{15}). Otherwise, if the search returns a nil value in act **1010**, the cache manager **110** checks, in act **1014**, whether the cache (e.g., the storage memory **306**_a_) has sufficient size (space) for this new RDD $((C-\sum_{i=1}^{N_{\mathbb{R}}}|\mathbb{R}|)<|r|)$. If there is insufficient space, the cache manager **110** then selects one or more victim RDDs to evict from the cache, in act **1016**. The cache manager **110** then generates (computes) the new RDD r if necessary (i.e., needToGenRDDFlag is False, in act **1018**) and inserts it into the memory, in acts **1020** and **1022**. Acts **1004** and **1006** together represent a cache hit (act **1007**), acts **1008** to **1012** together represent a cache deduplication process **1013**, and acts **1014** to **1022** represent a cache miss (act **1023**).

[0084] As acts **1014** to **1022** are substantially similar to acts **508** to **516** of FIG. **5**, a detailed description thereof may not be repeated. The process **1000** is also shown as the pseudo code below:

Process 1000

```
Procedure RDDCache (r)
    for each new RDD r do
        needToGenRDDFlag = True
        if r ∈ℝ then
            load r
        else
            r_M = ℝ .matchGenChain(r)
            if r_M ≠ Nil then
                r = genRDDFrom(r_M,r)
                needToGenRDDFlag = False
```

-continued

Process 1000

```
            if (C_S − Σ_{i=1}^{N_ℝ} |ℝ |) < |r| then
                𝕍 = Ø
                𝕍 =ℝ .selectVictimRDDs( 𝕍 ,r)
                updateScoreTable(ℝ , 𝕍 )
                ℝ .evict( 𝕍 )
                if needToGenRDDFlag = True then
                    r = generate(r)
                ℝ .insert(r)
    return
```

[0085] FIG. **11** is a flow diagram illustrating the process **1008** of matching generation chains (i.e., the matchGenChain function) performed by the cache manager **110**, according to some example embodiments of the present invention.

[0086] Process **1008** (the matchGenChain function) implements what may be called the RDD detection policy. The cache manager **110** may operate to detect those RDDs and corresponding operations that users are not explicitly declaring in their application. To this end, the cache manager **110** first searches each stored RDD from inside to outside (e.g., left to right or "source to destination" in FIGS. **9**A/**9**B/**9**C) of their generating chain (source RDD, to each functions). According to some embodiments, the cache manager **110** identifies (finds) the longest reusable chain and reuses that as a detected chain. The process **1008** is also shown in pseudo code below:

Process 1008

```
Procedure matchGenChain (r)
    for each candidate RDD r_C do
        matchFuncFlag = True
        If matchRDD(r_C.srcRDD,r.srcRDD) and
        |r_C.getGenChain( )| ≤
|r.getGenChain( )| then
            for i ∈ [1, |r_C.getGenChain( )|] do /* from inside
            to outside */
                if matchFunc(r_C.func[i],r.func[i]) == False then
                    matchFuncFlag = False
                    break
            if matchFuncFlag = True then
            𝕄 += r_C
    return maxGenChainLen( 𝕄 )
```

[0087] where $\mathbb{M}$ represents a set of victim RDDs to be evicted from the storage memory **306***a*.

[0088] In acts **1100** to **1106**, the cache manager **110** checks whether the iterated RDD r_c has the same source RDD as the requested new RDD r. The cache manager **110** further ensures that candidate cached RDD r_C is not longer than the new RDD r, because the cache manager **110** aims to reuse cached RDD(s) as part of the new RDD r. In acts **1108** to **112**, the cache manager **110** iterates the comparison process following a from-inside-to-outside-of-functions order to check whether each operation (i.e., function) of the current RDD matches the corresponding function of the new RDD r. For example, if $r_C=f_4(f_3(f_2(f_1(f_0(R_0)))))$, then the cache manager **110** checks R_0, f_0, f_1, f_2, f_3, and f_4 of r_C against the generating logic chain of the new RDD r. Only when all the iterated functions of a cached RDD match the new RDD's functions, the cache manager **110** adds the current iterated cached RDD into the candidate set $\mathbb{M}$, in acts **1114** and **116**. Finally, in acts **1118**, the cache manager **110** returns the longest RDD GenChain that may be reused by the new RDD r from the candidate set $\mathbb{M}$ (via the maxGenChainLen function). In other words, the cache manager **110** identifies a subset of intermediate data, each one of the subset of intermediate data having a same intermediate data source as the new intermediate data and a generating logic chain that is not longer than that of the new intermediate data. The cache manager **110** further compares functions of the generating logic chain of each one of the subset of intermediate data to those of the new intermediate data, in a sequence, to identify the reusable intermediate data as one of the subset of intermediate data having a longest sequence of matching functions with the new intermediate data.

[0089] According to some embodiments, the process **1008** described above may be further improved (e.g., optimized or sped up) by using some advanced data structure to record the detected duplicated RDDs and corresponding generating chains (such as a mapping table with assign functions of both RDD and functions) and adding them into the score table (e.g., Table 2). For example, if RDD_1 and RDD_2 are detected as identical, then embodiments of the present invention record both RDD_1 and RDD_2 in one row together (in the RDD column) with their GenChains (in the GenChain column), which mean the row will be [R_1,R_2; R_1's GenChain, R_2's GenChain; . . .]. Additionally, performance may be improved by further analyzing and making decisions on whether to store the RDD in the memory (e.g., the storage memory **306***a*) or not (i.e., re-compute next time), and advanced techniques to achieve that goal may be based on any suitable method known in the art (e.g., Kathpal et al., "Analyzing Compute vs. Storage Tradeoff for Video-aware Storage Efficiency," HotStorage (2012); and Khanafer et al., "The Constrained Ski-Rental Problem and its Application to Online Cloud Cost Optimization," INFOCOM, 2013 Proceedings IEEE. IEEE (2013), the entire contents of which are incorporated herein by reference).

[0090] FIG. **12** is a flow diagram illustrating the process **1104** of matching RDDs (i.e., the matchRDD function) performed by the cache manager **110**, according to some example embodiments of the present invention.

[0091] According to some embodiments, the cache manager **110** first checks, in act **1200**, whether the two input RDDs have the same size and the same type (in terms of data structures, objects, etc.; act **1200**). If the stated conditions are satisfied, the cache manager **110** uses a byte-to-byte comparison (the byteToByteCompare function; act **1202**) to further detect whether the two input RDDs are identical. However, embodiments of the present invention are not limited a byte-to-byte comparison, and any suitable comparison may be utilized. The process **1104** is also shown in pseudo code below:

Process 1104

```
Procedure matchRDD (r_1,r_2)
    if |r_1| == |r_2| and r_1.type( ) == r_2.type( ) then
        return byteToByteCompare(r_1,r_2)
    else
        return Flase
```

[0092] In Spark, operations (i.e., functions) can be categorized into transformations, actions and other categories. Table 3 lists some example RDD transformations and actions available in Spark, where a signature of each operation is chosen to match other APIs in Scala and other functional languages, and type parameters are shown in square brackets (Seq[T] denotes a sequence of elements of type T). A difference between transformations and actions is that the former are lazy operations that define a new RDD, while the latter launch a computation to return a value to the program or write data to external storage.

TABLE 3

Categories	Functions	Details
Transformations	map(f: T ⇒ U)	RDD[T] ⇒ RDD[U]
	filter(f: T ⇒ Bool)	RDD[T] ⇒ RDD[T]
	flatMap(f: T ⇒ Seq[U])	RDD[T] ⇒ RDD[U]
	sample(fraction: Float)	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	RDD[(K,V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f: (V, V) ⇒ V)	RDD[(K,V)] ⇒ RDD[(K, V)]
	union()	(RDD[T], RDD[T] ⇒ RDD[T]
	join()	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f: V ⇒ W)	RDD[(K,V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c: Comparator[K]	RDD[(K,V)] ⇒ RDD[(K, V)]
	partitionBy(p: Partitioner[K])	RDD[(K,V)] ⇒ RDD[(K, V)]
Actions	count()	RDD[T] ⇒ Long
	collect()	RDD[T] ⇒ Seq[T]
	reduce(f: (T, T) ⇒ T)	RDD[T] ⇒ T
	lookup(k, K)	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(path: String)	Outputs RDD to a storage system, e.g., HDFS

[0093] According to some embodiments of the present invention, the cache manager **110** detects those identical functions that result in exactly the same output when supplied with the same input, while these functions are not explicitly hardcoded by the users. Identical functions may be uniquely determined according to the type (signature) and input RDD(s) (act **1200**). For example, in FIG. **9**A, $R_0=R_8$, $f_1(\ )=f_9(\ )$, etc. Accordingly, the cache manager **110** may merge these duplicated generating chains and refine (e.g., reduce or minimize) the topological graph of RDD DAG, and thus, reduce system resource consumption, such as amount of used memory (e.g., local/remote volatile and/or non-volatile memory).

[0094] In some embodiments, the cache manager **110** differentiates between deterministic and non-deterministic operations and automatically tracks changes to the stored RDDs and their generating chains (e.g., time cost of re-computing if a cached RDD's parent RDD is evicted). However, not all functions can be uniquely determined by their signature/type and input. As such, according to some examples, the cache manager **110** operates only on deterministic functions, which may be determined uniquely (e.g., groupByKey, reduceByKey and sort can result in an identical hash or range partitioned RDD with the same input RDD(s)), and may not operate on non-deterministic functions, such as shuffling (or network traffic in general), which may output values in a non-deterministic manner.

[0095] FIG. **13** is a flow diagram illustrating the process **1110** of matching functions (i.e., the matchFunc function) performed by the cache manager **110**, according to some example embodiments of the present invention.

[0096] In act **1300**, the cache manager **110** first determines whether the two input functions are deterministic functions, which, for example, may be determined by searching through a list of preset deterministic functions (such as those listed in Table 3). If both input functions are deterministic, in act **1302**, the cache manager **110** then compares the two functions to determine whether they are of the same type (i.e., signature) or not. In act **1302**, the cache manager **110** returns the result of this comparison. If either of these two functions is not a deterministic function, the cache manager **110** returns False in act **1304**. The process **1110** is also shown in pseudo code below:

Process 1110

```
Procedure matchFunc (func_1,func_2)
    if isDetermisticFunc(func_1) and isDetermisticFunc(func_2) then
        return sameTypeFunc(func_1,func_2)
    else
        return False
```

[0097] FIG. **14** is a flow diagram illustrating the process **1012** of obtaining the longest reusable chain (i.e., the genRDDFrom function) performed by the cache manager **110**, according to some example embodiments of the present invention.

[0098] In act **1400** to **1406**, the cache manager **110** computes the longest reusable RDD generation logic chain (GenChain) by calling runTransformation from the longest reusable generating logic chain, following the new RDD r's remaining generation logic chain. The process **1012** is also shown in pseudo code below:

Process 1012

```
Procedure genRDDFrom (r_M,r)
    currR = r_M
    for i ∈ [|r.getGenChain( )|− |r_M.getGenChain( )|,
    |r.getGenChain( )|] do
        currR = runFunction(r.func[i],currR)
    return currR
```

[0099] It will be understood by a person of ordinary skill in the art that the flow diagrams and the pseudo-code presented above are only for illustrating the concepts of the present invention, and that the processes described may be modified in a suitable manner to be performed more efficiently using more advanced approaches. For example, in process **512**, dynamic programming may be adopted to avoid updating those r_Vs that will be evicted later (i.e., $r_V \in \mathbb{V} \cap \mathbb{R}$).

[0100] It will be understood that, although the terms "first," "second," "third," etc., may be used herein to describe various elements, components, regions, layers, and/or sections, these elements, components, regions, layers, and/or sections should not be limited by these terms. These terms are used to distinguish one element, component, region, layer, or section from another element, component, region, layer or section. Thus, a first element, component, region, layer, or section described herein could be termed a second element, component, region, layer, or section, without departing from the spirit and scope of the present invention.

[0101] The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the present invention. As used herein, the singular forms "a" and "an" are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms "comprises," "comprising," "includes," and "including," when used in this specification, specify the presence of the stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components, and/or groups thereof. As used herein, the term "and/or" includes any and all combinations of one or more of the associated listed items. Expressions such as "at least one of," when preceding a list of elements, modify the entire list of elements and do not modify the individual elements of the list.

[0102] As used herein, the term "substantially," "about," and similar terms are used as terms of approximation and not as terms of degree, and are intended to account for the inherent variations in measured or calculated values that would be recognized by those of ordinary skill in the art. Further, the use of "may" when describing embodiments of the present invention refers to "one or more embodiments of the present invention." As used herein, the terms "use," "using," and "used" may be considered synonymous with the terms "utilize," "utilizing," and "utilized," respectively. Also, the term "exemplary" is intended to refer to an example or illustration.

[0103] When a certain embodiment may be implemented differently, a specific process order may be performed differently from the described order. For example, two consecutively described processes may be performed substantially at the same time or performed in an order opposite to the described order.

[0104] The electronic or electric devices and/or any other relevant devices or components according to embodiments of the present invention described herein may be implemented utilizing any suitable hardware, firmware (e.g., an application-specific integrated circuit), software, or a combination of software, firmware, and hardware. For example, the various components of these devices may be formed on one integrated circuit (IC) chip or on separate IC chips. Further, the various components of these devices may be implemented on a flexible printed circuit film, a tape carrier package (TCP), a printed circuit board (PCB), or formed on one substrate. Further, the various components of these devices may be a process or thread, running on one or more processors, in one or more computing devices, executing computer program instructions and interacting with other system components for performing the various functionalities described herein. The computer program instructions are stored in a memory which may be implemented in a computing device using a standard memory device, such as, for example, a random access memory (RAM). The computer program instructions may also be stored in other non-transitory computer readable media such as, for example, a CD-ROM, flash drive, or the like. Also, a person of skill in the art should recognize that the functionality of various computing devices may be combined or integrated into a single computing device, or the functionality of a particular computing device may be distributed across one or more other computing devices without departing from the spirit and scope of the exemplary embodiments of the present invention.

[0105] Unless otherwise defined, all terms (including technical and scientific terms) used herein have the same meaning as would be commonly understood by one of ordinary skill in the art to which the present invention belongs. It will be further understood that terms, such as those defined in commonly used dictionaries, should be interpreted as having meanings that are consistent with their meanings in the context of the relevant art and/or the present specification, and should not be interpreted in an idealized or overly formal sense, unless expressly so defined herein.

[0106] While this invention has been described in detail with particular references to illustrative embodiments thereof, the embodiments described herein are not intended to be exhaustive or to limit the scope of the invention to the exact forms disclosed. Persons skilled in the art and technology to which this invention pertains will appreciate that alterations and changes in the described structures and methods of assembly and operation can be practiced without meaningfully departing from the principles, spirit, and scope of this invention, as set forth in the following claims and equivalents thereof.

What is claimed is:

1. A cache management system for managing a plurality of intermediate data, the cache management system comprising:

a processor; and

a memory having stored thereon the plurality of intermediate data and instructions that when executed by the processor, cause the processor to perform:

identifying a new intermediate data to be accessed;

loading the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data;

in response to not identifying the new intermediate data as one of the plurality of intermediate data:

selecting a set of victim intermediate data from the plurality of intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data, the plurality of scores being based on a score table stored in the memory;

evicting the set of victim intermediate data from the memory;

updating the score table based on the set of victim intermediate data; and

adding the new intermediate data to the plurality of intermediate data stored in the memory.

2. The cache management system of claim 1, wherein the score table comprises for each intermediate data of the plurality of intermediate data a number of children of the intermediate data, a size of the intermediate data, a re-compute time of the intermediate data, and a generating logic chain of the intermediate data.

3. The cache management system of claim 1, wherein each one of the plurality of intermediate data is a resilient distributed data (RDD), and

wherein the cache management system operates under a Spark framework.

4. The cache management system of claim 1, wherein the selecting of the set of victim intermediate data comprises:

identifying a first set of intermediate data of the plurality of intermediate data, each one of the first set of intermediate data having a size less than that of the new intermediate data.

5. The cache management system of claim 4, wherein the selecting of the set of victim intermediate data comprises:

in response to the first set of intermediate data not being empty:

sorting the first set of intermediate data in a sorted order based on first scores associated with respective ones of the first set of intermediate data; and

adding ones of the first set of intermediate data to the set of victim intermediate data to be evicted, based on the sorted order, while a cumulative size of the set of victim intermediate data is less than the new intermediate data.

6. The cache management system of claim 5, wherein the order is an ascending order.

7. The cache management system of claim 5, wherein the selecting of the set of victim intermediate data further comprises:

generating each one of the first scores based on a re-compute time of, a number of children of, and a size of an associated one of the first set of intermediate data stored at the score table.

8. The cache management system of claim 5, wherein the generating of each one of the first scores comprises:

calculating a product of a re-compute time and a number of children of an associated one of the first set of intermediate data; and

dividing the calculated product by a size of the associated one of the first set of intermediate data to calculate the one of the first scores.

9. The cache management system of claim 4, wherein the selecting of the set of victim intermediate data comprises:

in response to the first set of intermediate data having a size smaller than the new intermediate data:

emptying the set of victim intermediate data;

identifying a second set of intermediate data of the plurality of intermediate data, each one of the second set of intermediate data having a size greater than or equal to that of the new intermediate data;

generating second scores associated with respective ones of the plurality of intermediate data based on the score table; and

selecting one intermediate data, as the set of victim intermediate data, from a second set of intermediate data having a lowest associated score among the second scores.

10. The cache management system of claim 9, wherein the generating of the second scores comprises:

generating each one of the second scores based on a re-compute time of, a number of children of, and a size of an associated one of the second set of intermediate data stored at the score table.

11. The cache management system of claim 1, wherein the updating of the score table comprises:

for each one of the set of victim intermediate data:

identifying a child intermediate data associated with the one of the set of victim intermediate data;

calculating a re-compute time of the child intermediate data based on a re-compute time of the one of the set of victim intermediate data; and

updating the score table based on the calculated re-compute time of the child intermediate data.

12. A cache management system for managing a plurality of intermediate data, the cache management system comprising:

a processor; and

a memory having stored thereon a plurality of intermediate data and instructions that when executed by the processor, cause the processor to perform:

identifying a new intermediate data to be accessed;

loading the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data;

in response to not identifying the new intermediate data as one of the plurality of intermediate data:

generating a plurality of scores associated with respective ones of the plurality of intermediate data, each of the plurality of scores being based on a re-compute time of, a number of children of, and a size of an associated one of the plurality of intermediate data stored at a score table in the memory;

selecting a set of victim intermediate data from the plurality of intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data;

evicting the set of victim intermediate data from the memory;

updating the score table based on the set of victim intermediate data; and

adding the new intermediate data to the plurality of intermediate data stored in the memory.

13. A method of managing a plurality of intermediate data stored in a memory, the method comprising:

identifying, by a processor, a new intermediate data to be accessed;

loading, by the processor, the intermediate data from the memory in response to identifying the new intermediate data as one of the plurality of intermediate data;

in response to not identifying the new intermediate data as one of the plurality of intermediate data:

selecting, by the processor, a set of victim intermediate data from the plurality of intermediate data to evict from the memory based on a plurality of scores associated with respective ones of the plurality of intermediate data, the plurality of scores being based on a score table stored in the memory;

evicting, by the processor, the set of victim intermediate data from the memory;

updating, by the processor, the score table based on the set of victim intermediate data; and adding, by the processor, the new intermediate data to the plurality of intermediate data stored in the memory.

14. The method of claim **13**, wherein the score table comprises for each intermediate data of the plurality of intermediate data a number of children of the intermediate data, a size of the intermediate data, a re-compute time of the intermediate data, and a generating logic chain of the intermediate data.

15. The method of claim **13**, wherein the selecting of the set of victim intermediate data comprises:

identifying a first set of intermediate data of the plurality of intermediate data, each one of the first set of intermediate data having a size less than that of the new intermediate data.

16. The method of claim **15**, wherein the selecting of the set of victim intermediate data comprises:

in response to the first set of intermediate data not being empty:

sorting the first set of intermediate data in a sorted order based on first scores associated with respective ones of the first set of intermediate data; and

adding ones of the first set of intermediate data to the set of victim intermediate data to be evicted, based on the sorted order, while a cumulative size of the set of victim intermediate data is less than or equal to the new intermediate data, wherein the order is an ascending order.

17. The method of claim **16**, wherein the selecting of the set of victim intermediate data further comprises:

generating each one of the first scores based on a re-compute time of, a number of children of, and a size of an associated one of the first set of intermediate data stored at the score table.

18. The method of claim **15**, wherein the selecting of the set of victim intermediate data comprises:

in response to the first set of intermediate data having a size smaller than the new intermediate data:

emptying the set of victim intermediate data;

identifying a second set of intermediate data of the plurality of intermediate data, each one of the second set of intermediate data having a size greater than or equal to that of the new intermediate data;

generating second scores associated with respective ones of the plurality of intermediate data based on the score table; and

selecting one intermediate data, as the set of victim intermediate data, from a second set of intermediate data having a lowest associated score among the second scores.

19. The method of claim **18**, wherein the generating of the second scores comprises:

generating each one of the second scores based on a re-compute time of, a number of children of, and a size of an associated one of the second set of intermediate data stored at the score table.

20. The method of claim **13**, wherein the updating of the score table comprises:

for each one of the set of victim intermediate data:

identifying a child intermediate data associated with the one of the set of victim intermediate data;

calculating a re-compute time of the child intermediate data based on a re-compute time of the one of the set of victim intermediate data; and

updating the score table based on the calculated re-compute time of the child intermediate data.

* * * * *