(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 3 889 776 A1

(12) EUROPEAN PATENT APPLICATION
published in accordance with Art. 153(4) EPC

(43) Date of publication:
06.10.2021 Bulletin 2021/40

(21) Application number: **19898187.0**

(22) Date of filing: **03.12.2019**

(51) Int Cl.:
G06F 9/54 *(2006.01)*

(86) International application number:
PCT/CN2019/122523

(87) International publication number:
WO 2020/125396 (25.06.2020 Gazette 2020/26)

(84) Designated Contracting States:
AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HR HU IE IS IT LI LT LU LV MC MK MT NL NO PL PT RO RS SE SI SK SM TR
Designated Extension States:
BA ME
Designated Validation States:
KH MA MD TN

(30) Priority: **17.12.2018 CN 201811545173**

(71) Applicant: **Huawei Technologies Co., Ltd.**
Shenzhen Guangdong 518129 (CN)

(72) Inventors:
- **TAN, Jun**
Shenzhen, Guangdong 518129 (CN)
- **YUAN, Zhengfan**
Shenzhen, Guangdong 518129 (CN)

(74) Representative: **Epping - Hermann - Fischer Patentanwaltsgesellschaft mbH**
Schloßschmidstraße 5
80639 München (DE)

(54) PROCESSING METHOD AND DEVICE FOR SHARED DATA AND SERVER

(57) A method for processing shared data, an apparatus, and a server are provided, and relate to the field of communications technologies, so that shared data can be cached across application programs, which facilitates data sharing across applications, can reduce repeated computations, and helps accelerate a computing speed of a Spark architecture. The method includes: receiving a first instruction; starting a first Spark context for a first application program, to create a DAG of the first application program, and caching the DAG of the first application program in a first area of a first server; receiving a second instruction; starting a second Spark context for a second application program, to create a DAG of the second application program, and caching the DAG of the second application program in the first area of the first server; reading m DAGs from the first area, where the m DAGs include the DAG of the first application program and the DAG of the second application program; determining a to-be-cached shareable RDD based on the m DAGs; and caching the to-be-cached shareable RDD in a main memory of a second server, where the shareable RDD is an RDD included in at least two DAGs of the m DAGs.

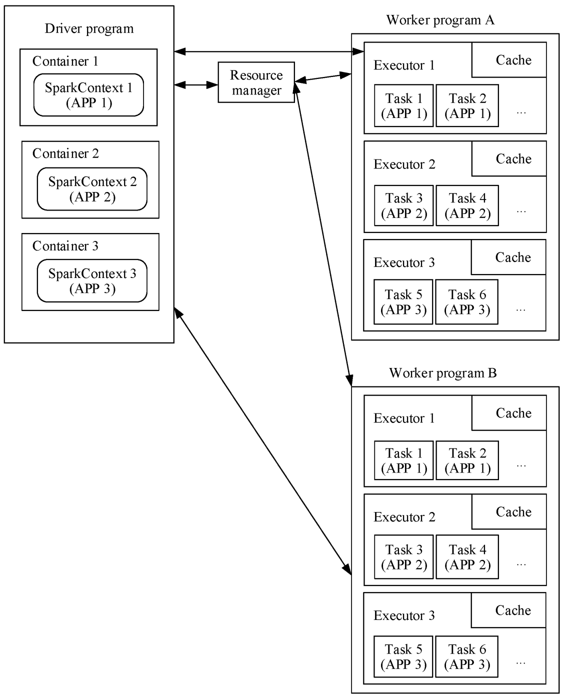

FIG. 5A

Description

[0001] This application claims priority to Chinese Patent Application No. 201811545173.6, filed with China National Intellectual Property Administration on December 17, 2018 and entitled "METHOD FOR PROCESSING TO-BE-SHARED DATA, APPARATUS, AND SERVER", which is incorporated herein by reference in its entirety.

TECHNICAL FIELD

[0002] This application relates to the field of communications technologies, and in particular, to a method for processing to-be-shared data, an apparatus, and a server.

BACKGROUND

[0003] Spark is a fast and universal computing engine for processing large-scale data. FIG. 1 is a diagram of a Spark architecture. The Spark architecture uses a master-slave (Maser-Slave) model in distributed computing, and includes a master node 100 and at least one slave node 200. The master node 100 and the slave node 200 form a cluster, and jointly execute one or more application (application) programs.

[0004] In a process of Spark executing an application program, the master node 100 starts a driver (Driver) program (the master node may also be referred to as a driver node), and the slave nodes 200 separately start their own worker (Worker) programs (the slave node may also be referred to as a worker node). The driver program is a start point of application program execution, and is responsible for distributing a specific task of an application program. The worker programs separately create a dedicated executor (Executor) process of the application program, and run, in a multi-thread manner in the executor process, a task allocated by the driver program.

[0005] Therefore, from a scheduling perspective (each driver program schedules its own task) or from a running perspective (tasks of different application programs run in different executors), execution of application programs in Spark is isolated from each other. In other words, data cannot be shared among application programs in Spark.

[0006] However, in an actual scenario, a large amount of same data may exist in different application programs, for example, in different application programs of a same service. Data cannot be shared among the application programs. As a result, Spark performs a large quantity of repeated computations and stores a large amount of duplicate data, which affects a computing speed of Spark.

SUMMARY

[0007] This application provides a method for processing to-be-shared data, an apparatus, and a server, so that to-be-shared data can be cached across application programs, which facilitates data sharing across applications, can reduce repeated computations, and helps accelerate a computing speed of a Spark architecture and a running speed of a device in the Spark architecture, and improve device performance.

[0008] According to a first aspect, this application provides a method for processing to-be-shared data, where the method is applied to a server cluster, and the method includes:

[0009] A first server receives a first instruction. The first server starts a first Spark context for a first application program, creates a directed acyclic graph DAG of the first application program, and caches the DAG of the first application program in a first area of the first server. The first server receives a second instruction. The first server starts a second Spark context for a second application program, creates a DAG of the second application program, and caches the DAG of the second application program in the first area of the first server. The first server reads m DAGs from the first area, where the m DAGs include the DAG of the first application program and the DAG of the second application program. The first server determines a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs, and caches the to-be-cached shareable RDD in memory of the server cluster, where the shareable RDD is an RDD included in at least two DAGs of the m DAGs.

[0010] Therefore, according to the method provided in this embodiment of this application, shareable RDDs in DAGs of a plurality of application programs may be determined and cached based on the DAGs of the plurality of application programs, thereby implementing caching of the shareable RDDs across applications. The DAGs of the plurality of application programs include more shareable RDDs. Therefore, when these shareable RDDs are used subsequently, a quantity of repeated computations performed by Spark can be greatly reduced and a computing speed of Spark can be accelerated.

[0011] In a possible implementation, that the first server determines a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs, and caches the to-be-cached shareable RDD in memory of the server cluster includes: computing a quantity of shareable RDDs in each of the m DAGs; determining a first DAG and a shareable RDD in the first DAG, where the first DAG is a DAG with a maximum quantity of shareable RDDs in the m DAGs;

determining a to-be-cached shareable RDD included in the first DAG based on remaining space in the memory of the server cluster, and caching the to-be-cached shareable RDD included in the first DAG; determining a second DAG and a shareable RDD in the second DAG, where the second DAG is a DAG with a maximum quantity of shareable RDDs in the m DAGs other than the first DAG; and determining a to-be-cached shareable RDD included in the second DAG based on remaining space in the memory of the server cluster, and caching the to-be-cached shareable RDD included in the second DAG.

[0012] Because a DAG with a maximum quantity of reusable RDDs is selected first, the reusable RDDs included in the DAG are cached. Because the DAG has a maximum quantity of RDDs that can be reused by another DAG, the reusable RDDs included in the DAG are preferentially cached. This can greatly reduce a quantity of repeated computations and reduce an amount of repeatedly cached data, which helps accelerate a computing speed of Spark and save cache costs.

[0013] In a possible implementation, the determining a to-be-cached shareable RDD included in the first DAG based on remaining space in the memory of the server cluster includes:
if the remaining space in the memory of the server cluster is greater than or equal to a sum of sizes of all shareable RDDs in the first DAG, determining all the shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or if the remaining space in the memory of the server cluster is less than the sum of the sizes of all the shareable RDDs in the first DAG, determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs.

[0014] In a possible implementation, the determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs includes:
determining, based on a size, a quantity of reuse times, or a computing time of each shareable RDD included in the first DAG, or a size of a dependent parent RDD of each shareable RDD included in the first DAG, some shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or determining a shareable RDD that is found by query in the first DAG as the to-be-cached shareable RDD.

[0015] In a possible implementation, the determining a to-be-cached shareable RDD included in the first DAG based on remaining space in the memory of the server cluster also includes:
if the remaining space in the memory of the server cluster is greater than or equal to a sum of sizes of all shareable RDDs in the first DAG, determining all the shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or if the remaining space in the memory of the server cluster is less than the sum of the sizes of all the shareable RDDs in the first DAG, determining a to-be-replaced RDD in the memory of the server cluster, and determining the to-be-cached shareable RDD included in the first DAG based on the remaining space in the memory of the server cluster and a size of the to-be-replaced RDD.

[0016] In a possible implementation, the determining a to-be-replaced RDD in the memory of the server cluster includes: determining the to-be-replaced RDD based on a size, a quantity of reuse times, or a computing time of each shareable RDD cached in the memory of the server cluster, or a size of a dependent parent RDD of each shareable RDD cached in the memory of the server cluster.

[0017] In a possible implementation, the determining the to-be-cached shareable RDD included in the first DAG based on the remaining space in the memory of the server cluster and a size of the to-be-replaced RDD includes:
if a sum of the remaining space in the memory of the server cluster and the size of the to-be-replaced RDD is greater than or equal to the sum of the sizes of all the shareable RDDs in the first DAG, determining all the shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or if the sum of the remaining space in the memory of the server cluster and the size of the to-be-replaced RDD is less than the sum of the sizes of all the shareable RDDs in the first DAG, determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs.

[0018] In a possible implementation, the determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs includes:
determining, based on a size, a quantity of reuse times, or a computing time of each shareable RDD included in the first DAG, or a size of a dependent parent RDD of each shareable RDD included in the first DAG, some shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or determining a shareable RDD that is found by query in the first DAG as the to-be-cached shareable RDD.

[0019] In a possible implementation, the caching the to-be-cached shareable RDD included in the first DAG includes: removing the to-be-replaced RDD from the memory of the server cluster, and caching the to-be-cached shareable RDD included in the first DAG.

[0020] In a possible implementation, the method further includes: determining a third DAG in the m DAGs; and reading a shareable RDD included in the third DAG from the memory of the server cluster, and transforming the third DAG into a fourth DAG based on the shareable RDD included in the third DAG, where the fourth DAG and the third DAG have a same execution result.

[0021] In a possible implementation, an execution time of the fourth DAG is less than an execution time of the third DAG.

[0022] According to a second aspect, a server is provided, including a processor, a memory, and a communications interface. The memory and the communications interface are coupled to the processor. The memory is configured to

store computer program code, where the computer program code includes a computer instruction. When the processor reads the computer instruction from the main memory, the server is enabled to perform the following operations:
receiving a first instruction; starting a first Spark context for a first application program, creating a directed acyclic graph DAG of the first application program, and caching the DAG of the first application program in a first area of a first server; receiving a second instruction; starting a second Spark context for a second application program, creating a DAG of the second application program, and caching the DAG of the second application program in the first area of the first server; reading m DAGs from the first area, where the m DAGs include the DAG of the first application program and the DAG of the second application program; and determining a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs, and caching the to-be-cached shareable RDD in memory of a server cluster, where the shareable RDD is an RDD included in at least two DAGs of the m DAGs.

[0023] In a possible implementation, the determining a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs, and caching the to-be-cached shareable RDD in memory of a server cluster includes:
computing a quantity of shareable RDDs in each of the m DAGs; determining a first DAG and a shareable RDD in the first DAG, where the first DAG is a DAG with a maximum quantity of shareable RDDs in the m DAGs; determining a to-be-cached shareable RDD included in the first DAG based on remaining space in the memory of the server cluster, and caching the to-be-cached shareable RDD included in the first DAG; determining a second DAG and a shareable RDD in the second DAG, where the second DAG is a DAG with a maximum quantity of shareable RDDs in the m DAGs other than the first DAG; and determining a to-be-cached shareable RDD included in the second DAG based on remaining space in the memory of the server cluster, and caching the to-be-cached shareable RDD included in the second DAG.

[0024] In a possible implementation, the determining a to-be-cached shareable RDD included in the first DAG based on remaining space in the memory of the server cluster includes:
if the remaining space in the memory of the server cluster is greater than or equal to a sum of sizes of all shareable RDDs in the first DAG, determining all the shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or if the remaining space in the memory of the server cluster is less than the sum of the sizes of all the shareable RDDs in the first DAG, determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs.

[0025] In a possible implementation, the determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs includes:
determining, based on a size, a quantity of reuse times, or a computing time of each shareable RDD included in the first DAG, or a size of a dependent parent RDD of each shareable RDD included in the first DAG, some shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or determining a shareable RDD that is found by query in the first DAG as the to-be-cached shareable RDD.

[0026] In a possible implementation, the determining a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs, and caching the to-be-cached shareable RDD in memory of a server cluster further includes:
if the remaining space in the memory of the server cluster is greater than or equal to a sum of sizes of all shareable RDDs in the first DAG, determining all the shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or if the remaining space in the memory of the server cluster is less than the sum of the sizes of all the shareable RDDs in the first DAG, determining a to-be-replaced RDD in the memory of the server cluster, and determining the to-be-cached shareable RDD included in the first DAG based on the remaining space in the memory of the server cluster and a size of the to-be-replaced RDD.

[0027] In a possible implementation, the determining a to-be-replaced RDD in the memory of the server cluster includes:
determining the to-be-replaced RDD based on a size, a quantity of reuse times, or a computing time of each shareable RDD cached in the memory of the server cluster, or a size of a dependent parent RDD of each shareable RDD cached in the memory of the server cluster.

[0028] In a possible implementation, the determining the to-be-cached shareable RDD included in the first DAG based on the remaining space in the memory of the server cluster and a size of the to-be-replaced RDD includes:
if a sum of the remaining space in the memory of the server cluster and the size of the to-be-replaced RDD is greater than or equal to the sum of the sizes of all the shareable RDDs in the first DAG, determining all the shareable RDDs in the first DAG as the to-be-cached shareable RDDs; or if the sum of the remaining space in the memory of the server cluster and the size of the to-be-replaced RDD is less than the sum of the sizes of all the shareable RDDs in the first DAG, determining some shareable RDDs in the first DAG as the to-be-cached shareable RDDs.

[0029] In a possible implementation, the caching the to-be-cached shareable RDD included in the first DAG includes:
removing the to-be-replaced RDD from the memory of the server cluster, and caching the to-be-cached shareable RDD included in the first DAG.

[0030] In a possible implementation, the server further performs the following operations:
determining a third DAG in the m DAGs; and reading a shareable RDD included in the third DAG from the memory of the server cluster, and transforming the third DAG into a fourth DAG based on the shareable RDD included in the third DAG, where the fourth DAG and the third DAG have a same execution result.

[0031] According to a third aspect, a computer storage medium is provided, including a computer instruction. When the computer instruction runs on a server, the server is enabled to perform the method provided in any one of the first aspect or the possible implementations of the first aspect.

[0032] According to a fourth aspect, a computer program product is provided. When the computer program product runs on a computer, the computer is enabled to perform the method provided in any one of the first aspect or the possible implementations of the first aspect.

[0033] According to a fifth aspect, a server is provided, including one or more processors, and one or more memories coupled to the one or more processors. The one or more memories are configured to store computer program code, where the computer program code includes a computer instruction. When the one or more processors read the computer instruction from the one or more memories, the server is enabled to perform the steps described in the foregoing method embodiment.

BRIEF DESCRIPTION OF DRAWINGS

[0034]

FIG. 1 is a network structure diagram of a Spark framework in the prior art;
FIG. 2 is a schematic diagram of a method for transforming a DAG in the prior art;
FIG. 3 is a schematic diagram of a method for storing an RDD in the prior art;
FIG. 4 is a schematic flowchart of Spark running an application program in the prior art;
FIG. 5A is a schematic flowchart of Spark running a plurality of application programs in the prior art;
FIG. 5B is a schematic flowchart of Spark running a plurality of application programs according to an embodiment of this application;
FIG. 5C is a schematic flowchart of a method for caching a shareable RDD according to an embodiment of this application;
FIG. 6A is a schematic structural diagram of a server according to an embodiment of this application;
FIG. 6B is a schematic diagram of a framework of an RDD sharing system according to an embodiment of this application;
FIG. 7 is a schematic flowchart of another method for caching a shareable RDD according to an embodiment of this application;
FIG. 8 is a schematic diagram of a method for determining whether two RDDs are the same according to an embodiment of this application;
FIG. 9A and FIG. 9B are a schematic flowchart of still another method for caching a sharable RDD according to an embodiment of this application; and
FIG. 10 is a schematic flowchart of a method for using a shareable RDD according to an embodiment of this application.

DESCRIPTION OF EMBODIMENTS

[0035] The following describes the technical solutions in the embodiments of this application with reference to the accompanying drawings in the embodiments of this application. In description of the embodiments of this application, "/" means "or" unless otherwise specified. For example, A/B may represent A or B. In this specification, "and/or" describes only an association relationship for describing associated objects and represents that three relationships may exist. For example, A and/or B may represent the following three cases: Only A exists, both A and B exist, and only B exists.

[0036] The following terms "first" and "second" are merely intended for a purpose of description, and shall not be understood as an indication or implication of relative importance or implicit indication of the number of indicated technical features. Therefore, a feature limited by "first" or "second" may explicitly or implicitly include one or more features. In the description of the embodiment of this application, unless otherwise stated, "multiple" means two or more than two.

[0037] Spark is one of frameworks for a new generation of big data distributed processing and can be used in scenarios of large-scale data computations. For example, Spark can be applied to recommendation services or services for computing user preference. For example, on a server side, content that users are interested in may be determined based on a large amount of data about the users' use habits, for example, information such as duration or a time period of watching a specific type of video or a specific video, listening to a specific type music or a specific piece of music, or viewing a specific type of news, so that related content and the like can be pushed to the users on a terminal side.

[0038] In Spark, an innovative concept of resilient distributed dataset (Resilient Distributed Datasets, RDD) is proposed. An RDD is a distributed object set, and is essentially a read-only partition record set. In addition, each RDD may be divided into a plurality of partitions (Partition), and each partition is a dataset segment. In addition, different partitions of an RDD may be stored on different nodes in a cluster, so that parallel computations may be performed on different nodes in the cluster.

[0039] For better understanding of the technical solutions provided in the embodiments of this application, terms related to an RDD are first described.

[0040] RDD transformation (transformation) operation: Generate an RDD by using a data source (a text file, a chart file, or the like), or transform an existing RDD (for example, an RDD 1) into another RDD (for example, an RDD 2). The RDD 1 is referred to as a parent RDD of the RDD 2, and the RDD 2 is referred to as a child RDD of the RDD 1. Common transformation includes a map (map) operation, a filter (filter) operation, a flatMap (flatMap) operation, a union operation, a join operation, a reduceByKey operation, and the like.

[0041] Directed acyclic graph (Directed Acyclic Graph, DAG): A plurality of transformation operations are provided for an RDD, and each transformation operation generates a new RDD. In other words, a new RDD depends on an original RDD, and the dependency between the RDDs forms a DAG. For example, as shown in FIG. 2, a DAG 1 is that a map operation is performed on a text file (source data) to generate an RDD 1, and then a filter operation is performed to generate an RDD 2.

[0042] Dependency between RDDs: Different RDD transformation operations result in different dependencies between partitions of different RDDs. The dependencies are specifically divided into a narrow dependency (Narrow Dependency) and a wide dependency (Wide Dependency). In the narrow dependency, a partition of a parent RDD corresponds to a partition of a child RDD, or partitions of a plurality of parent RDDs correspond to a partition of a child RDD. In the wide dependency, a partition of a parent RDD corresponds to a plurality of partitions of a child RDD. The RDD dependency design enables an RDD to be fault-tolerant and helps accelerate an execution speed of Spark.

[0043] RDD sharing: During RDD transformation, if two or more DAGs have a same RDD, the same RDD can be reused by the DAGs, thereby implementing sharing. The same RDD may be referred to as a shareable RDD, or may be referred to as a reusable RDD. For example, still as shown in FIG. 2, by comparing the DAG 1 and a DAG 2, it can be learned that the DAG 1 and the DAG 2 have a same part, that is, a process of generating the RDD 1 by using a text file. To avoid repeated computations performed on the same part, the RDD 1 may be persisted (or cached) when the RDD 1 is generated after a map transformation operation is performed on the text file. In this way, the RDD 1 may be directly invoked when the RDD 2 and an RDD 3 are generated subsequently, where the RDD 1 is a shareable RDD or a reusable RDD.

[0044] RDD persistence: Not every RDD is stored during RDD transformation. If an RDD is to be reused or computing costs of the RDD are high, the RDD may be stored by using a method for displaying and invoking persist() or cache() provided by the RDD.

[0045] RDD storage: RDD data is stored in a block (block) form on a plurality of machines (that may be physical machines, virtual machines, or the like). Figure 3 shows an RDD storage architecture of Spark. Each worker node starts a slave block manager (BlockManagerSlave) to manage some blocks, but metadata of the blocks is stored by a master block manager (BlockManagerMaster) of a driver node. After generating a block, the BlockManagerSlave registers the block with the BlockManagerMaster. The BlockManagerMaster manages a relationship between an RDD and a block. When an RDD does not need to be stored any more, the BlockManagerMaster sends an instruction to the BlockManagerSlave to delete a corresponding block.

[0046] The following briefly describes a process of Spark running an application program. FIG. 4 is a schematic flowchart of Spark running an application program. The process includes the following steps.

[0047] Step 1: When starting an application program, a driver program first constructs a running environment of the application program, that is, starts a Spark context (SparkContext).

[0048] Step 2: The SparkContext applies for a resource from a resource manager, where the resource manager is configured to allocate a resource and monitor resource usage.

[0049] The resource manager may also be referred to as a cluster manager (Cluster Manager), and is an external service for obtaining a resource in the cluster. Spark supports different running modes, including Local, Standalone, Mesoses, and Yarn. The resource manager varies depending on the running mode. For example, in the Standalone running mode, the resource manager serves as native resource management of Spark, and a master is responsible for resource allocation. In the Hadoop Yarn running mode, the resource manager mainly refers to a resource manager (ResourceManager) in Yarn. In the Apache Mesos running mode, the resource manager is a resource scheduling framework that is well compatible with Hadoop MapReduce.

[0050] Step 3: The resource manager allocates a resource, and a dedicated executor of the application program is separately created in a plurality of worker programs.

[0051] Step 4: Each executor applies for a task from the SparkContext.

[0052] Step 5: The SparkContext loads source data (such as a text file and a chart file) from each executor, creates an RDD, and constructs a DAG graph. An RDD in the DAG graph is decomposed into stages, the stages are decomposed into tasks, and the tasks and dependent data of the tasks are serialized and then sent to each executor.

[0053] A stage includes a plurality of groups of tasks, which form a task set (TaskSet).

[0054] Step 6: Each executor runs the task allocated to it and releases all resources after the running is completed.

[0055] In the prior art, after receiving a request for starting an application program, a resource manager allocates a container (container) to a driver node. Then, the driver node starts a SparkContext of the application program in the container. That is, the resource manager allocates a container to each application program and a SparkContext of the application program is started in the container. In other words, SparkContexts of different application programs correspond to different containers. In addition, mutual data access cannot be performed among the different containers. Therefore, the SparkContexts of the different application programs cannot share data.

[0056] For example, FIG. 5A is a schematic diagram of Spark starting a plurality of application programs in the prior art. A resource manager allocates a container 1, a container 2, and a container 3 to an APP 1, an APP 2, and an APP 3 respectively. Then, a driver node starts a SparkContext 1 of the APP 1 in the container 1, starts a SparkContext 2 of the APP 2 in the container 2, and starts a SparkContext 3 of the APP 3 in the container 3. The SparkContext 1, the SparkContext 2, and the SparkContext 3 are located in different containers. Therefore, the SparkContext 1, the Spark-Context 2, and the SparkContext 3 cannot access each other's data. Subsequently, the three SparkContexts apply for resources of a plurality of worker nodes (for example, a worker program A and a worker program B), and create three executors on the plurality of worker nodes separately. Each executor executes a task corresponding to an application program of the executor. It can be learned that data cannot be shared among different application programs.

[0057] However, in an actual scenario, a large amount of duplicate data may exist in a plurality of application programs. For example, a large quantity of same RDDs usually exists in different application programs of same services (services of a video type, services of a music type, services of a payment type, or the like). If RDD sharing cannot be implemented among different application programs, a large quantity of repeated computations occur, wasting computing resources.

[0058] Therefore, an embodiment of this application provides a data sharing method that may be performed on a driver node, and can implement RDD data sharing among a plurality of application programs in Spark, which helps reduce a large quantity of repeated computations and accelerate a computing speed of Spark.

[0059] In this application, when a resource manager allocates a container to the driver node, the resource manager allocates one container to the plurality of application programs. Then, the driver node starts a SparkContext of each application program in the container. In other words, the SparkContexts of the plurality of application programs are started in the same container. Because the SparkContexts in the same container can access each other's data, the plurality of application programs in the same container can access each other's data. In other words, the driver node may access data of different application programs in the same container, for example, DAGs created by SparkContexts of different application programs in the same container. That is, the driver node may access DAGs of the plurality of application programs in the same container. In this way, the DAGs of the plurality of application programs in the same container may be processed in a centralized manner, to implement data sharing among the plurality of application programs, thereby reducing a quantity of repeated computations performed by Spark and accelerating a computing speed of Spark.

[0060] The plurality of application programs may be a plurality of application programs of a same service (or similar services). Because the plurality of application programs belongs to the same service or the similar services, they have a large amount of same data, and use same or similar computations. Then, a large quantity of same RDDs is generated. These same RDDs are shareable RDDs, and may also be referred to as reusable RDDs. Obviously, RDD sharing among the plurality of application programs can greatly avoid repeated computations, improve running speeds of the plurality of application programs, and accelerate a computing speed of a Spark system. It may be understood that the plurality of application programs may alternatively be other similar or associated application programs. A relationship between the plurality of application programs is not limited in this embodiment of this application.

[0061] For example, FIG. 5B is a schematic diagram of Spark starting a plurality of application programs according to this application. A resource manager allocates a container 1 to an APP 1, an APP 2, and an APP 3. Then, a driver node starts a SparkContext 1 of the APP 1, a SparkContext 2 of the APP 2, and a SparkContext 3 of the APP3 in the container 1. The SparkContext 1, the SparkContext 2, and the SparkContext 3 are located in the same container. Therefore, the SparkContext 1, the SparkContext 2, and the SparkContext 3 can access each other's data. In this case, for a DAG constructed by the SparkContext 1, a DAG constructed by the SparkContext 2, and a DAG constructed by the Spark-Context 3, the SparkContext 1, the SparkContext 2, and the SparkContext 3 can also access each other's DAG.

[0062] Simply put, when DAGs of a plurality of application programs in a same container are processed subsequently, reusable RDDs in the DAGs of the plurality of application programs may be determined and cached. Subsequently, the DAGs of the plurality of application programs are transformed based on the cached reusable RDDs. Post-transformation DAGs and pre-transformation DAGs have same computing results. However, the post-transformation DAGs can directly invoke the cached reusable RDDs during a computation, which greatly accelerates a computing speed. It should be noted that, in this embodiment of this application, the DAGs of the plurality of application programs are used as processing objects, the determined reusable RDDs may also be used across applications. That is, RDD data can be shared among the plurality of application programs in Spark.

[0063] FIG. 5C is a flowchart of a method for caching a shareable RDD according to an embodiment of this application. The method may be performed on a first server (that is, a driver node), and specifically includes the following steps.

[0064] S401: The first server receives a first instruction.

[0065] The first instruction may be an instruction for requesting the first server to start a first application program.

[0066] S402: The first server starts a first Spark context (for example, a SparkContext 1) for the first application program, creates a directed acyclic graph DAG of the first application program, and caches the DAG of the first application program in a first area of the first server.

[0067] The first area of the first server is a resource for which the first server applies from a resource manager, and may be understood as a container allocated by the resource manager to the first application program, as mentioned above.

[0068] S403: The first server receives a second instruction.

[0069] The second instruction may be an instruction for requesting the first server to start a second application program.

[0070] S404: The first server starts a second Spark context (for example, a SparkContext 2) for the second application program, creates a DAG of the second application program, and caches the DAG of the second application program in the first area of the first server.

[0071] It may be understood that the first server starts the second Spark context in the container allocated by the resource manager to the first application program, and then the DAG that is of the second application program and that is created by the second Spark context is cached in the container. That is, the DAG of the first application program and the DAG of the second application program are cached in the same container.

[0072] S405: The first server reads m DAGs from the first area, where the m DAGs include the DAG of the first application program and the DAG of the second application program.

[0073] S406: The first server determines a to-be-cached shareable RDD based on the m DAGs, and caches the to-be-cached shareable RDD in memory of a second server.

[0074] The shareable RDD is an RDD included in at least two DAGs of the m DAGs.

[0075] The memory of the second server is memory of the first server, memory of the second server, or a sum of all or some of the memory of the first server and all or some of the memory of the second server.

[0076] It should be noted that, in a Spark framework, memory may be shared among a plurality of servers, and as mentioned above, RDD data is stored in a block (block) form on a plurality of servers (that may be physical machines, virtual machines, or the like). Therefore, the memory of the second server refers to total memory of a plurality of servers in a cluster, and memory of a specific server or memory of a plurality of servers are not specifically distinguished.

[0077] The data sharing method provided in this embodiment of this application may be applied to the Spark framework shown in FIG. 1. The Spark framework includes one or more driver nodes and one or more worker nodes. The driver node and the worker node may be a same server or different servers. For example, the driver node and the worker node may be a server 300 shown in FIG. 6A.

[0078] FIG. 6A is a schematic diagram of a hardware structure of a server 300 disclosed in an embodiment of this application. The server 300 includes at least one processor 301, at least one memory 302, and at least one communications interface 303. Optionally, the server 300 may further include an output device and an input device, which are not shown in the figure.

[0079] The processor 301, the memory 302, and the communications interface 303 are connected by using a bus. The processor 301 may be a general-purpose central processing unit (Central Processing Unit, CPU), a microprocessor, an application-specific integrated circuit (Application-Specific Integrated Circuit, ASIC), or one or more integrated circuits configured to control program execution in the solutions of this application. The processor 301 may also include a plurality of CPUs, and the processor 301 may be a single-core (single-CPU) processor or a multi-core (multi-CPU) processor. The processor herein may be one or more devices, circuits, or processing cores configured to process data (for example, a computer program instruction).

[0080] In this embodiment of this application, the processor 301 is specifically configured to: compute a quantity of reusable RDDs in each DAG in a first window, determine a DAG with a maximum quantity of reusable RDDs as a first DAG, determine, based on remaining space in memory, whether all or some of the reusable RDDs in the first DAG may be cached, select a to-be-replaced RDD from cached RDDs for marking, and the like.

[0081] The memory 302 may be a read-only memory (Read-Only Memory, ROM) or another type of static storage device capable of storing static information and an instruction, or a random access memory (Random Access Memory, RAM) or another type of dynamic storage device capable of storing information and an instruction. The memory 302 may also be an electrically erasable programmable read-only memory (Electrically Erasable Programmable Read-Only Memory, EEPROM), a compact disc read-only memory (Compact Disc Read-Only Memory, CD-ROM) or other compact disc storage, optical disc storage (including a compressed optical disc, a laser disc, an optical disc, a digital versatile disc, a Blu-ray disc, and the like), a magnetic disk storage medium or another magnetic storage device, or any other medium capable of carrying or storing expected program code in a form of an instruction or a data structure and capable of being accessed by a computer, but is not limited thereto. The memory 302 may exist independently, and is connected to the processor 301 by using a bus. The memory 302 may be alternatively integrated with the processor 301. The memory 302 is configured to store application program code for executing the solutions of this application, and the execution is controlled by the processor 301. The processor 301 is configured to execute the computer program code stored in the memory 302, to implement the data sharing method in this embodiment of this application.

[0082] In this embodiment of this application, the memory 302 may be configured to store a reusable RDD, and the like.

[0083] The communications interface 303 may be configured to communicate with another device or communications network, such as an ethernet or a wireless local area network (wireless local area networks, WLAN).

[0084] In this embodiment of this application, the communications interface 303 may be specifically configured for communication between driver nodes, communication between worker nodes, communication between a driver node and a worker node, or the like.

[0085] The output device communicates with the processor, and may display information in a plurality of manners. For example, the output device may be a liquid crystal display (Liquid Crystal Display, LCD), a light-emitting diode (Light Emitting Diode, LED) display device, a cathode-ray tube (Cathode Ray Tube, CRT) display device, or a projector (projector). The input device communicates with the processor, and may receive, in a plurality of manners, input from a user. For example, the input device may be a mouse, a keyboard, a touchscreen device, or a sensing device.

[0086] The following describes in detail a method for sharing data in DAGs of a plurality of application programs in a same container with reference to accompanying drawings.

[0087] FIG. 6B is a schematic diagram of a framework of an RDD sharing system according to an embodiment of this application. The RDD sharing system may run on a driver node, and may include a DAG waiting queue, a DAG matcher, a DAG transformer, distributed shared memory (which may be referred to as shared memory for short), and a disk.

[0088] In some examples, the DAG waiting queue may include a first window and a second window. For DAGs arranged in the first window, when there is a vacancy in the second window, a DAG meeting a condition may be selected from the first window for place filling. That is, after transformation, a DAG enters the second window from the first window. DAGs arranged in the second window are waiting DAGs that can be executed when a Spark kernel is idle. For example, a DAG is decomposed into stages, the stages are decomposed into tasks, and then the tasks are distributed to worker nodes so that the worker nodes can run the tasks, and the like. In some other examples, the DAG waiting queue may further include a third window, and the third window includes a to-be-executed DAG arranged after the first window.

[0089] A size of each window may be understood as a quantity of DAGs, an amount of data, or the like included in each window. For example, a quantity of DAGs that can be arranged in the first window may be 10^3 to 10^5. In some examples, the size of each window may be determined based on service types (for example, a video service, a music service, or a payment service) of a plurality of application programs running in a cluster. The window size may vary according to the service type. In some other examples, the size of each window may also be determined based on a hardware capability (for example, a memory or CPU running rate) of a server in the cluster and a quantity of servers in the cluster. A larger quantity of servers and a stronger hardware capability indicate a larger size of each window. In still some other examples, the size of each window may be further determined based on an empirical value, a setting of a user, or the like. This is not limited in this embodiment of this application.

[0090] For example, referring to FIG. 6B, m DAGs are arranged in the first window, which are a DAG n+1, a DAG n+2, ..., and a DAG n+m separately; n DAGs are arranged in the second window, which are a DAG 1, a DAG 2, ..., and a DAG n separately; and a DAG n+m+1 and another to-be-executed DAG are arranged in the third window.

[0091] The DAG matcher is configured to determine a reusable RDD in a DAG in the first window, and cache the determined reusable RDD in the shared memory. In some embodiments, when space in the shared memory is insufficient for storing a to-be-cached RDD, some RDDs in the shared memory may be selected and stored in a disk, and space originally used for storing the RDDs is used for storing the to-be-cached reusable RDD.

[0092] The DAG transformer is configured to read a reusable RDD from the shared memory, transform a DAG in the first window based on the reusable RDD, and place a post-transformation DAG in the second window for waiting for execution.

[0093] The RDD sharing method provided in this application may include a process of caching a reusable RDD and a process of transforming a DAG based on a reusable RDD. The following describes the two processes separately.

[0094] 1. The process of caching a reusable RDD is described in the following.

[0095] FIG. 7 is a schematic flowchart of a method for caching a reusable RDD according to an embodiment of this application. The caching method specifically includes the following.

[0096] S101: There are m to-be-executed DAGs in a first window. A DAG matcher computes a quantity of reusable RDDs in each of the m to-be-executed DAGs.

[0097] A size of the first window may be understood as a quantity of DAGs, an amount of data, or the like included in the first window. For example, a quantity of DAGs that can be arranged in the first window may be 10^3 to 10^5. In some examples, the size of each window may be determined based on service types (for example, a video service, a music service, or a payment service) of a plurality of application programs running in a cluster. The size of the first window may be the same or different for different service types. In some other examples, the size of the first window may also be determined based on a hardware capability (for example, a memory or CPU running rate) of a server in the cluster and a quantity of servers in the cluster. A larger quantity of servers and a stronger hardware capability indicate a larger size of the first window. In still some other examples, the size of the first window may be further determined based on an empirical value, a setting of a user, or the like. This is not limited in this embodiment of this application.

[0098] S102: The DAG matcher determines a DAG with a maximum quantity of reusable RDDs as a first DAG in the m to-be-executed DAGs, and determines a reusable RDD in the first DAG.

[0099] In steps S101 and S102, that a DAG can reuse an RDD means that the DAG has the same RDD as another DAG in the first window, the same RDD means that a same task is executed, and therefore the RDD can be reused. That is, after the reusable RDD is cached, the another DAG that includes the reusable RDD may directly invoke the reusable RDD during a computation, thereby achieving effects of reducing a quantity of computations and accelerating a computing speed.

[0100] For determining whether two RDDs are the same, when the two RDDs meet the following conditions, it is considered that the two RDDs are the same, and the two RDDs may be determined as reusable RDDs. The conditions are described in the following.

[0101] Condition 1: Partitions (partition) are consistent.

[0102] Each RDD may be divided into a plurality of partitions. Each partition is a dataset segment and is a minimum unit for parallel RDD execution. In addition, different partitions of an RDD may be stored on different workers in the cluster, so that parallel computations may be performed on the different workers in the cluster. Therefore, when it is determined whether two or more RDDs are the same, it needs to be determined whether the two RDDs include a same quantity of partitions. Then, it is further determined, according to a condition 2, a condition 3, and a condition 4, whether content of each partition of the RDDs is the same.

[0103] Condition 2: A previous RDD that is depended on is the same.

[0104] As mentioned above, an RDD is obtained after a transformation operation is performed on an existing RDD. When it is determined whether two RDDs are the same, it needs to be determined whether the two RDDs depend on a same previous RDD (that is, a parent RDD). There may be one or more previous RDDs.

[0105] Condition 3: A previous transformation operation is the same.

[0106] As mentioned above, an RDD is obtained after a transformation operation is performed on an existing RDD. When it is determined whether two RDDs are the same, it needs to be determined whether a previous transformation operation performed on the two RDDs is the same.

[0107] Condition 4: Source data that is read is the same.

[0108] When an RDD is created, a dataset (that is, source data) in memory is read first, then a first RDD is created, and a subsequent RDD is formed after a plurality of transformation operations are performed on the first RDD. Therefore, when it is determined whether two or more RDDs are the same, it also needs to be determined whether source data that is read is the same.

[0109] For example, FIG. 8 is a schematic diagram of dependencies in a DAG 3. An RDD d includes three partitions. A partition 1 of the RDD d is obtained after an operation 1 (for example, join) is performed on a partition 1 of an RDD b and a partition 2 of an RDD c. A partition 2 of the RDD d is obtained after the operation 1 (for example, join) is performed on a partition 2 of the RDD b, a partition 1 of the RDD c, and the partition 2 of the RDD c. A partition 3 of the RDD d is obtained after the operation 1 (for example, join) is performed on a partition 3 of the RDD b, the partition 1 of the RDD c, and the partition 2 of the RDD c.

[0110] When it is determined whether an RDD (for example, an RDD e) is the same as the RDD d, a quantity of partitions of the RDD e is first determined, and then it is determined whether the quantity of partitions of the RDD e is the same as a quantity of partitions of the RDD d. If the quantity of partitions of the RDD e is the same as the quantity of partitions of the RDD d, it is considered that the partitions of the RDD e are consistent with the partitions of the RDD d. It is determined that previous parent RDDs that the RDD e depends on are an RDD x and an RDD y. Then it is further determined whether the RDD x and the RDD y are the same as the RDD b and the RDD c respectively. If the RDD x and the RDD y are the same as the RDD b and the RDD c respectively, it is considered that the RDD e and the RDD d depend on the same previous RDDs. It is determined that a previous transformation operation performed on the RDD e is an operation z. Then it is further determined whether the operation z is the same as the operation 1. If the operation z is the same as the operation 1, it is determined that the same previous transformation operation is performed on the RDD e and the RDD d. It is determined that source data used by the RDD e is source data i, source data j, and the like. Then it is determined whether the source data (the source data i, the source data j, and the like) used by the RDD e is the same as source data 1, source data 2, source data 3, source data 4, and source data 5 respectively. If the source data (the source data i, the source data j, and the like) used by the RDD e is the same as the source data 1, the source data 2, the source data 3, the source data 4, and the source data 5 respectively, it is determined that the RDD e and the RDD d have the same source data. When the four conditions are met, it may be determined that the RDD e is the same as the RDD d.

[0111] A quantity of reusable RDDs in a DAG is a total quantity of reusable RDDs determined after the DAG is compared with the other DAGs in the first window one by one.

[0112] For example, there are m DAGs in the first window, which are a DAG n+1, a DAG n+2, ..., and a DAG n+m separately. Computation of a quantity of reusable RDDs in the DAG n+1 is described. For example, the DAG n+1 may be separately compared with the other m-1 DAGs in the first window, to determine whether there is a reusable RDD. It is assumed that the DAG n+1 and the DAG n+2 have two reusable RDDs, which are an RDD 3 and an RDD 7 separately; the DAG n+1 and a DAG n+3 have two reusable RDDs, which are the RDD 3 and an RDD 5 separately; a DAG n+4 and the DAG n+1 have one reusable RDD, which is an RDD 2; a DAG n+5 and the DAG n+1 have one reusable RDD, which is an RDD 6; and the other DAGs and the DAG n+1 have no reusable RDD. In this case, there are five reusable RDDs in the DAG n+1, which are the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 separately. In addition, in the prior art.

[0113] By analogy, a quantity of reusable RDDs in another DAG in the first window is computed. It is assumed that there are three reusable RDDs in the DAG n+2, there are two reusable RDDs in the DAG n+3, and there is one reusable RDD in each of the other DAGs. In this case, it may be determined that the DAG n+1 has the maximum quantity of reusable RDDs. The reusable RDDs (the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6) included in the DAG n+1 may be cached.

[0114] In some embodiments, a DAG with a maximum quantity of reusable RDDs may be selected first, and the reusable RDDs included in the DAG may be cached. Because the DAG has the maximum quantity of RDDs that can be reused by another DAG, the reusable RDDs included in the DAG are preferentially cached. This can greatly reduce a quantity of repeated computations and reduce an amount of repeatedly cached data, which helps accelerate a computing speed of Spark and save cache costs.

[0115] S103: The DAG matcher determines, based on remaining space in shared memory, whether all reusable RDDs included in the first DAG can be cached. If all the reusable RDDs included in the first DAG can be cached, step S104 is performed. Otherwise, step S105 is performed.

[0116] Specifically, the DAG matcher reads the remaining space in the shared memory, and compares the remaining space with total space that needs to be occupied for caching all the reusable RDDs included in the first DAG. If the remaining space in the shared memory is greater than or equal to the total space that needs to be occupied by all the reusable RDDs included in the first DAG, it is determined that all the reusable RDDs included in the first DAG can be cached, which is referred to as "all can be cached" for short. Otherwise, it is determined that not all the reusable RDDs included in the first DAG can be cached.

[0117] For example, the DAG n+1 is still used as an example. There are five reusable RDDs in the DAG n+1, which are the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 separately. Total space that needs to be occupied for caching all the reusable RDDs included in the DAG n+1 indicates space in the shared memory that needs to be occupied when all the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 are cached in the shared memory.

[0118] In some examples, the DAG matcher may alternatively first determine whether all the reusable RDDs in the first DAG can be found by query. If all the reusable RDDs in the first DAG can be found by query, the DAG matcher may further determine whether all the reusable RDDs can be cached in the remaining space in the shared memory. If a reusable RDD is not found, it may be directly determined that not all the reusable RDDs can be cached. It is further determined whether some reusable RDDs that are found by query and that are depended on in the first DAG can be cached. That is, step S105 is performed.

[0119] S104: The DAG matcher caches all the reusable RDDs included in the first DAG in the shared memory.

[0120] Then, step S107 is performed.

[0121] S105: The DAG matcher determines a to-be-cached reusable RDD in the reusable RDDs included in the first DAG.

[0122] Specifically, the DAG matcher may select, based on the remaining space in the shared memory, some reusable RDDs from the reusable RDDs in the first DAG as to-be-cached reusable RDDs, and cache the to-be-cached reusable RDDs. That is, some of the reusable RDDs included in the first DAG are cached. That is, at least one reusable RDD included in the first DAG is not cached.

[0123] For example, the example in which the first DAG is the DAG n+1 is still used for description. There are five reusable RDDs in the DAG n+1, which are the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 separately. When it is determined in step S103 that the remaining space in the shared memory is insufficient for caching the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6, some RDDs may be selected from the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 for caching. Caching some of the reusable RDDs included in the DAG n+1 means that any one, two, three, or four of the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 may be cached.

[0124] In some embodiments, when the DAG matcher selects a to-be-cached reusable RDD from the reusable RDDs in the first DAG, the DAG matcher may determine the to-be-cached reusable RDD based on factors such as use frequency (a quantity of times that an RDD can be reused) of a reusable RDD, a computing time of a reusable RDD, a size of a parent RDD that a reusable RDD depends on, a size of a reusable RDD, or a weight of a reusable RDD. The DAG matcher may alternatively select and cache some of RDDs that are found by query. A method for selecting a reusable RDD is not limited in the embodiments of this application.

[0125] For example, a pre-determined quantity of reusable RDDs with higher use frequency may be selected from the reusable RDDs in the first DAG as the to-be-cached reusable RDDs. Alternatively, a pre-determined quantity of reusable RDDs with a longer computing time may be selected from the reusable RDDs in the first DAG as the to-be-cached reusable RDDs. Alternatively, a pre-determined quantity of reusable RDDs with a larger size of a dependent parent RDD may be selected from the reusable RDDs in the first DAG as the to-be-cached reusable RDDs. Alternatively, a pre-determined quantity of reusable RDDs with a larger reusable RDD size may be selected from the reusable RDDs in the first DAG as the to-be-cached reusable RDDs. Alternatively, a pre-determined quantity of reusable RDDs with a larger weight may be selected from the reusable RDDs in the first DAG as the to-be cached reusable RDDs.

[0126] For example, the example in which the first DAG is the DAG n+1 is still used for description. There are five reusable RDDs in the DAG n+1, which are the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 separately. When it is determined in step S103 that the remaining space in the shared memory is insufficient for caching the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6, some RDDs may be selected from the RDD 3, the RDD 7, the RDD 5, the RDD 2, and the RDD 6 for caching. It is assumed that, in the first window, the RDD 3 can be reused six times, the RDD 7 can be reused five times, the RDD 5 can be reused four times, the RDD 2 can be reused three times, and the RDD 6 can be reused two times. An RDD with the maximum quantity of reuse times may be selected for caching, for example, the RDD 3. Alternatively, a preset quantity of RDDs with a large quantity of reuse times may be selected for caching, for example, the RDD 3 and the RDD 7. Details are not described herein.

[0127] A weight of an RDD may be determined based on a service type of an application program, may be determined based on a setting of a user, or may be determined according to a predefined method. The following provides an example of a method for computing a weight of an RDD, that is, a weight of an RDD is determined according to a formula 1.

$$W(\mathrm{RDDi}) = \frac{Parent(\mathrm{RDDi}) \times \mathrm{c} \times \mathrm{F}}{RDDsize} \quad \text{(Formula 1)}$$

[0128] In the formula 1, W(RDDi) is a weight of an ith RDD; Parent(RDDi) is a sum of sizes of all parent RDDs of the ith RDD; c is computing cost of the ith RDD, and may reflect computing complexity of the ith RDD, for example, reflecting time spent in computing the ith RDD; F is a quantity of times that the ith RDD is used, that is, a quantity of times that the ith RDD can be reused; and RDDsize is a size of the ith RDD.

[0129] Computing cost of an RDD may be specifically computing complexity of a transformation operation for the RDD. Different transformation operations have different computing complexity. For example, common transformation operations mentioned above include map, filter, flatmap, reduceByKey, join, union, and the like. A simple comparison of computing complexity is provided here. Computing complexity of map is similar to that of filter. Computing complexity of flatmap is similar to that of reduceByKey. The computing complexity of map and the computing complexity of filter are lower than the computing complexity of flatmap and the computing complexity of reduceByKey. Computing complexity of join and computing complexity of union are related to a quantity of included partitions. A larger quantity of partitions indicates higher computing complexity. A specific algorithm for computing cost of an RDD is not limited in this application.

[0130] It should be further noted that, in the formula 1, a parameter on the right side of the equation may include any one or more of Parent(RDDi), c, F, and RDDsize, or more parameters are included. The method for computing a weight of an RDD is not limited in this application.

[0131] For example, the formula for computing a weight of an RDD may alternatively be any one of a formula 2 to a formula 7 in the following:

$$W(\mathrm{RDDi}) = \frac{Parent(\mathrm{RDDi}) \times \mathrm{c}}{RDDsize} \quad \text{(Formula 2)}$$

$$W(\mathrm{RDDi}) = \frac{Parent(\mathrm{RDDi}) \times \mathrm{F}}{RDDsize} \quad \text{(Formula 3)}$$

$$W(\mathrm{RDDi}) = \frac{\mathrm{c} \times \mathrm{F}}{RDDsize} \quad \text{(Formula 4)}$$

$$W(\mathrm{RDDi}) = Parent(\mathrm{RDDi}) \times \mathrm{c} \times \mathrm{F} \quad \text{(Formula 5)}$$

$$W(\mathrm{RDDi}) = Parent(\mathrm{RDDi}) \times \mathrm{c} \quad \text{(Formula 6)}$$

$$W(\mathrm{RDDi}) = \frac{Parent(\mathrm{RDDi})}{RDDsize} \quad \text{(Formula 7)}$$

[0132] In some other embodiments, the DAG matcher may preset a quantity of to-be-cached reusable RDDs, that is, statically specify the quantity of to-be-cached reusable RDDs as a preset quantity, for example, one or more. Alternatively, the DAG matcher may dynamically determine the quantity of to-be-cached reusable RDDs based on the remaining space in the shared memory. For example, a to-be-cached reusable RDD is first selected from the first DAG, and then it is determined, based on the remaining space in the shared memory and a size of the selected to-be-cached reusable RDD, whether the to-be-cached reusable RDD can be cached. Then, a next to-be-cached reusable RDD is selected from the first DAG, and so on until no more reusable RDD included in the first DAG can be cached in remaining space in the shared memory after a selected to-be-cached reusable RDD is cached.

[0133] S106: The DAG matcher caches the to-be-cached reusable RDD in the shared memory.

[0134] S107: The DAG matcher determines a DAG with a maximum quantity of reusable RDDs as a second DAG in the remaining m—1 to-be-executed DAGs (other than the first DAG) in the first window, and a shareable RDD in the second DAG.

[0135] For specific implementation of this step, refer to step S102. Details are not described herein again.

[0136] S108: The DAG matcher determines to cache all reusable RDDs included in the second DAG, or cache some of the reusable RDDs included in the second DAG.

[0137] For specific implementation of this step, refer to step S103 to step S106. Details are not described herein again.

[0138] By analogy, it is sequentially determined, based on an order of quantities of reusable RDDs, whether all or some of reusable RDDs included in each of the m DAGs in the first window can be cached. After it is determined whether all or some of the reusable RDDs included in the m DAGs in the first window can be cached, this process ends. In some examples, when the DAG matcher executes this process, if it is determined that remaining space in the shared memory reaches a specific threshold, it may be considered that the remaining space in the shared memory is insufficient for caching a subsequent reusable RDD, and the process may be stopped.

[0139] When there is a vacancy in the second window, a DAG arranged in the front in the first window enters the second window after being transformed by a DAG transformer. In this case, there is also a vacancy in the first window. Then, a DAG arranged in the front in a third window enters the first window. In other words, the m DAGs in the first window enter the second window one by one, and at the same time, new DAGs are added to the first window one by one from the third window.

[0140] In this case, the DAG matcher performs operations the same as those in steps S101 to S108 on a next batch of m DAGs arranged in the first window. Details are not described herein again.

[0141] In some embodiments of this application, when some DAGs in the next batch of m DAGs are arranged in the first window, the DAG matcher may start to perform the operations the same as those in steps S101 to S108 on the next batch of m DAGs. The DAG matcher may alternatively start to perform a corresponding operation only when the next batch of m DAGs are all arranged in the first window. This is not limited in the embodiments of this application.

[0142] By analogy, the DAG matcher performs the operations the same as those in steps S101 to S108 on every m DAGs in the first window. Details are not described herein again.

[0143] In some other embodiments, when remaining space in the shared memory is insufficient for all reusable RDDs in a DAG, these to-be-cached reusable RDDs may be used to replace some less important RDDs that are originally cached in the shared memory as to-be-replaced RDDs, for example, an RDD with a small weight. In other words, steps S104 to S106 in the foregoing embodiment may be replaced with steps S201 to S206. FIG. 9A and FIG. 9B are a flowchart of still another method for caching a reusable RDD according to an embodiment of this application. The method specifically includes steps S101 to S104, steps S201 to S206, and step S107. For steps S101 to S104 and step S107, refer to the foregoing description. Steps S201 to S206 are specifically described in the following.

[0144] S201: A DAG matcher selects a to-be-replaced RDD from RDDs cached in the shared memory, and marks the to-be-replaced RDD.

[0145] In some examples, the DAG matcher may select some less important RDDs as the to-be-replaced RDDs, where the to-be-replaced RDD may be an RDD with lower use frequency, an RDD with a shorter computing time, an RDD with a smaller size of a dependent parent RDD, an RDD with a larger size, an RDD with a smaller weight, or the like. This is not limited in this embodiment of this application.

[0146] S202: The DAG matcher determines, based on the remaining space in the shared memory and a size of the marked RDD, whether all the reusable RDDs included in the first DAG can be cached. If all the reusable RDDs included in the first DAG can be cached, step S204 is performed. Otherwise, step S203 is performed.

[0147] S203: The DAG matcher determines whether some of the reusable RDDs included in the first DAG can be cached. If some of the reusable RDDs included in the first DAG can be cached, step S204 is performed. Otherwise, step S206 is performed.

[0148] Specifically, the DAG matcher determines, based on the remaining space in the shared memory and the size of the marked RDD, whether some of the reusable RDDs included in the first DAG can be cached. For description of selection of some of the reusable RDDs from the first DAG, refer to related description in step S105. Details are not described herein again.

[0149] S204: The DAG matcher removes the marked RDD from the shared memory.

[0150] In some examples, the DAG matcher may directly delete the marked RDD from the shared memory, or may store the marked RDD in external storage, for example, in a disk.

[0151] S205: The DAG matcher caches, in the shared memory, all or some of the reusable RDDs included in the first DAG.

[0152] Then, step S107 and subsequent steps are performed.

[0153] S206: The DAG matcher selects a next to-be-replaced RDD other than the marked RDD from the RDDs cached in the shared memory, and marks the next to-be-replaced RDD.

[0154] Then, step S202 is performed.

[0155] 2. The process of transforming a DAG based on a reusable RDD is described in the following.

[0156] FIG. 10 is a schematic flowchart of a method for using a reusable RDD according to an embodiment of this application. The method specifically includes the following.

[0157] S301: When there is a vacancy in a second window, a DAG transformer reads, from shared memory, a reusable RDD corresponding to a DAG (which may be denoted as a third DAG) located in a first place in a first window.

[0158] The DAG located in the first place in the first window is a to-be-executed DAG located at the beginning of a queue in the first window, that is, a to-be-executed DAG that is in the first window and that is adjacent to the second window.

[0159] S302: The DAG transformer transforms the third DAG into a fourth DAG, and adds the fourth DAG to the second window for waiting for execution. The fourth DAG and the third DAG have a same computing result.

[0160] Specifically, the third DAG is transformed based on the reusable RDD (that is, the reusable RDD read in step S301) that is cached in the shared memory, and a part that is in the third DAG and that is the same as the reusable RDD is replaced with a part that depends on the reusable RDD, thereby obtaining the fourth DAG.

[0161] For example, an example in which the third DAG is the DAG 1 shown in FIG. 2 is used for description. The third DAG is that a map operation is performed on a text file (source data) to generate an RDD 1, and then a filter operation is performed to generate an RDD 2, where the RDD 1 is a reusable RDD that is cached in the shared memory. When reading the RDD 1, the DAG transformer transforms the DAG 1. The fourth DAG is that the filter operation is performed on the RDD 1 to generate the RDD 2, where the RDD 1 is the reusable RDD. The operation is available in the prior art, and details are not described herein.

[0162] It can be learned that the third DAG includes two transformation operations: the map operation and the filter operation, whereas the fourth DAG includes only the filter operation. Obviously, time for computing the fourth DAG is less than time for computing the third DAG.

[0163] S303: When a Spark kernel is idle, execute a DAG (which may be denoted as a fifth DAG) located in a first place in the second window.

[0164] The DAG located in the first place in the second window is a DAG located at the beginning of a queue in the second window, that is, a DAG for which transformation based a reusable RDD is currently completed first in the second window, and that waits for execution performed by the Spark kernel. For example, when the Spark kernel is idle, a SparkContext decomposes an RDD included in a to-be-executed DAG in the second window into stages, further decomposes the stages into tasks, and allocates the tasks to corresponding workers for execution.

[0165] S304: After the fifth DAG is executed, there is a vacancy in the second window again. The DAG transformer reads, from the shared memory, data of a reusable RDD corresponding to a DAG (which may be denoted as a sixth DAG) currently located in the first place in the first window, transforms the sixth DAG, and adds a post-transformation DAG to the end of the queue in the second window for waiting for execution performed by the Spark kernel.

[0166] For a specific operation, refer to step S301 to step S303. Details are not described herein again.

[0167] By analogy, when the Spark kernel is idle, DAGs in the second window are executed sequentially. Whenever there is a vacancy in the second window, a DAG located in the first place in the first window is read sequentially. After the DAG is transformed, a post-transformation DAG is added to the end of the queue in the second window to wait for execution.

[0168] In some examples, after executing a batch of DAGs, the Spark kernel may delete reusable RDDs corresponding to the batch of DAGs, to release space in the shared memory, cache a reusable RDD included in another DAG in the first window, and improve utilization of the shared memory. For example, by using m DAGs in the first window as a unit, after the Spark kernel completes executing DAGs generated by transforming the m DAGs, the Spark kernel may delete reusable RDDs corresponding to the m DAGs. Optionally, after the Spark kernel completes executing DAGs generated by transforming $n \times m$ DAGs (n is an integer greater than 2), the Spark kernel may alternatively delete reusable RDDs corresponding to the $n \times m$ DAGs.

[0169] It may be understood that, to implement the functions described in the foregoing method embodiments, the device such as a terminal includes a corresponding hardware structure and/or a software module for performing the functions. A person of ordinary skill in the art should easily be aware that, in combination with the examples described in the embodiments disclosed in this specification, units, algorithms, and steps may be implemented by hardware or a combination of hardware and computer software. Whether a function is performed by hardware or hardware driven by computer software depends on particular application programs and design constraints of the technical solutions. A person skilled in the art may use different methods to implement the described functions for each particular application program, but it should not be considered that the implementation goes beyond the scope of the embodiments of the present invention.

[0170] In the embodiments of this application, functional module division may be performed on the terminal and the like based on the foregoing method examples. For example, each functional module may be obtained through division based on a corresponding function, or two or more functions may be integrated into one processing module. The integrated module may be implemented in a form of hardware, or may be implemented in a form of a software functional module. It should be noted that, in this embodiment of the present invention, module division is exemplary, and is merely a logical function division. In actual implementation, another division manner may be used.

[0171] The foregoing descriptions about implementations allow a person skilled in the art to clearly understand that, for the purpose of convenient and brief description, division of the foregoing function modules is taken as an example for illustration. In actual application program, the foregoing functions can be allocated to different modules and implemented according to a requirement, that is, an inner structure of an apparatus is divided into different function modules to implement all or some of the functions described above. For a detailed working process of the foregoing system, apparatus, and unit, refer to a corresponding process in the foregoing method embodiments, and details are not described herein again.

[0172] Functional units in the embodiments of this application may be integrated into one processing unit, or each of the units may exist alone physically, or two or more units are integrated into one unit. The integrated unit may be implemented in a form of hardware, or may be implemented in a form of a software functional unit.

[0173] When the integrated unit is implemented in the form of a software functional unit and sold or used as an independent product, the integrated unit may be stored in a computer-readable storage medium. Based on such an understanding, the technical solutions of the embodiments of this application essentially, or the part contributing to the prior art, or all or some of the technical solutions may be implemented in the form of a software product. The software product is stored in a storage medium and includes several instructions for instructing a computer device (which may be a personal computer, a server, or a network device) to perform all or some of the steps of the methods described in the embodiments of this application. The foregoing storage medium includes: any medium that can store program code, such as a flash memory, a removable hard disk, a read-only memory, a random access memory, a magnetic disk, or an optical disc.

[0174] The foregoing descriptions are merely specific implementations of this application, but are not intended to limit the protection scope of this application. Any variation or replacement within the technical scope disclosed in this application shall fall within the protection scope of this application. Therefore, the protection scope of this application shall be subject to the protection scope of the claims.

Claims

1. A method for processing shared data, wherein the method is applied to a server cluster, and the method comprises:

receiving, by a first server, a first instruction for starting a first application program;
in response to the first instruction, starting, by the first server, a first Spark context for the first application program, to create a directed acyclic graph DAG of the first application program, and caching the DAG of the first application program in a first area of the first server;
receiving, by the first server, a second instruction for starting a second application program;
in response to the second instruction, starting, by the first server, a second Spark context for the second application program, to create a DAG of the second application program, and caching the DAG of the second application program in the first area of the first server;
reading, by the first server, m DAGs from the first area, wherein the m DAGs comprise the DAG of the first application program and the DAG of the second application program;
determining, by the first server, a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs; and
caching the to-be-cached shareable RDD in a main memory of a second server, wherein the shareable RDD is an RDD comprised in at least two DAGs of the m DAGs.

2. The method for processing shared data according to claim 1, wherein the determining, by the first server, a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs; and caching the to-be-cached shareable RDD in a main memory of a second server comprises:

   computing a quantity of shareable RDDs comprised in each of the m DAGs;
   determining a first DAG and a shareable RDD comprised in the first DAG, wherein the first DAG is a DAG with a largest quantity of shareable RDDs in the m DAGs; and
   determining, based on remaining space in the main memory of the second server, a to-be-cached shareable RDD comprised in the first DAG, and caching the to-be-cached shareable RDD comprised in the first DAG.

3. The method for processing shared data according to claim 2, wherein the determining, based on remaining space in the main memory of the second server, a to-be-cached shareable RDD comprised in the first DAG comprises:

   if the remaining space in the main memory of the second server is greater than or equal to a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining all the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
   if the remaining space in the main memory of the second server is less than a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs.

4. The method for processing shared data according to claim 3, wherein the determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs comprises:

   determining, based on a magnitude, a quantity of reuse times, or computing duration of each shareable RDD comprised in the first DAG, or a magnitude of a dependent parent RDD of each shareable RDD comprised in the first DAG, some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
   determining a shareable RDD that is obtained through a query in the first DAG as the to-be-cached shareable RDD.

5. The method for processing shared data according to claim 2, wherein the determining, based on remaining space in the main memory of the second server, a to-be-cached shareable RDD comprised in the first DAG further comprises:

   if the remaining space in the main memory of the second server is greater than or equal to a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining all the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
   if the remaining space in the main memory of the second server is less than a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining a to-be-replaced RDD comprised in the main memory of the second server, and determining, based on the remaining space in the main memory of the second server and a magnitude of the to-be-replaced RDD, the to-be-cached shareable RDD comprised in the first DAG.

6. The method for processing shared data according to claim 5, wherein the determining a to-be-replaced RDD comprised in the main memory of the second server comprises:
   determining the to-be-replaced RDD based on a magnitude, a quantity of reuse times, or computing duration of each shareable RDD cached in the main memory of the second server, or based on a magnitude of a dependent parent RDD of each shareable RDD cached in the main memory of the second server.

7. The method for processing shared data according to claim 5, wherein the determining, based on the remaining space in the main memory of the second server and a magnitude of the to-be-replaced RDD, the to-be-cached shareable RDD comprised in the first DAG comprises:

   if a sum of the remaining space in the main memory of the second server and the magnitude of the to-be-replaced RDD is greater than or equal to the sum of the magnitudes of all the shareable RDDs comprised in the first DAG, determining all the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
   if a sum of the remaining space in the main memory of the second server and the magnitude of the to-be-replaced RDD is less than the sum of the magnitudes of all the shareable RDDs comprised in the first DAG, determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs.

8. The method for processing shared data according to claim 7, wherein the determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs comprises:

determining, based on a magnitude, a quantity of reuse times, or computing duration of each shareable RDD comprised in the first DAG, or a magnitude of a dependent parent RDD of each shareable RDD comprised in the first DAG, some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or determining a shareable RDD that is obtained through a query in the first DAG as the to-be-cached shareable RDD.

9. The method for processing shared data according to claim 5, wherein the caching the to-be-cached shareable RDD comprised in the first DAG comprises:
removing the to-be-replaced RDD from the main memory of the second server, and caching the to-be-cached shareable RDD comprised in the first DAG.

10. The method for processing shared data according to any one of claims 1 to 9, further comprising:

determining a third DAG in the m DAGs; and
reading a shareable RDD comprised in the third DAG from the main memory of the second server, and transforming the third DAG into a fourth DAG based on the shareable RDD comprised in the third DAG, wherein an execution result the fourth DAG and an execution result of the third DAG are the same, and execution duration of the fourth DAG is less than execution duration of the third DAG.

11. The method for processing shared data according to claim 10, wherein the main memory of the second server is a main memory of the first server, or the main memory of the second server, or a sum of all or some of a main memory of the first server and all or some of the main memory of the second server.

12. A server, comprising a processor, a memory, and a communications interface, wherein the memory and the communications interface are coupled to the processor, the memory is configured to store computer program code, the computer program code comprises computer instructions, and when the processor reads the computer instructions from the memory, the server is enabled to perform the following operations:

receiving a first instruction for starting a first application program; starting a first Spark context for the first application program, to create a directed acyclic graph DAG of the first application program, and caching the DAG of the first application program in a first area of the first server;
receiving a second instruction for starting a second application program; starting a second Spark context for the second application program, to create a DAG of the second application program, and caching the DAG of the second application program in the first area of the first server;
reading m DAGs from the first area, wherein the m DAGs comprise the DAG of the first application program and the DAG of the second application program;
determining a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs; and caching the to-be-cached shareable RDD in a main memory of a second server, wherein the shareable RDD is an RDD comprised in at least two DAGs of the m DAGs.

13. The server according to claim 12, wherein the determining a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs; and caching the to-be-cached shareable RDD in a main memory of a second server comprises:

computing a quantity of shareable RDDs comprised in each of the m DAGs;
determining a first DAG and a shareable RDD comprised in the first DAG, wherein the first DAG is a DAG with a largest quantity of shareable RDDs in the m DAGs; and
determining, based on remaining space in the main memory of the second server, a to-be-cached shareable RDD comprised in the first DAG, and caching the to-be-cached shareable RDD comprised in the first DAG.

14. The server according to claim 13, wherein the determining, based on remaining space in the main memory of the second server, a to-be-cached shareable RDD comprised in the first DAG comprises:

if the remaining space in the main memory of the second server is greater than or equal to a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining all the shareable RDDs comprised in the first

DAG as the to-be-cached shareable RDDs; or
if the remaining space in the main memory of the second server is less than a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs.

15. The server according to claim 14, wherein the determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs comprises:

determining, based on a magnitude, a quantity of reuse times, or computing duration of each shareable RDD comprised in the first DAG, or a magnitude of a dependent parent RDD of each shareable RDD comprised in the first DAG, some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
determining a shareable RDD that is obtained through a query in the first DAG as the to-be-cached shareable RDD.

16. The server according to claim 13, wherein the determining a to-be-cached shareable resilient distributed dataset RDD based on the m DAGs; and caching the to-be-cached shareable RDD in the main memory of a second server comprises:

if the remaining space in the main memory of the second server is greater than or equal to a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining all the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
if the remaining space in the main memory of the second server is less than a sum of magnitudes of all shareable RDDs comprised in the first DAG, determining a to-be-replaced RDD comprised in the main memory of the second server, and determining, based on the remaining space in the main memory of the second server and a magnitude of the to-be-replaced RDD, the to-be-cached shareable RDD comprised in the first DAG.

17. The server according to claim 16, wherein the determining a to-be-replaced RDD comprised in the main memory of the second server comprises:
determining the to-be-replaced RDD based on a magnitude, a quantity of reuse times, or computing duration of each shareable RDD cached in the main memory of the second server, or based on a magnitude of a dependent parent RDD of each shareable RDD cached in the main memory of the second server.

18. The server according to claim 16, wherein the determining, based on the remaining space in the main memory of the second server and a magnitude of the to-be-replaced RDD, the to-be-cached shareable RDD comprised in the first DAG comprises:

if a sum of the remaining space in the main memory of the second server and the magnitude of the to-be-replaced RDD is greater than or equal to the sum of the magnitudes of all the shareable RDDs comprised in the first DAG, determining all the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs; or
if a sum of the remaining space in the main memory of the second server and the magnitude of the to-be-replaced RDD is less than the sum of the magnitudes of all the shareable RDDs comprised in the first DAG, determining some of the shareable RDDs comprised in the first DAG as the to-be-cached shareable RDDs.

19. The server according to claim 16, wherein the caching the to-be-cached shareable RDD comprised in the first DAG comprises:
removing the to-be-replaced RDD from the main memory of the second server, and caching the to-be-cached shareable RDD comprised in the first DAG.

20. The server according to any one of claims 12 to 19, further performing the following operations:

determining a third DAG in the m DAGs; and
reading a shareable RDD comprised in the third DAG from the main memory of the second server, and transforming the third DAG into a fourth DAG based on the shareable RDD comprised in the third DAG, wherein an execution result the fourth DAG and an execution result of the third DAG are the same.

21. The server according to claim 20, wherein the main memory of the second server is a main memory of the first server, or the main memory of the second server, or a sum of all or some of a main memory of the first server and all or some of the main memory of the second server.

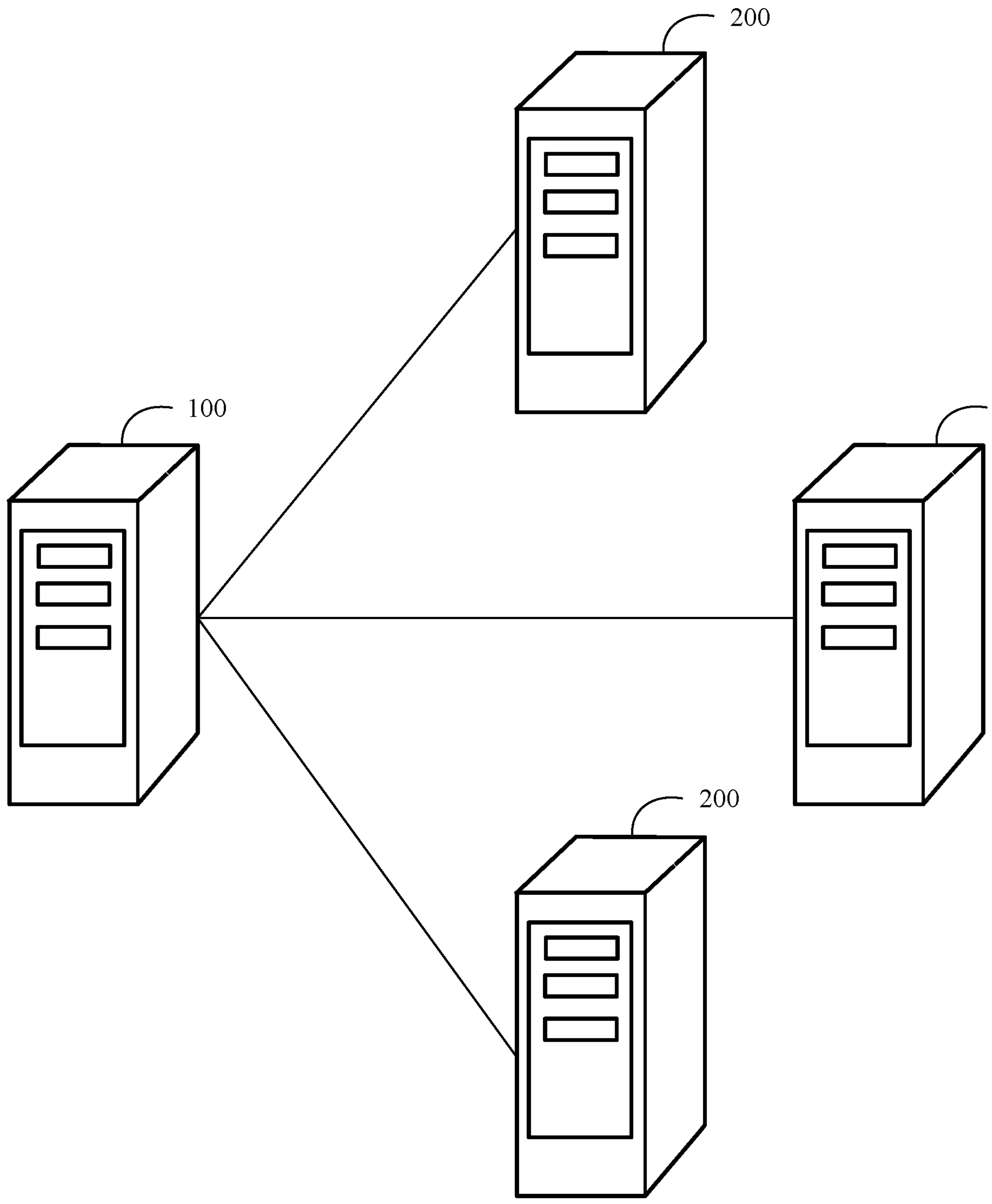

FIG. 1

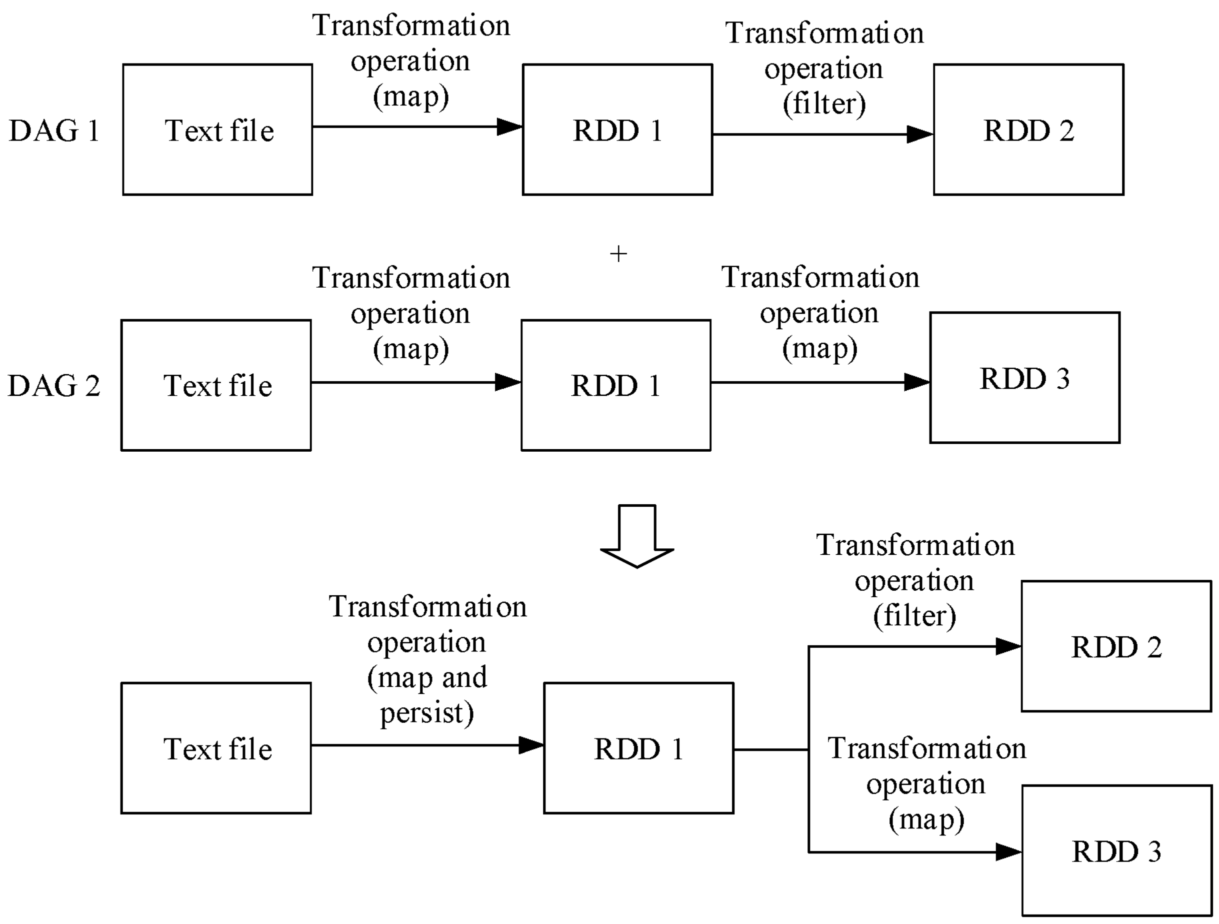
DAG 1
Text file
Transformation operation (map)
RDD 1
Transformation operation (filter)
RDD 2
+
DAG 2
Text file
Transformation operation (map)
RDD 1
Transformation operation (map)
RDD 3
Text file
Transformation operation (map and persist)
RDD 1
Transformation operation (filter)
RDD 2
Transformation operation (map)
RDD 3

FIG. 2

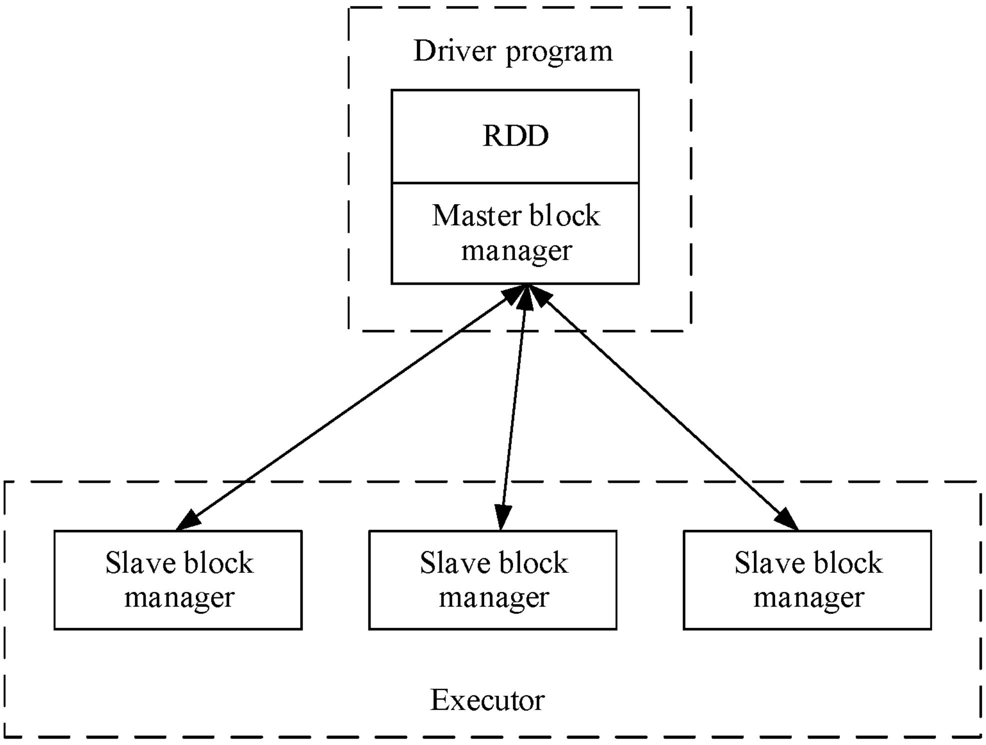
Driver program
RDD
Master block manager
Slave block manager
Slave block manager
Slave block manager
Executor

FIG. 3

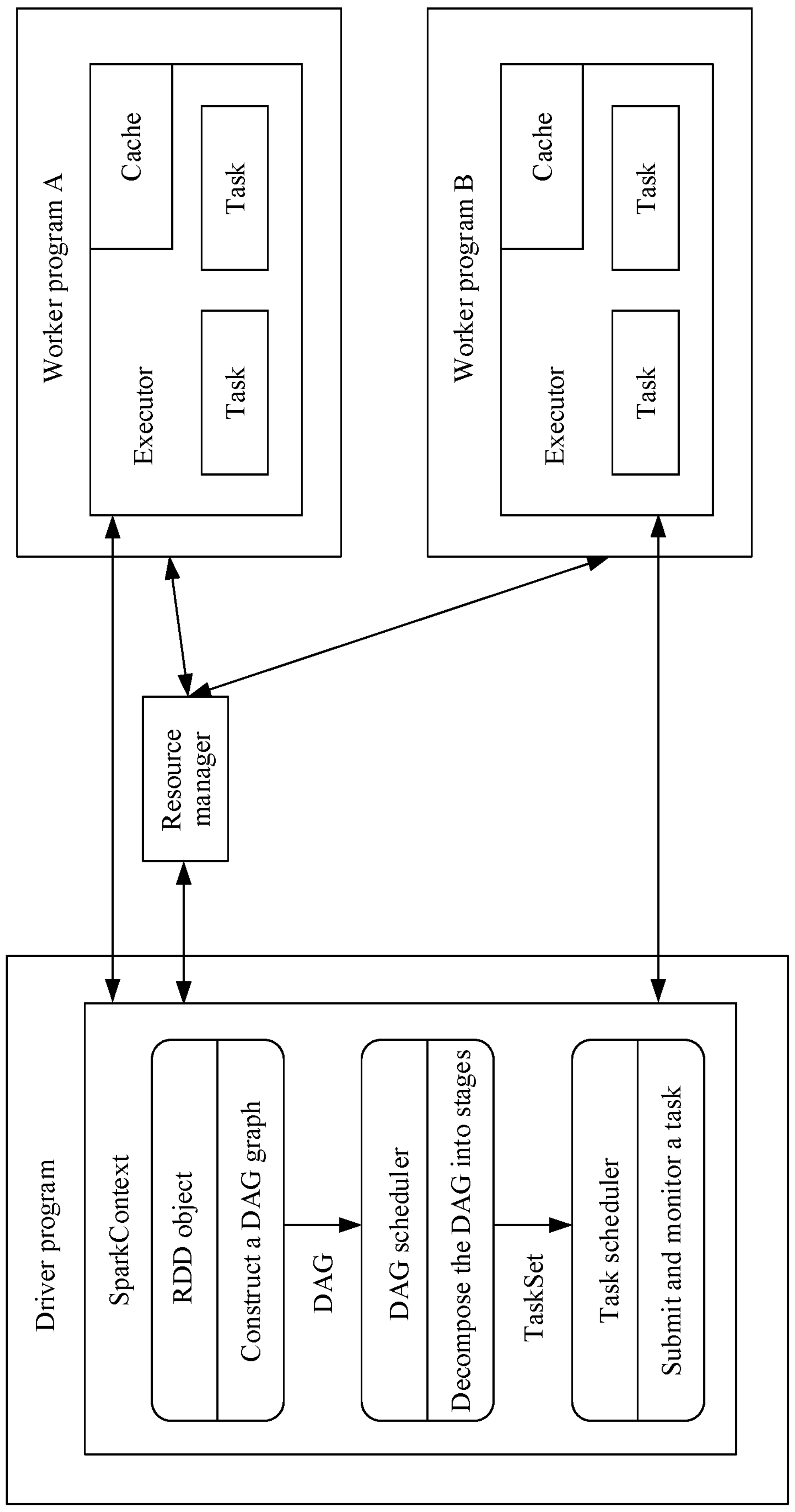
Worker program A
Cache
Executor
Task
Task
Worker program B
Cache
Executor
Task
Task
Resource manager
Driver program
SparkContext
RDD object
Construct a DAG graph
DAG
DAG scheduler
Decompose the DAG into stages
TaskSet
Task scheduler
Submit and monitor a task

FIG. 4

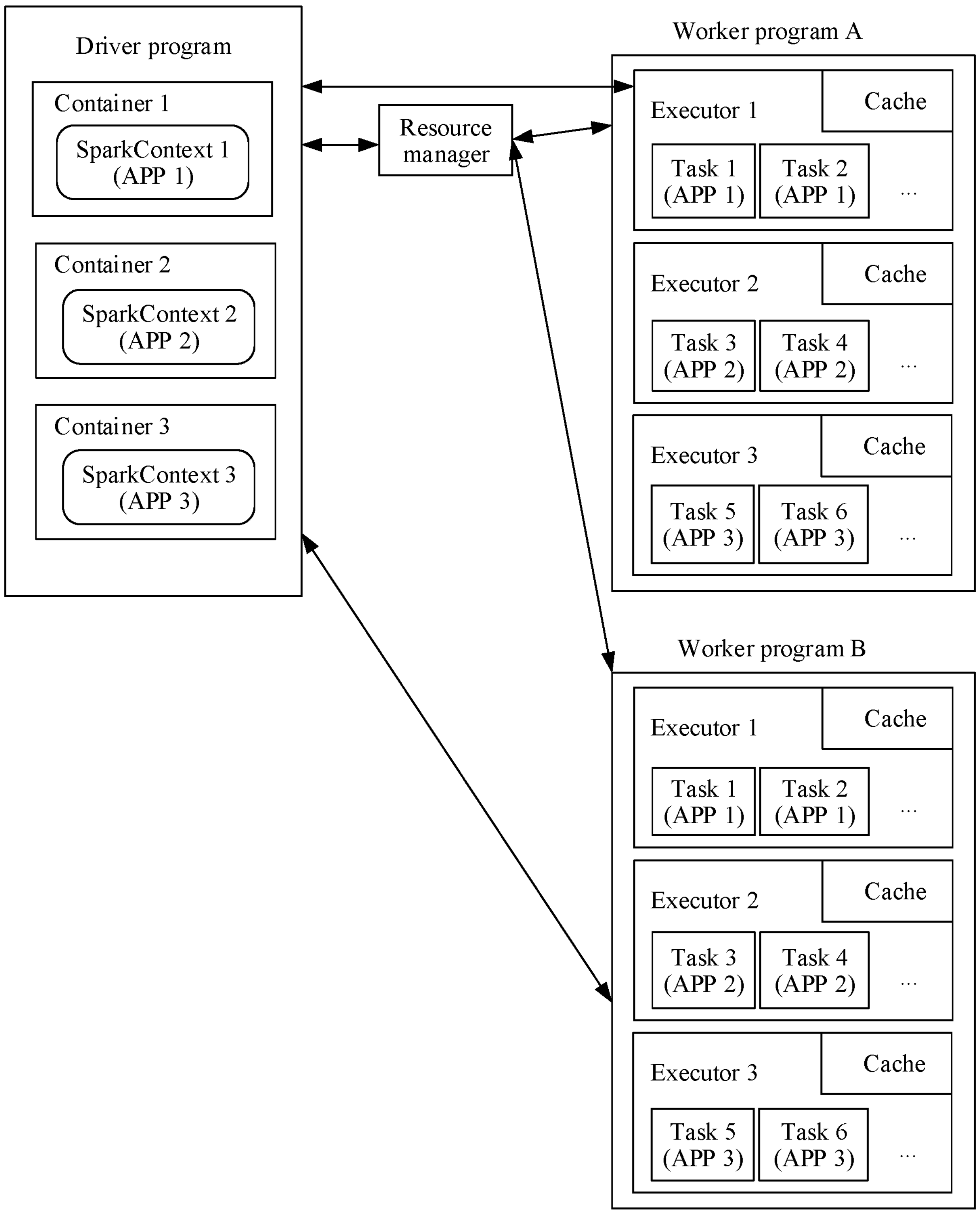
Driver program
Container 1
SparkContext 1 (APP 1)
Container 2
SparkContext 2 (APP 2)
Container 3
SparkContext 3 (APP 3)
Resource manager
Worker program A
Executor 1
Cache
Task 1 (APP 1)
Task 2 (APP 1)
...
Executor 2
Cache
Task 3 (APP 2)
Task 4 (APP 2)
...
Executor 3
Cache
Task 5 (APP 3)
Task 6 (APP 3)
...
Worker program B
Executor 1
Cache
Task 1 (APP 1)
Task 2 (APP 1)
...
Executor 2
Cache
Task 3 (APP 2)
Task 4 (APP 2)
...
Executor 3
Cache
Task 5 (APP 3)
Task 6 (APP 3)
...

FIG. 5A

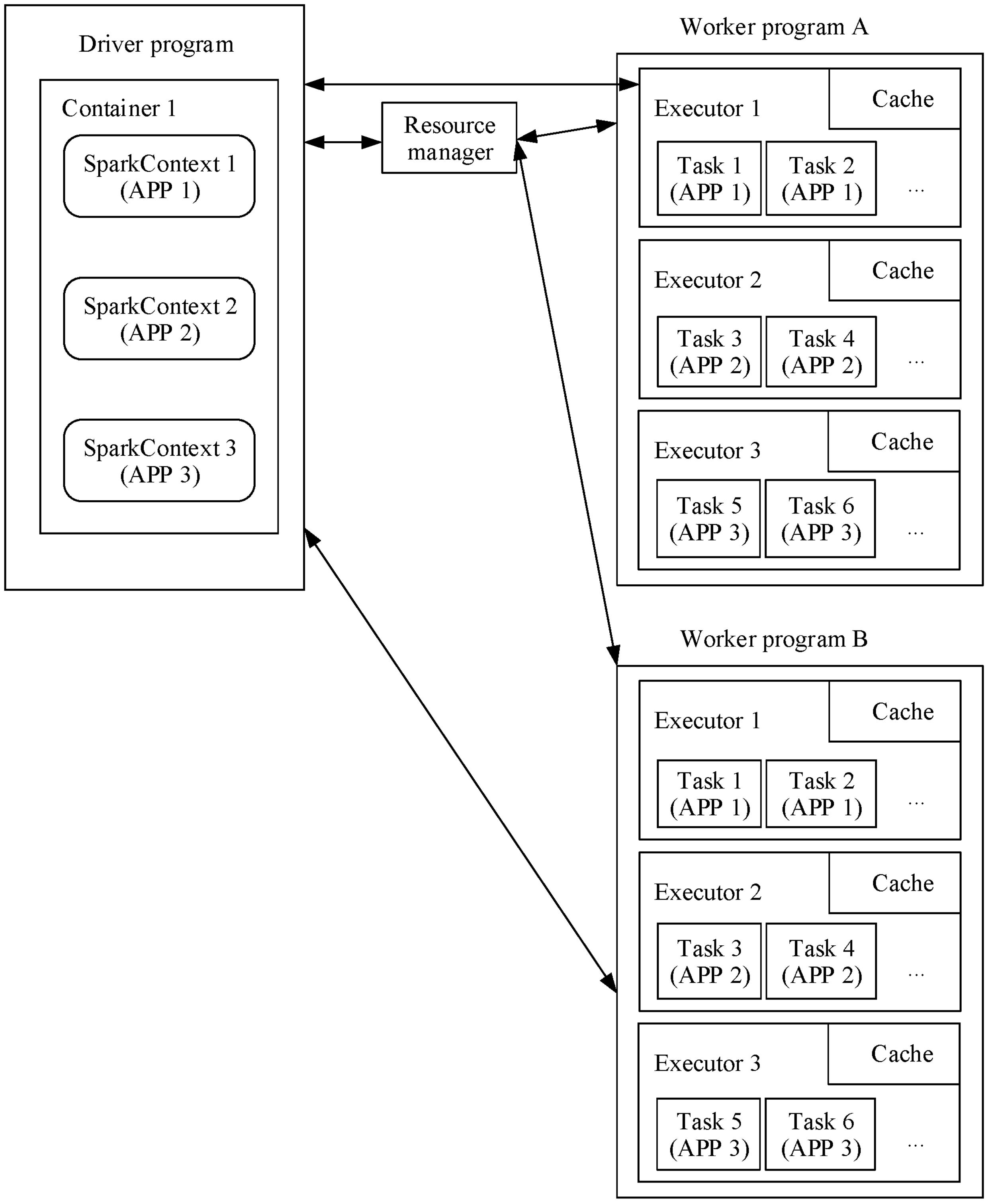
Driver program
Container 1
SparkContext 1 (APP 1)
SparkContext 2 (APP 2)
SparkContext 3 (APP 3)
Resource manager
Worker program A
Executor 1
Cache
Task 1 (APP 1)
Task 2 (APP 1)
...
Executor 2
Cache
Task 3 (APP 2)
Task 4 (APP 2)
...
Executor 3
Cache
Task 5 (APP 3)
Task 6 (APP 3)
...
Worker program B
Executor 1
Cache
Task 1 (APP 1)
Task 2 (APP 1)
...
Executor 2
Cache
Task 3 (APP 2)
Task 4 (APP 2)
...
Executor 3
Cache
Task 5 (APP 3)
Task 6 (APP 3)
...

FIG. 5B

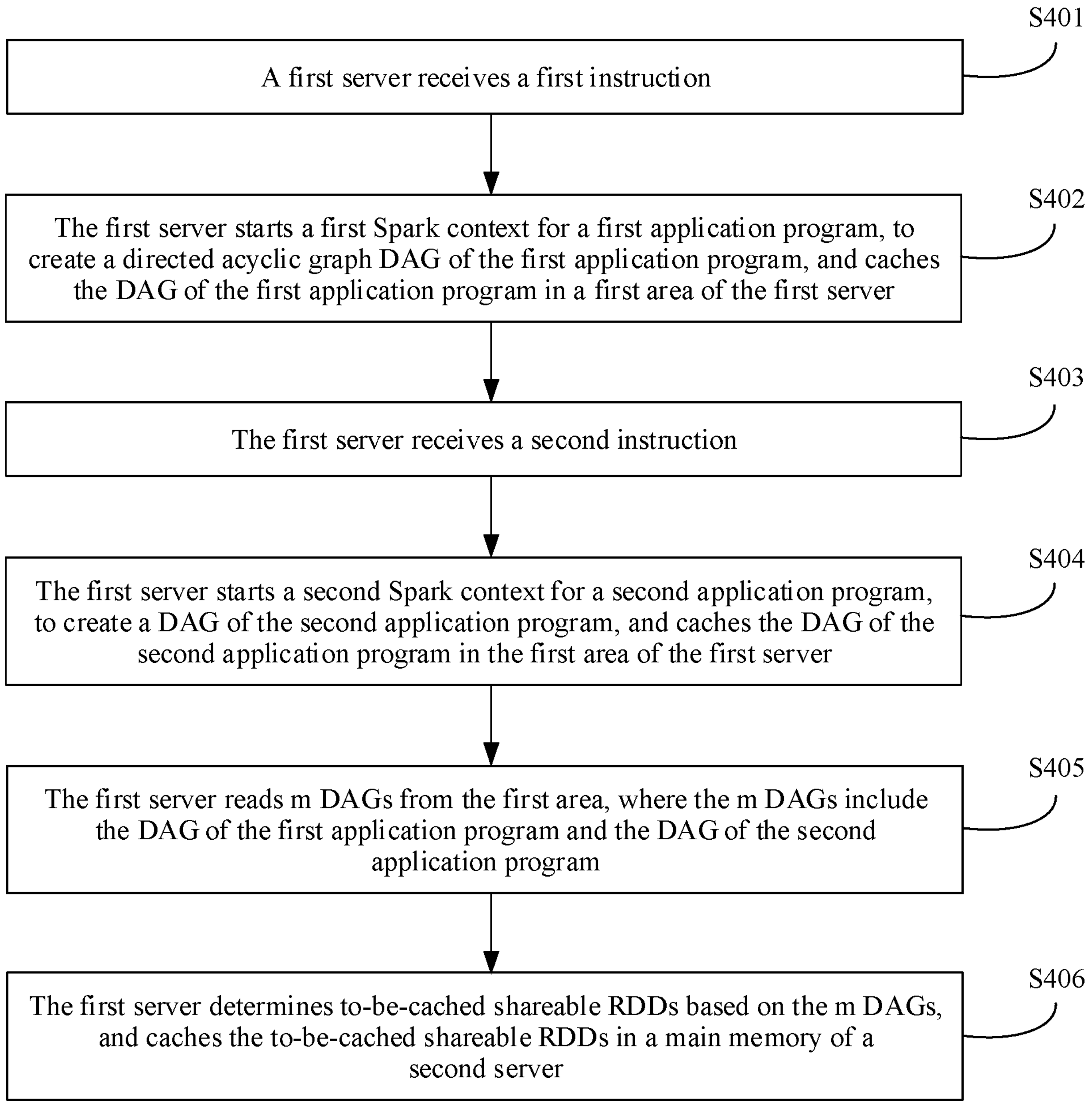
S401
A first server receives a first instruction
S402
The first server starts a first Spark context for a first application program, to create a directed acyclic graph DAG of the first application program, and caches the DAG of the first application program in a first area of the first server
S403
The first server receives a second instruction
S404
The first server starts a second Spark context for a second application program, to create a DAG of the second application program, and caches the DAG of the second application program in the first area of the first server
S405
The first server reads m DAGs from the first area, where the m DAGs include the DAG of the first application program and the DAG of the second application program
S406
The first server determines to-be-cached shareable RDDs based on the m DAGs, and caches the to-be-cached shareable RDDs in a main memory of a second server

FIG. 5C

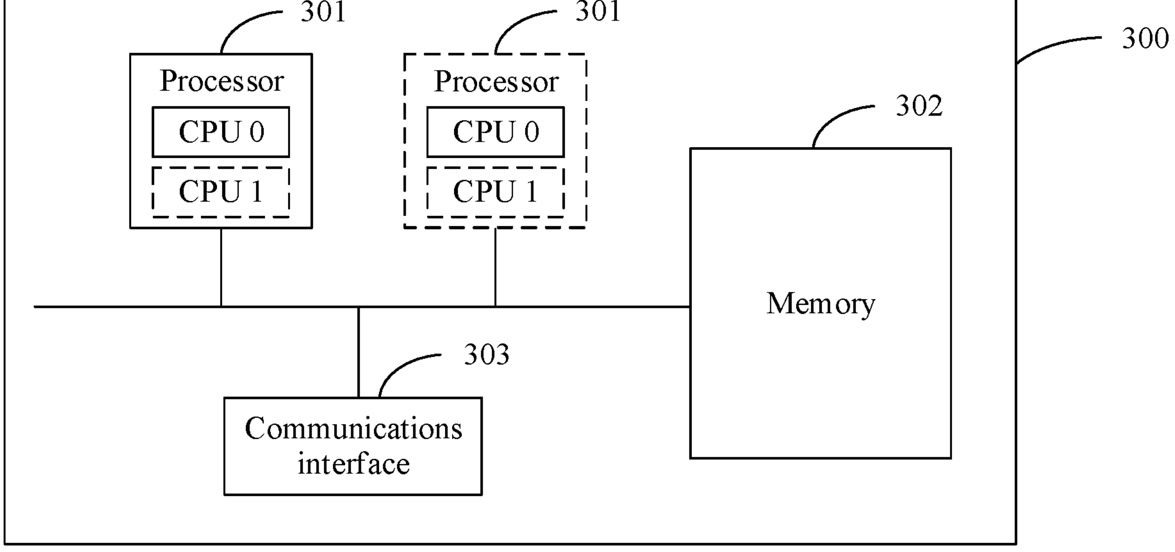
301
301
300
Processor
CPU 0
CPU 1
Processor
CPU 0
CPU 1
302
Memory
303
Communications interface

FIG. 6A

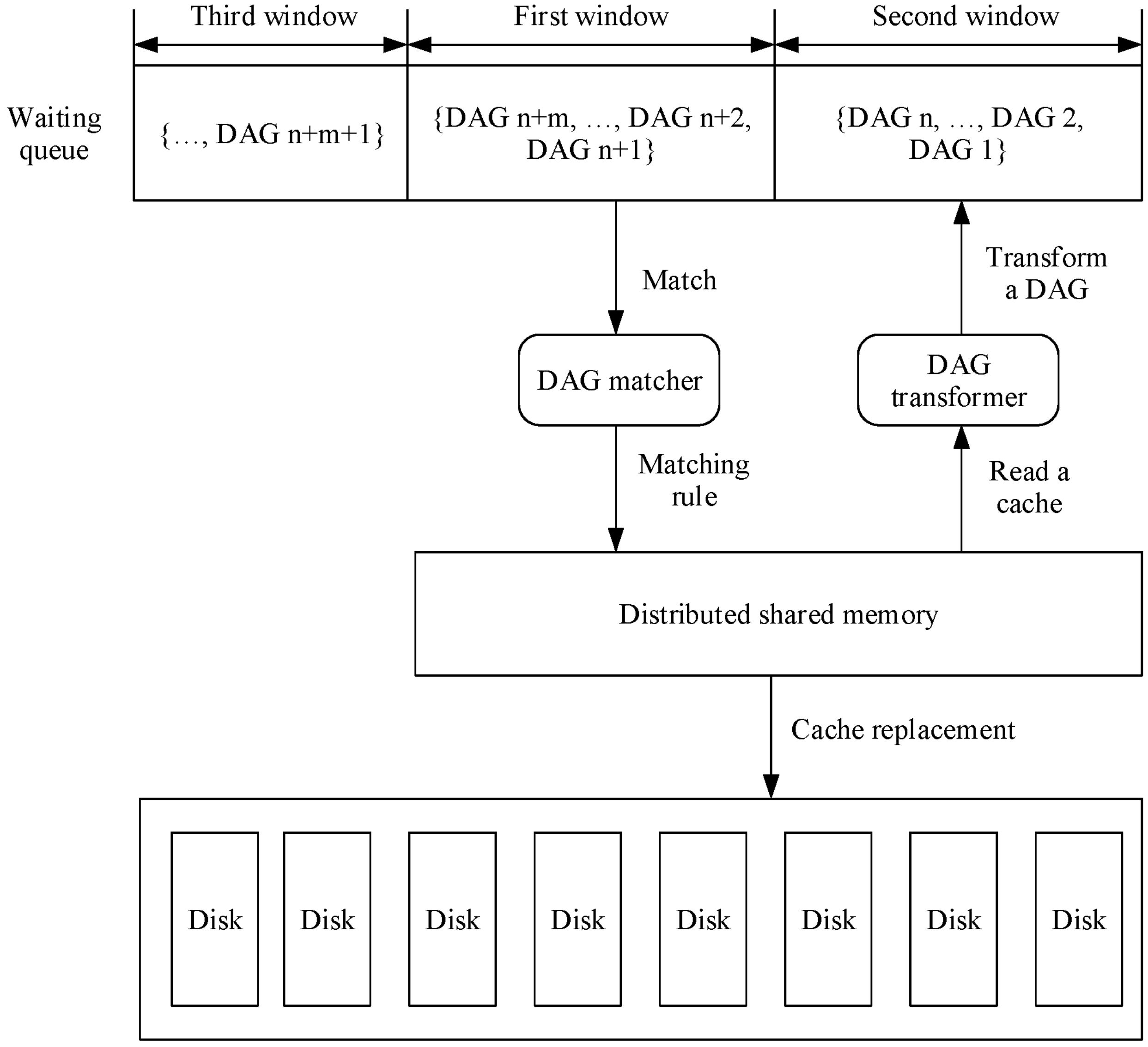
Third window
First window
Second window
Waiting queue
{…, DAG n+m+1}
{DAG n+m, …, DAG n+2, DAG n+1}
{DAG n, …, DAG 2, DAG 1}
Match
Transform a DAG
DAG matcher
DAG transformer
Matching rule
Read a cache
Distributed shared memory
Cache replacement
Disk
Disk
Disk
Disk
Disk
Disk
Disk
Disk

FIG. 6B

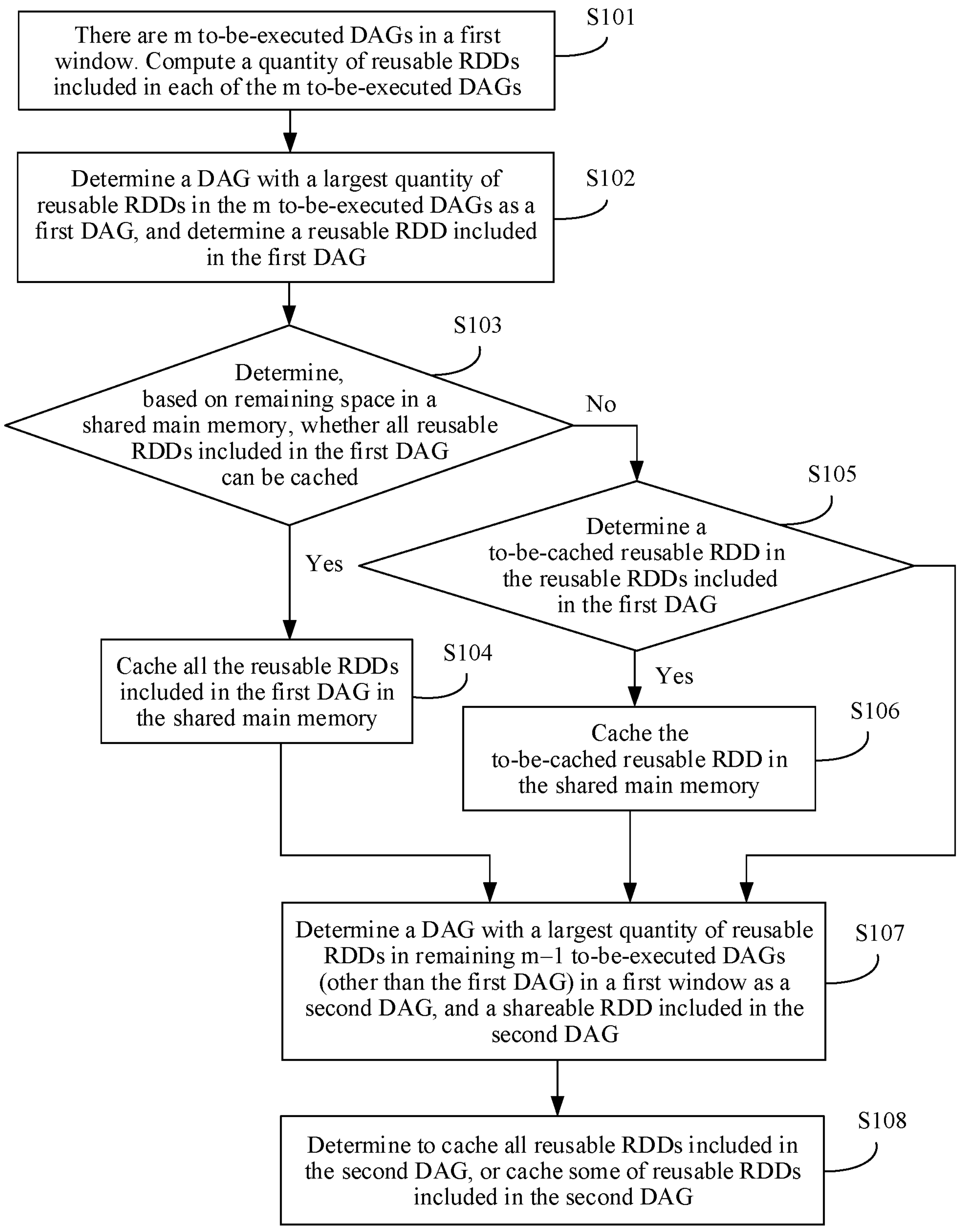
There are m to-be-executed DAGs in a first window. Compute a quantity of reusable RDDs included in each of the m to-be-executed DAGs
S101
Determine a DAG with a largest quantity of reusable RDDs in the m to-be-executed DAGs as a first DAG, and determine a reusable RDD included in the first DAG
S102
Determine, based on remaining space in a shared main memory, whether all reusable RDDs included in the first DAG can be cached
S103
No
Yes
Cache all the reusable RDDs included in the first DAG in the shared main memory
S104
Determine a to-be-cached reusable RDD in the reusable RDDs included in the first DAG
S105
Yes
Cache the to-be-cached reusable RDD in the shared main memory
S106
Determine a DAG with a largest quantity of reusable RDDs in remaining m−1 to-be-executed DAGs (other than the first DAG) in a first window as a second DAG, and a shareable RDD included in the second DAG
S107
Determine to cache all reusable RDDs included in the second DAG, or cache some of reusable RDDs included in the second DAG
S108

FIG. 7

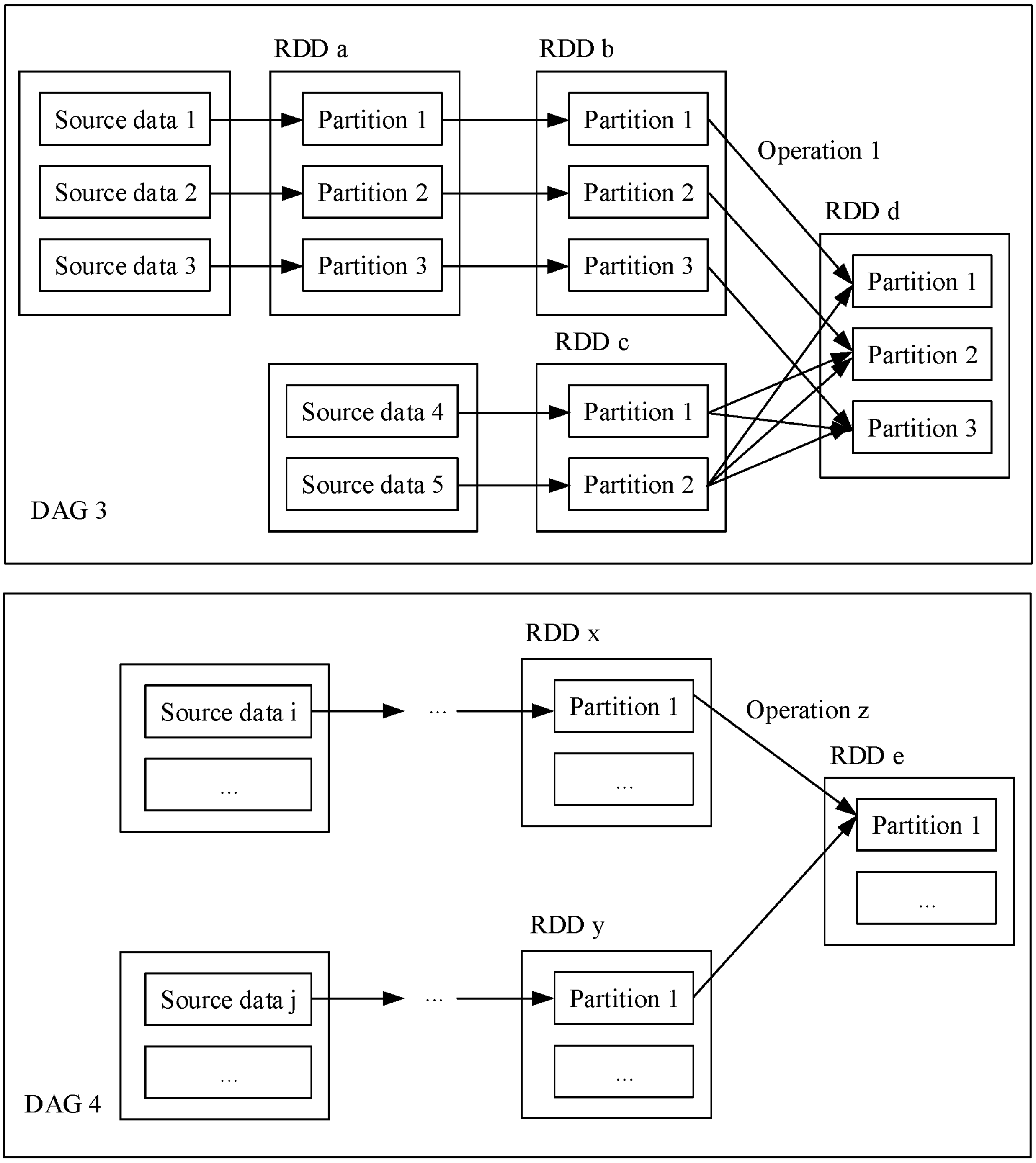
RDD a
RDD b
Source data 1
Source data 2
Source data 3
Partition 1
Partition 2
Partition 3
Operation 1
RDD d
RDD c
Source data 4
Source data 5
DAG 3
RDD x
Source data i
...
Operation z
RDD e
RDD y
Source data j
DAG 4

FIG. 8

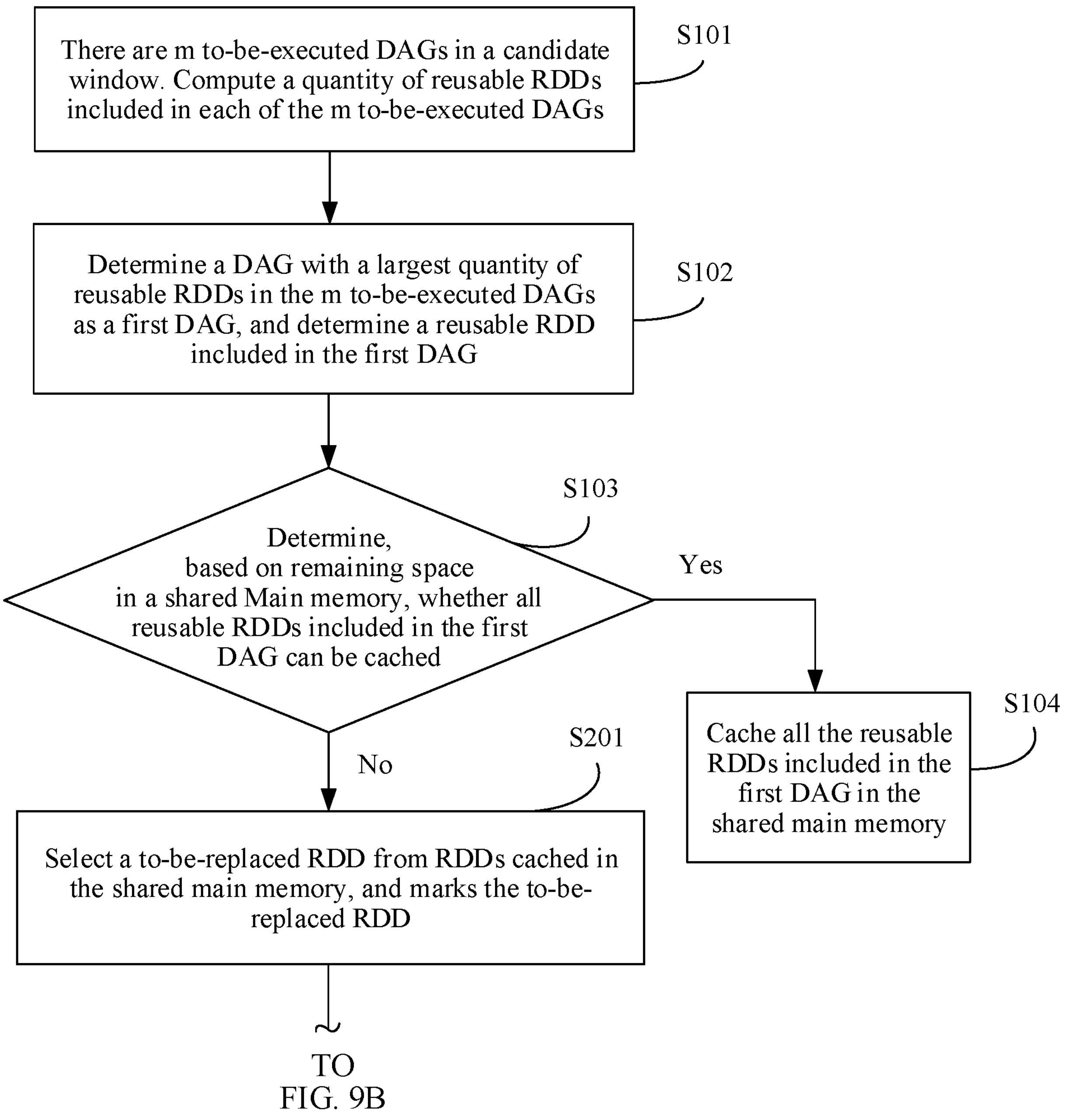
There are m to-be-executed DAGs in a candidate window. Compute a quantity of reusable RDDs included in each of the m to-be-executed DAGs
S101
Determine a DAG with a largest quantity of reusable RDDs in the m to-be-executed DAGs as a first DAG, and determine a reusable RDD included in the first DAG
S102
Determine, based on remaining space in a shared Main memory, whether all reusable RDDs included in the first DAG can be cached
S103
Yes
Cache all the reusable RDDs included in the first DAG in the shared main memory
S104
No
S201
Select a to-be-replaced RDD from RDDs cached in the shared main memory, and marks the to-be-replaced RDD
TO FIG. 9B

FIG. 9A

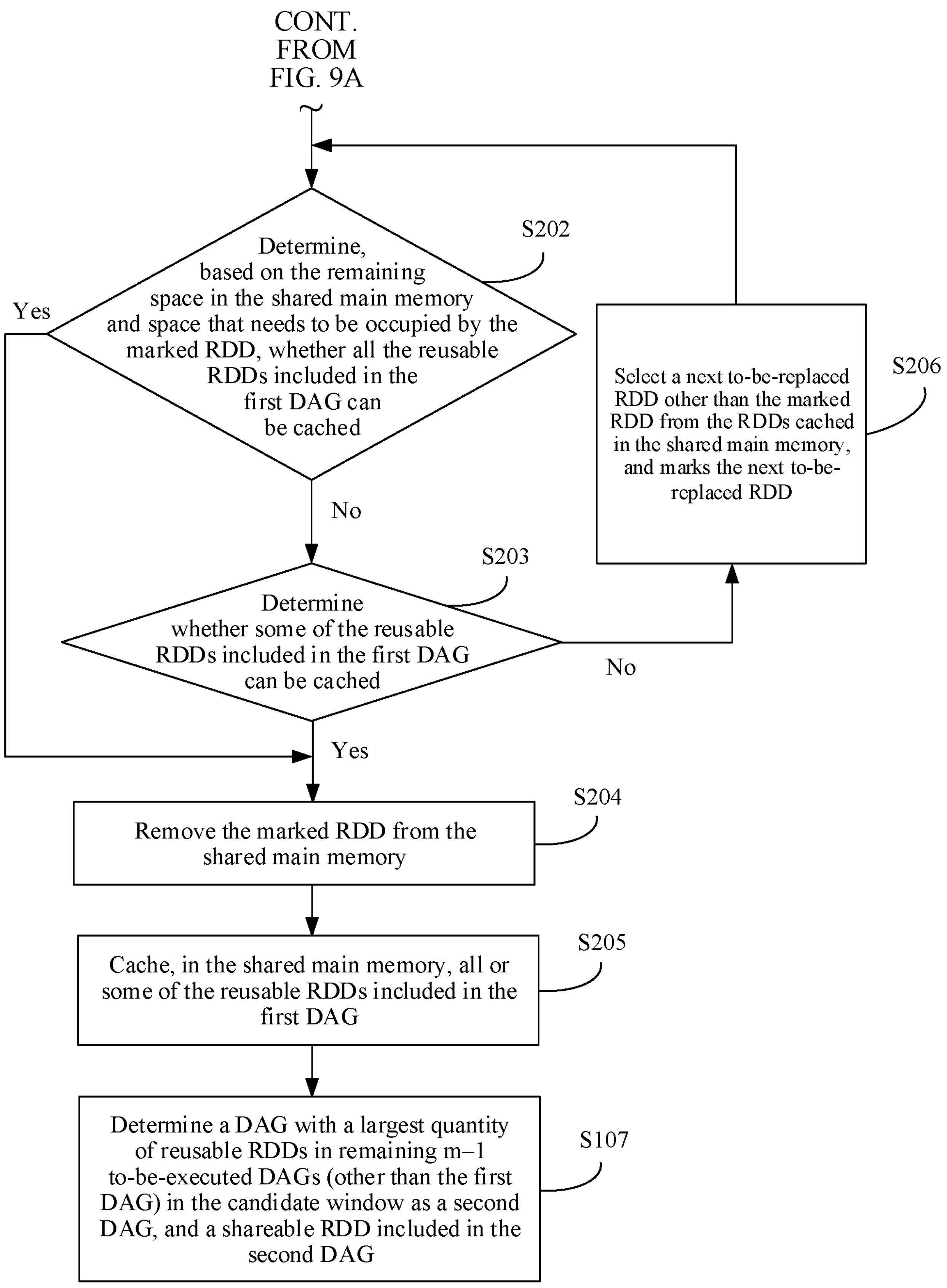
CONT. FROM FIG. 9A
S202
Determine, based on the remaining space in the shared main memory and space that needs to be occupied by the marked RDD, whether all the reusable RDDs included in the first DAG can be cached
Yes
No
S203
Determine whether some of the reusable RDDs included in the first DAG can be cached
No
Yes
S206
Select a next to-be-replaced RDD other than the marked RDD from the RDDs cached in the shared main memory, and marks the next to-be-replaced RDD
S204
Remove the marked RDD from the shared main memory
S205
Cache, in the shared main memory, all or some of the reusable RDDs included in the first DAG
S107
Determine a DAG with a largest quantity of reusable RDDs in remaining m−1 to-be-executed DAGs (other than the first DAG) in the candidate window as a second DAG, and a shareable RDD included in the second DAG

FIG. 9B

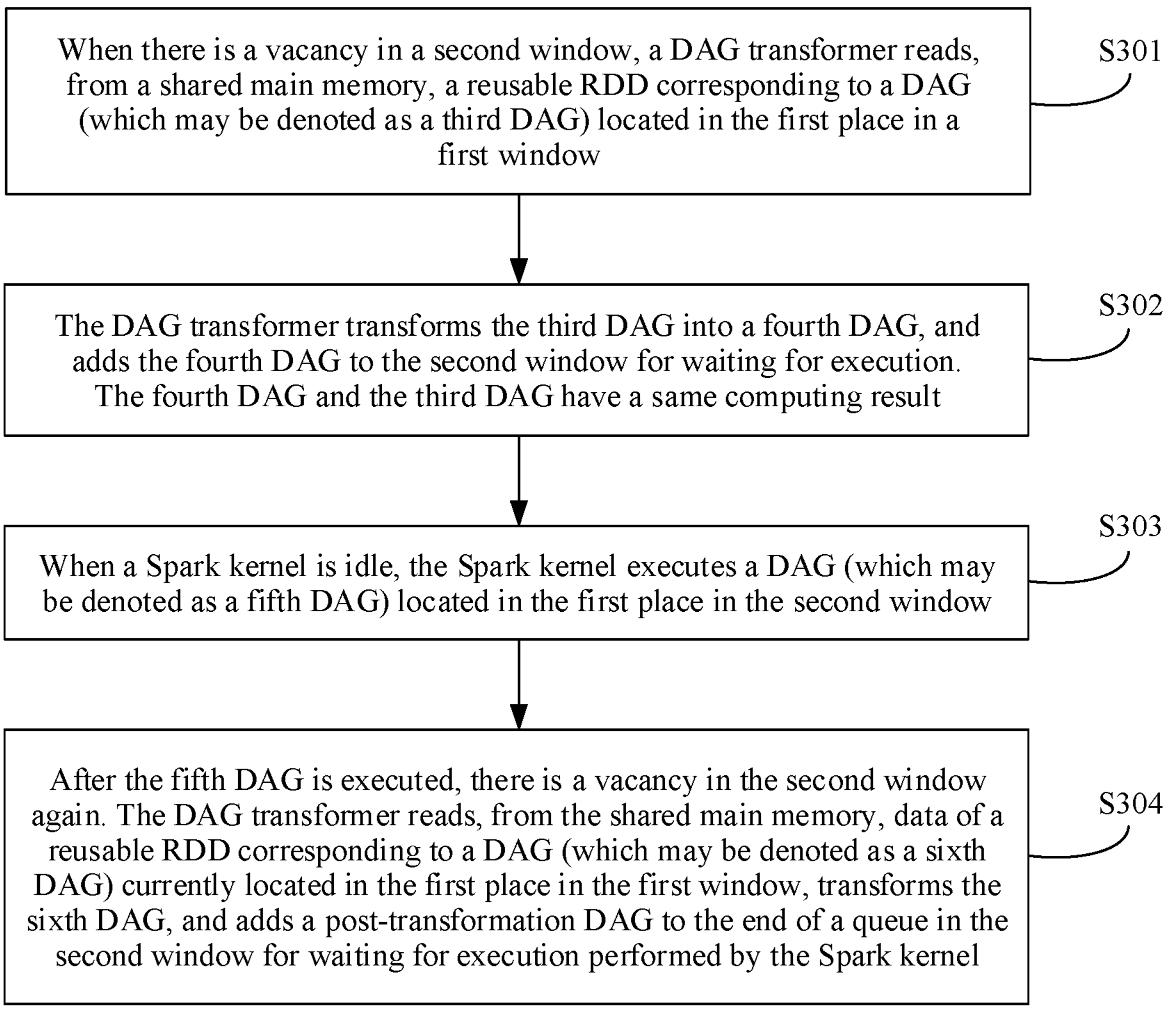
When there is a vacancy in a second window, a DAG transformer reads, from a shared main memory, a reusable RDD corresponding to a DAG (which may be denoted as a third DAG) located in the first place in a first window
S301
The DAG transformer transforms the third DAG into a fourth DAG, and adds the fourth DAG to the second window for waiting for execution. The fourth DAG and the third DAG have a same computing result
S302
When a Spark kernel is idle, the Spark kernel executes a DAG (which may be denoted as a fifth DAG) located in the first place in the second window
S303
After the fifth DAG is executed, there is a vacancy in the second window again. The DAG transformer reads, from the shared main memory, data of a reusable RDD corresponding to a DAG (which may be denoted as a sixth DAG) currently located in the first place in the first window, transforms the sixth DAG, and adds a post-transformation DAG to the end of a queue in the second window for waiting for execution performed by the Spark kernel
S304

FIG. 10

INTERNATIONAL SEARCH REPORT

International application No.
PCT/CN2019/122523

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/54(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

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

WPI, EPODOC, CNPAT, CNKI: 共享, 应用程序, 服务器, 启动, 有向无环图, 缓存, 弹性分布式数据集, share, application, server, start, DAG, SPARK, RDD

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
PX	CN 109800092 A (HUAWEI TECHNOLOGIES CO., LTD.) 24 May 2019 (2019-05-24) claims 1-21	1-21
A	CN 103631730 A (RESEARCH INSTITUTE OF TSINGHUA UNIVERSITY IN SHENZHEN) 12 March 2014 (2014-03-12) claim 1, and description, paragraph [0002]	1-21
A	CN 106339458 A (HUAWEI TECHNOLOGIES CO., LTD.) 18 January 2017 (2017-01-18) entire document	1-21
A	CN 107612886 A (UNIVERSITY OF CHINESE ACADEMY OF SCIENCES) 19 January 2018 (2018-01-19) entire document	1-21
A	CN 107870763 A (SHENZHEN HCF TECHNOLOGY CO., LTD.) 03 April 2018 (2018-04-03) entire document	1-21
A	US 9880823 B1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 30 January 2018 (2018-01-30) entire document	1-21

☐ 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 **17 February 2020**	Date of mailing of the international search report **26 February 2020**
Name and mailing address of the ISA/CN **China National Intellectual Property Administration (ISA/CN)** **No. 6, Xitucheng Road, Jimenqiao Haidian District, Beijing 100088** **China**	Authorized officer
Facsimile No. **(86-10)62019451**	Telephone No.

Form PCT/ISA/210 (second sheet) (January 2015)

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/CN2019/122523

Patent document cited in search report			Publication date (day/month/year)	Patent family member(s)			Publication date (day/month/year)
CN	109800092	A	24 May 2019	None			
CN	103631730	A	12 March 2014	CN	103631730	B	27 April 2016
CN	106339458	A	18 January 2017	None			
CN	107612886	A	19 January 2018	None			
CN	107870763	A	03 April 2018	None			
US	9880823	B1	30 January 2018	None			

Form PCT/ISA/210 (patent family annex) (January 2015)

REFERENCES CITED IN THE DESCRIPTION

Patent documents cited in the description

- CN 201811545173 **[0001]**