

ACTION-BASED GRAPH FRAMEWORK AND QUERY METHOD

Technical Field

5 The present invention relates, in general terms, to a network for storing and querying data represented as a graph. More particularly, the present invention relates to a network in which a graph is separated into subgraphs that can be queried in parallel.

10 Background

This background description is provided for the purpose of generally presenting the context of the disclosure. Contents of this background section are neither expressly nor impliedly admitted as prior art against the present
15 disclosure.

Many graph storage schemes have been proposed for storing data as a graph that can be queried. Such schemes include Janus Graph, Nebula Graph and TigerGraph. JanusGraph is a graph database engine focused on compact graph
20 serialization, rich graph data modelling, and efficient query execution. JanusGraph implements robust, modular interfaces for data persistence, data indexing, and client access.

Although JanusGraph implements rich external adapters and flexible
25 frameworks, it also has many drawback. At the storage level, graph data is not native, and it is transformed into row-oriented or column-oriented tables. This means that schema transformation and mapping from graph to table is required, which leads to increased storage overhead. Where external storage is used in the storage backend and that external storage has its own query
30 language, querying the graph will take longer and the latency of the request will increase, because JanusGraph needs to parse the query syntax twice. In addition, at the storage level, JanusGraph completely separates the connection between entities in the graph and indexes such as property indexes, label/type indexes, and so on. The adjacency access of the graph may need to run on
35 multiple nodes, increasing the overheads.

In Nebula Graph, graph data is stored in a storage service designed to be a shared-nothing distributed architecture. Each storage host has multiple local key/value stores as the physical data storage. A quorum consensus protocol is built on top of the key/value stores. Together, they provide a distributed key/value store. On top of this distributed key/value store, a graph semantic layer is provided. It translates the graph operations into key/value operations. The graph data (vertices and edges) are hashed into multiple partitions by the rule $(\text{vertex_id} \% \text{the_number_of_partitions})$. A partition is a virtual set of data in Nebula Graph. These partitions are allocated to all storage hosts.

This design ensures that graph data and index data is evenly distributed across partitions. However graph data is converted and stored in partitions by hashing. This breaks the local connectivity of the graph itself. Consequently, Nebula Graph has increased redundant storage requirements. Data accessed by a graph operation may be distributed across multiple partitions and storage engines, which requires frequent cross-partition accesses and consequently increases network overhead. In addition, the graph operation is run sequentially over the storage layer. This precludes full utilisation of CPU or memory over the nodes in the Nebula Graph.

TigerGraph is a Native Parallel Graph (NPG) system that forms a complete, distributed, graph analytics platform supporting web-scale data analytics in real time. The TigerGraph NPG is built around both local storage and computation, supports real-time graph updates, and works like a parallel computation engine. The massive parallelism process of TigerGraph is implemented depending on the accumulators. There are a number of different types of accumulators in TigerGraph, each providing specific accumulation functions. Accumulators are declared to have one of two types of association: global or vertex-attached.

Despite running processes in parallel, TigerGraph still suffers from the drawbacks. At the storage level, TigerGraph retains coupled edges and adjacencies. When the graph travels, it needs to scan the data of all the adjacency edges, which contain information about the attributes of edges. If there is no need for attribute or label filtering, the edge attribute information

- is useless, resulting in wasted memory. Also, when a query has a large number of aggregated calculations, many global or vertex-attached aggregators are created. This increases inter-thread communication due to more metadata and status switches being required. Accumulator declarations
- 5 define the accumulator instances to be created by a query, specifying their type and name. It means graph query language has to support the accumulator and users need to understand the concept of accumulation calculation before they can define query statements.
- 10 It would be desirable to provide a framework that overcomes or ameliorates at least one of the above-described problems, or at least provides a useful alternative.

Summary

- 15 With the above drawbacks of the prior art in mind, the present disclosure provides an action-based parallel graph process framework and strategy for large numbers of entities. The present framework realizes parallel execution of graph processing and improves performance on the basis of guaranteed native
- 20 graphs. Moreover, connections between entities stored in the graph are not destroyed. The present framework is compatible with existing graph query languages such as Gremlin, Cypher, etc, and users do not need to understand parallel execution logic.
- 25 The present framework operates by dividing a large graph into smaller, subgraphs that are stored separately and can be queried without loss of interrelatedness in the dataset as a whole. It enables a distributed dataset to be queried without loss of information.
- 30 Disclosed is a network comprising:
- a plurality of graph storage instances;
 - an application program interface (API) gateway for receiving a request from a requestor and converting the request to a query;
 - a graph query instance for generating an execution pipeline based on
- 35 the query, the execution pipeline comprising a plurality of actions;

a metadata service for managing the execution pipeline; and
wherein the graph storage instances collectively store a graph formed
from a plurality of entities and comprising a vertex for each entity,
edges connecting pairs of vertices, and adjacency relations, each graph
storage instance storing a subgraph of the graph, and comprising:
5 a partition comprising:
a vertex set of all the vertices in the respective subgraph,
the vertex sets of the plurality of graph storage instances being
non-overlapping;
10 an edge set of all the edges between vertexes in the
respective vertex set; and
the adjacency relations for the edges in the respective
edge set and for each edge connecting a vertex in the vertex set
to a vertex in another partition;
15 an executor for executing one or more of the actions over the
respective subgraph; and
an action monitor for supplying actions to the respective executor.

Also disclosed is a method for executing a query over the network described
20 herein, comprising:
receiving a new request for a requestor;
converting the new request to a new query;
generating a new execution pipeline based on the new query, the new
execution pipeline comprising a topology graph of new actions, each new
25 action comprising an Action ID for the new action, and Partition ID for the
graph storage instance on which the respective new action is to be executed
(corresponding graph storage instance), and a type for the new action;
for each said new action:
sending the new action to the corresponding graph storage
30 instance based on the execution pipeline;
executing the new action at the execution monitor of the
corresponding graph storage instance; and
returning a result of execution of the new action;
generating an aggregated result from the results of execution of each
35 new action; and

outputting the aggregated result to the requestor.

Also disclosed is a method for storing data, comprising:

receiving a plurality of entities;

5 forming a graph from the plurality of entities, the graph comprising a vertex for each entity, edges connecting pairs of vertices, and adjacency relations;

dividing the graph into a plurality of non-overlapping subgraphs; and

10 allocating each subgraph to a respective graph storage instance, each graph storage instance comprising a partition comprising a vertex set of all vertices in the respective subgraph, an edge set of all edges between vertexes in the respective vertex set, and the adjacency relations for the edges in the respective edge set and for each edge connecting a vertex in the vertex set to a vertex in another partition, each graph
15 storage instance further comprising an executor for executing actions over the respective subgraph and an action monitor for supplying actions to the respective executor.

Brief description of the drawings

20

Embodiments of the present invention will now be described, by way of non-limiting example, with reference to the drawings in which:

Figure 1 shows a graph storage scheme in accordance with present teachings,
25 for storing a graph across multiple partitions, where each partition contains all entities necessary for an action executed over the respective partition;

Figure 2 illustrates a method for storing the data in accordance with the scheme of Figure 1;

30

Figure 3 is a schematic system architecture for storing and querying a graph, in accordance with present teachings;

Figure 4 shows the process of running a query through the architecture of
35 Figure 3; and

Figure 5 illustrates interaction between an execution pipeline and various modules (herein interchangeably referred to as "instances") of the system architecture of Figure 3.

5

Detailed description

Described below is a high-performance parallel graph process framework. Embodiments of the framework can be used for real-time retrieval and updating of large-scale entities (i.e. data points) such as trillion-scale entities. In each embodiment illustrated below, graph processes are run on native graphs or subgraphs. That means entity adjacency is stored natively and accessed cheaper. Also, queries can be executed in parallel, to access the computing potential of each node.

15

Using an API, the present framework is compatible with existing graph query languages such as Gremlin, and users do not need to understand the execution logic of parallel graph processes.

20 The term "node" may refer to a graph query module, graph storage module, or any other part of the network, as provided for by context.

Figure 1 illustrates the distributed storage strategy for massively parallel for graph processes premised on distributed native graph storage. Using this strategy, an overall graph 100 (interchangeably referred to as an integrity graph) comprising a very large number of vertices is stored as subgraphs in multiple partitions 102, 104, 106. Storage is achieved by cutting edges of the graph 100. The subgraphs are stored natively on the partitions without relying on any other engine or backend.

30

To maintain integrity of the graph 100, the properties and adjacency of edges that are cut are stored in both partitions that contain a vertex of the cut edge. For example, the edge between vertices D and K is cut such that vertex D only appears in partition 102 and vertex K only appears in partition 106. Consequently, the properties and adjacency of the edge joining vertex D with

35

vertex K in graph 100 is stored in both partitions 102 and 104. For a partition containing a cut edge and one of two vertices joining that edge, the edge may be stored in with the identifier (Partition ID) of the partition containing the other vertex. In some embodiments, this enables an action executed over one
5 partition to result in an action to be executed over the other partition.

Each vertex and edge (i.e. each entity) in a partition may be stored with the Partition ID. The Partition ID may, for example, form part of the unique identifier for each vertex and/or edge.

10

Each partition stores at least a set of vertices (vertex set 110) of the respective subgraph, a set of edges (edge set 112, being the set of all edges between vertices in the respective vertex set and any cut edges) and adjacency relations (adjacency 114, being the adjacency data for the edges in
15 the respective edge set, and any cut edges). The set of vertices may comprise a map of the vertex identifiers (Vertex IDs) and properties of each vertex in the set. The set of edges may comprise a map of edge identifiers (Edge IDs) and edge properties. The edges define relationships between entities stored in the source and target vertices, and may include an edge "type". The
20 adjacency relations are stored persistently and describe the mappings between edges, vertices and directionality of the edges. To illustrate directionality, for an edge with a direction from vertex A to vertex B, vertex A will be the source vertex and B will be the target vertex. Adjacency relations may include <Edge ID, source Vertex ID>, <Edge ID, target Vertex ID> and
25 <(Vertex ID + direction(input/output)), Edge ID>.

In some embodiments, each partition also stores the attributes and label indexes of vertices and edges.

30 Figure 2 illustrates a method 100 for storing data in the graph structure shown in Figure 1. Broadly, the method 100 involves receiving the data. In the case of a graph, the data comprises entities, each of which represents a data point with properties. Thus, receiving data involves receiving a plurality of entities (step 202). At step 204 a graph is formed similar to graph 100. The graph is a
35 set of vertices – one vertex for each entity – connected by edges, and having

adjacency data. The adjacency data may be inherent in the edges and their directionality, or may be stored in a separate topology graph.

5 The graph formed at step 204 is then divided into a plurality of non-overlapping subgraphs at step 206. The subgraphs are non-overlapping insofar as they each comprise a set of vertices none of which is stored in any other one of the subgraphs. At step 208, each subgraph is allocated to a respective graph storage module (graph storage modules are discussed with reference to Figure 3).

10

The method 100 thus takes a large graph and divides it into smaller, separate datasets (subgraphs). Each subgraph comprises vertices, edges and adjacency relations, and includes the adjacency and properties of each edge that is cut to form the subgraphs – i.e. each edge for which only the source or target vertex, but not both, is in the partition. The adjacency information for any edge that has been cut is stored in each of the partitions that contain the source and target vertices.

Figure 3 shows a network 300 used for querying the graph storage framework illustrated in Figure 1. The network 300 includes multiple graph storage modules 302, an application program interface (API) gateway 304, at least one, and presently multiple, graph query module 306 and a metadata service 308. The action-based parallel process framework described herein, and implemented using the network 300, is an improvement on the graph database execution mechanisms of previously known technologies. The network and graph storage framework described herein can be employed in any business constructed on the basis of graph databases.

Requests are received from requestors 310, through the API gateway 304. Requestors may be services offered by various parties, or any other type of business. The API gate 304 converts each request to a query. The requests from the requestor (e.g. a business) pass through API Gateway 304, where API servers extract the graph query statements (queries) and deliver them to any graph query instances. Allocation or delivery of queries to graph query instances may involve allocating a query to any unutilised graph query

35

instance, allocating to the graph query instance with the shortest existing execution pipeline or shortest queue of queries awaiting execution (where graph query instances store a list of queries), or in accordance with any other load balance strategy.

5

The graph query instance 306 receives the query and generates an execution pipeline based on the query – this can involve optimising and parsing the query into a topology graph of actions. The graph query instances are stateless insofar as a graph query instance requires no knowledge of prior
10 state to correctly interpret the query – all of the necessary information is included in the query itself.

The execution pipeline comprises a plurality of actions that are executed to respond to the query. To that end, the graph query instance 306 comprises a
15 pipeline manager 310 and an action monitor 312. The topology graph of actions is generated by the pipeline manager 310. This is achieved by the pipeline manager 310 recursively generating new action requests from which the action monitor 312 returns a corresponding new action that the pipeline manager 310 incorporates into the topology graph. Representing the
20 execution pipeline as a topology graph enables parallel execution over multiple subgraphs, where those subgraphs each contain data that needs to be queried before a subsequent action in the execution pipeline can be executed.

The action monitor 312 is responsible for monitoring the life cycle of actions
25 and scopes. In addition to generating new actions for the execution pipeline, the action monitor 312 delivers actions to the corresponding executors in partitions of graph storage instances 302. The action monitor is both a server and a client and they can communicate with each other.

30 In some embodiments, the execution pipeline is sent to action monitors of graph storage instances 302, directly from the graph query instance 306. This can occur for trivial queries – e.g. vertex or entity count. In other embodiments, execution pipelines are sent to the meta service 308. The meta service or metadata service 308 manages execution pipeline. It is a cache
35 service that supports high-speed read and write. Meta service 308 provides

setting/getting metadata and metadata management for all instances including graph query instances and graph storage instances.

Once the execution pipeline is generated, actions are sent to respective graph storage instances 302 for execution. The actions may therefore each have an Instance ID (for graph storage instances, the Instance ID may be the Partition ID) that directs the action to the correct graph storage instance 302. The action monitor 314 in each graph storage instance 302 receives the action and applies a status – e.g. created, initialized, executing, executed, as discussed below. Once an action has been created and then initialised by the respective action monitor, it is sent to the executor 316 in the same graph storage instance 302, for execution and the action status is changed to "executing".

The executors 316 in graph storage instances are responsible for execution of actions over the respective subgraph under the supervision of action monitors. Specifically, the vertices, edges, attributes, etc. needed for one execution or one action are available locally in the respective graph storage instance 302 and are not accessible across partitions. The graph storage instances 302 are stateful, meaning that previous (executed) actions may affect the current action. Every graph storage instance has one partition 318 (one master or one slaver) and executors.

The network or architecture 300 also includes a further plurality of graph storage instances 320. Each graph storage instance in the further plurality of graph storage instances 320 contains a partition that is duplicated from one of graph storage instances 306. Thus, every partition may have one or multiple replicas (duplicates). This ensures that the state of graph data on persistent (non-transitory) media such as a solid state drive (SSD) or hard disk drive (HDD) is permanently consistent. It is a high-available storage system or engine.

If graph storage instance 302 is broken or has no available computing resources, actions containing the Instance ID of the relevant graph storage instance 302 can be updated with the instance ID of a duplicate from graph storage instances 320, such that the query can be processed before the

relevant graph storage instance 302 would have managed to do so. In effect, multiple queries can therefore be concurrently run on the same data, without queueing queries.

- 5 In other embodiments, or in addition to the duplicated graph storage instances, each graph storage instance of the plurality of graph storage instances may contain multiple copies of the corresponding partition, executor and/or other components. An action received by the graph storage instance is executed over a first partition of the plurality of partitions, referred to as the master
10 partition. If execution fails due to the first partition being faulty, or the executor being faulty, the action monitor of the graph storage instance either receives no response within a given timeframe, or a response indicating failure of the partition or executor. The action monitor then directs the query to a second partition, referred to as a slave partition, and/or executor within the
15 same graph storage instance. That slave partition contains a duplicate of the information in the master partition, such that executing a query over the master or slave will yield the same result if both partitions are fault-free.

- Querying a graph such as graph 100, over a network such as network
20 architecture 300, therefore comprises receiving a request and converting it to a query, and using the pipeline manager to generate an execution pipeline. The execution pipeline comprises a topology graph of new actions, each action comprising an Action ID (i.e. an identifier) for the new action, and Instance ID for the graph storage instance on which the respective new action is to be
25 executed, and a type for the new action. The Instance ID for the graph storage instance or instances on which actions are to be executed may be performed by initially conducting a global search for the graph storage instance or instances. For example, with reference to Figure 5, each vertex may be a record of a service used by a user – e.g. purchase of food, use of a
30 ride-sharing service, delivery of a document and so on. A query *g* may invoke a function *toList()* that returns a list of vertices *V()* that have a code *code* corresponding to a use of a service (or other information as may be stored in a vertex) in Australia *AUS*. A graph query instance *Instance 1*, receives a query and searches all graph query instances in the plurality of graph query
35 instances, to find a vertex corresponding to the query – e.g. any vertex with

code = AUS. In this case, *Instance 2* does not include any relevant vertex, whereas *Instances 3, 4* and *5* each include a relevant vertex. Each graph query instance returns an indicator or signal advising whether or not it includes a vertex complying with the query. Actions are therefore inserted into the execution pipeline for each graph query instance comprising a relevant vertex – presently, each of *Instances 3, 4* and *5*. Once the execution pipeline is built, the relevant actions are sent to each instance comprising a relevant vertex – presently, *Instances 3, 4* and *5* at steps 504, 506 and 508. The result of execution – e.g. a list of VertexIDs for each vertex (each VertexID may comprise the corresponding unique graph storage ID for the graph storage instance in which the vertex is stored, plus an index or number corresponding to the particular vertex within that graph storage instance) for which the query is valid (i.e. code = AUS is valid) – is sent from each relevant instance, presently *Instances 3, 4* and *5*, to the first instance, *Instance 1*. The first instance then aggregates or processes the results – e.g. aggregates the result from each of *Instances 3, 4* and *5* and outputs or lists the result of that aggregation. The same process can apply where the query relates to edges or other properties of the graph.

The action monitor manages distribution of the execution pipeline and action life cycle. The action monitors can therefore receive their execution pipeline from the pipeline manager or from another action monitor (or the meta service 308), based on the execution pipeline (e.g. Instance ID in the next action in the execution pipeline). The action monitors in each graph storage instance pass the actions in the execution pipeline to the executor that executes the actions over the subgraph and returns the result. All the results are ultimately aggregated into an aggregated result depending on the query, and thus on the original request. The aggregated result is then outputted to the relevant requestor.

30

Actions

An "action" is an execution plan that can be assigned to a corresponding executor and executed on a unique partition, the executor and partition being in the graph storage instance corresponding to the Instance ID in the action.

In the present context, the term "unique partition" refers to the partition being

35

unique (and typically non-overlapping) in the graph storage instances 302, bearing in mind that instances 320 may comprise a duplicate in the event that the relevant graph storage instance 302 is broken or occupied. Notably, instances 302 may be stored in faster memory than instances 320, to save
5 cost. The required vertices, edges, attributes, etc. are located on the relevant partition during execution.

Action Type

Action types are abundant and include all necessary actions and information
10 required to execute all defined queries over the graph. Actions include all operations such as vertices, edges, properties, traverse, path, schema, index, graph, etc. In general, action types are equivalent to executor types. Some common types are as follows, which can obviously be expanded.

15 Table 1 Action Type

Type	Description
Start	Start marks for the pipeline topology graph and prepares the input arguments and metadata.
End	End marks for the pipeline topology graph and return query results to clients.
GetNeighbors	Get the set of adjacency entities such as vertices and edges for a specified vertex from users.
GetVertices	Get user-specified vertices attributes
GetEdges	Get user-specified edge attributes
Traverse	Some operations such as pattern matching or data aggregation, etc maybe occur in the overall traversal process with DFS or BFS.
ShortestPath	Get the shortest path between two vertices.
IndexScan	Support index lookup and match.
Filter	Select the results that meet the specified conditions.
Union	Merge A and B sets to ensure that elements in results are not duplicated.
Intersect	get the intersection of A and B sets
Relative Complement	The relative complement of B in A is the set of all elements that are members of A, but not members of B.

GetProperties	Get the specified properties about vertices or edges.
Sort	Sorts the elements of a list in the specified order such as DESC or ASC.
TopN	Returns the top N elements of an ordered list
Limit	Take elements with user-specified limits into results
Sample	Select the specified amount of elements according to the sampling rules.
Aggregate	Calculate a set of data uniformly. use values from multiple records as input and convert those values into one value to aggregate all records
Dedup	Remove duplicate elements from the specified data list.
GetAllPath	Get all paths from source vertex to target one.
Subgraph	Get subgraph including all vertices and edges form start vertex according to match or filter conditions
(Others)	...

Action Member

Each action comprises parameters, referred to as "members". One action consists of the following members.

5 Table 2 Action Members

Member	Description
QueryContext	Query meta information form requests sent by users, which includes the query ID.
ID	A unique identifier for the action.
InstanceID	Allocate actions to the specified graph storage instance where the partition is located or graph query instance.
Type	It indicates the type of action.
Status	Action status includes CREATED, INITIALIZED, EXECUTING, EXECUTED. CREATED: The new action is created by action monitors INITIALIZED: The input parameters and metadata is initialized before the action is executed. EXECUTING: This action is being executed by the executors. EXECUTED: This action is executed and its results are collected.

Input	The input parameters and metadata required for the action to be executed.
Output	The result of the action execution, including the results of the execution and the metadata required to be passed to the next actions.
PriorActionList	A list of prior action IDs of this action
NextActionList	A list of next action IDs of this action

Action Monitor

The action monitor is a monitor that is responsible for the creation, destruction, delivery, part-execution and monitoring of actions. There is one, and generally
5 only one, action monitor on each instance including graph query instances and graph storage instances.

Requests for creating new actions are sent to the action monitors on the graph storage instances, by the pipeline managers or executors according to the
10 execution pipeline – e.g. each action in the execution pipeline results in an action request being sent to the corresponding graph storage instance. When an action monitor in a graph storage instance receives a request, a new action is needed. If the new action uses data in the partition of the same graph storage instance as the action monitor, then a new action can be generated by
15 the action monitor. Once requests are accepted by action monitors, the new actions will be created. When actions are executed by executors or action monitors, they are destroyed and set to done (i.e. executed) state in the pipeline topology graph. When an action is accepted by action monitors, it will be delivered to another action monitors of the specified partitions. The another
20 action monitors will determine whether to accept this action according to partition ID, then execute it by executors in the same partitions or itself. For some actions, execution will require action to be taken by another graph storage instance – e.g. where execution of an action requires a different action type to be used or requires data accessible via a cut edge. In such cases, the
25 action monitor on which the current action is being executed may update the status of the action to "executed" and will generate a new action for incorporation into the execution pipeline, or send a new request to the partition holding the data necessary to execute the new action.

Feature Description

Table 3 Feature Description

Feature	Description
Create	Allocate unique IDs for new actions and initialize members of actions
Destroy	Get results of execution and some data is passed to the next actions. Change the action status to EXECUTED.
Send	Send actions to the destination action monitors. The destination address of action monitors is included in the delivery requests.
Execute	Normally, the actions accepted by action monitors are executed by the executors on the same partitions in graph storage instances. Some actions used for data collection or results sort, are executed by action monitors in graph query instances.
Monitor	In graph query instances, the process of generating pipeline topology graphs is monitored. In graph storage instances, the action execution is monitored. It is mainly to observe two situations that are whether a new action type is generated and the calculated entities are across partitions.

5

To illustrate the process, Figure 4 shows the flow of actions in the architecture of Figure 3. When the AST (Abstract Syntax Tree), metadata and arguments 400 from one query are inputted into the pipeline manager 402, the generation of the topology graph, and thus the action flow, starts. During the first stage, the pipeline manager 402 generates the topology graph generation by continuously sending requests of new actions to the local action monitor 404 according to rules determined from inputs 400. The action monitor 404 then creates and returns a new action to the pipeline managers 402 until the topology graph (execution pipeline) is constructed.

10

15

In the second stage, after the topology graph is stored in the meta service 406, the local action monitor will get the start action from the topology graph and prepare for executions, e.g. begin transaction. The local action monitor then
5 sends start signals or requests to the remote action monitors 408 in the graph storage instances according to the destination of next actions as determined from the Instance ID in each relevant action.

In the third stage, once the remote action monitors 408 receive the start
10 signals, an action is retrieved from the topology graph in the meta service 406, initialized and taken to the local executors 410. The local action monitor updates the status of the action to EXECUTING – the update may occur locally in the graph storage instance and/or may be sent to the topology graph or action monitor in the meta service 406. The action is then executed over the
15 partition 412 in the same graph storage instance and the executor 410.

In the fourth stage, which may not occur for all queries, some entities (vertices) may be determined, during execution, to be not on the partition 412 with the executor 410. In such cases, the executor 410 will report to the local
20 action monitor 408. After the local action monitor 408 receives this report, it will create a new action and update it to the topology graph. Once a new action is updated, the action being executed by executor 410 is stopped – e.g. is updated to EXECUTED. Therefore, execution can involve generating one or more further actions based on the report and incorporating the one or more
25 further actions into the new execution pipeline.

In the fifth stage, if some actions are executed or stopped by executors, the local action monitors will collect results and change the status of actions to EXECUTED. Then the Instance ID of the next action or actions is/are obtained
30 from the topology graph and the next requests is/are sent to the remote action monitor(s) corresponding to the Instance ID(s). Once the remote action monitors receive the next requests, the operations in the third to fifth stages repeat until the last action.

In stage seven, the last action is generally a data aggregation step resulting in serialization of the results in the final step of one query. That final action is executed by the action monitor in the graph query instances, or the graph query instance that originally generated the execution pipeline. If the destination of the last action is broken, occupied or otherwise unresponsive, the action monitor sending the next or final request will look for healthy instances (e.g. operational graph query instance) and modify the instance ID of the last action in the topology graph. The action monitor then sends the request to the new destination (healthy instance) again. The same can occur where a graph storage instance is broken or otherwise unresponsive – i.e. a determination may be made that, for one or more actions, the corresponding graph storage instance is broken in which case the Instance ID for the one or more actions is updated using the Instance ID of a duplicate graph storage instance (e.g. from instances 320 of Figure 3).

In the eighth stage, when the end action is obtained, the results of this query are returned to clients (requestors). At this point, the action flow is over, and all executions of this query are completed.

Key advancements in the present teachings are a massively parallel process for native graphs that executes actions in parallel on multiple partitions, relying on graph distributed storage. A simple example shown in Figure 5 illustrates the parallel process strategy. The topology graph 500 generated from a gremlin query contains 6 actions 502 to 512 – one action for each Instance ID, even though some actions (e.g. actions 504, 506 and 508) are the same. The actions 502 to 512 are allocated to the specified action monitors 514, 416, 518 according to the Instance IDs. When the property filter of vertices is in execution, the three VertexPropertyFilter actions 504, 506, 508 (i.e. actions to filter vertices in the subgraphs based on a particular property, presently the input condition "AUS" being for rides requested, or other service requests made, in Australia) are assigned to Instance 3, 4 and 5 respectively. The actions 504, 506, 508 are executed in parallel on the respective partitions of Instances 3, 4 and 5 – the data required for the calculation are all on their respective partitions without affecting each other.

The resources such as CPU or memory required for the calculation of these three actions are all on their respective partitions.

5 The results of actions 504 506, 508 are inputs of ListResult action 510 and aggregated to Instance 1 for the next execution.

10 In this way, queries can be run in massively parallel processes across distributed graph storage, based on native graph execution and local resource utilisation. This means that edge computing devices can be used or that the size of any subgraph in a graph storage instance can be sized to suit the amount of, and speed of, computing resources accessible to the graph storage instance. The framework described herein preserves connections between entities stored in different subgraphs and is compatible with existing graph query languages such as Gremlin, Cypher, etc, and users do not need to
15 understand parallel execution logic.

The reference in this specification to any prior publication (or information derived from it), or to any matter which is known, is not, and should not be taken as an acknowledgment or admission or any form of suggestion that that
20 prior publication (or information derived from it) or known matter forms part of the common general knowledge in the field of endeavor to which this specification relates.

25 Throughout this specification and the claims which follow, unless the context requires otherwise, the word "comprise", and variations such as "comprises" and "comprising", will be understood to imply the inclusion of a stated integer or step or group of integers or steps but not the exclusion of any other integer or step or group of integers or steps.

30 The scope of this disclosure encompasses all changes, substitutions, variations, alterations, and modifications to the example embodiments described or illustrated herein that a person having ordinary skill in the art would comprehend. The scope of this disclosure is not limited to the example embodiments described or illustrated herein. Moreover, although this
35 disclosure describes and illustrates respective embodiments herein as

including particular components, elements, feature, functions, operations, or steps, any of these embodiments may include any combination or permutation of any of the components, elements, features, functions, operations, or steps described or illustrated anywhere herein that a person having ordinary skill in

5 the art would comprehend. Although this disclosure describes or illustrates particular embodiments as providing particular advantages, particular embodiments may provide none, some, or all of these advantages.

Claims

1. A network comprising:
 - a plurality of graph storage instances;
 - 5 an application program interface (API) gateway for receiving a request from a requestor and converting the request to a query;
 - a graph query instance for generating an execution pipeline based on the query, the execution pipeline comprising a plurality of actions;
 - a metadata service for managing the execution pipeline; and
 - 10 wherein the graph storage instances collectively store a graph formed from a plurality of entities and comprising a vertex for each entity, edges connecting pairs of vertices, and adjacency relations, each graph storage instance storing a subgraph of the graph, and comprising:
 - a partition comprising:
 - 15 a vertex set of all the vertices in the respective subgraph, the vertex sets of the plurality of graph storage instances being non-overlapping;
 - an edge set of all the edges between vertexes in the respective vertex set; and
 - 20 the adjacency relations for the edges in the respective edge set and for each edge connecting a vertex in the vertex set to a vertex in another partition;
 - an executor for executing one or more of the actions over the respective subgraph; and
 - 25 an action monitor for supplying actions to the respective executor.
2. The network of claim 1, wherein the graph query instance comprises a pipeline manager and an action monitor, wherein the execution pipeline is built by the pipeline manager sending new action requests to the
30 action monitor based on the query, and the action monitor creates one or more actions according to each new action request, the execution pipeline comprising the created actions.
3. The network of claim 2, wherein the execution pipeline is stored as a
35 topology graph in the metadata service.

4. The network of any one of claims 1 to 3, wherein, for each graph storage instance, the partition is a master partition, the respective graph storage instance comprising one or more slave partitions, each slave partition being a duplicate of the master partition.
5. A method for executing a query over the network of any one of claims 1 to 3, comprising:
- receiving a new request for a requestor;
 - 10 converting the new request to a new query;
 - generating a new execution pipeline based on the new query, the new execution pipeline comprising a topology graph of new actions, each new action comprising an Action ID for the new action, and Partition ID for a corresponding graph storage instance on which the respective new action is to be executed, and a type for the new action;
 - 15 for each said new action:
 - sending the new action to the corresponding graph storage instance based on the execution pipeline;
 - executing the new action at the executor of the corresponding graph storage instance; and
 - 20 returning a result of execution of the new action;
 - generating an aggregated result from the results of execution of each new action; and
 - 25 outputting the aggregated result to the requestor.
6. The method of claim 5, wherein, for one or more said new actions, executing the new action comprises identifying one or more entities that are on a partition that does not correspond with the partition of corresponding graph storage instance, and returning the result of execution of the respective new action comprises sending a report comprising the one or more entities to the action monitor of the corresponding graph storage instance.
- 30
7. The method of claim 6, comprising generating one or more further actions based on the report and incorporating the one or more further
- 35

actions into the new execution pipeline.

- 5 8. The method of any one of claims 5 to 7, wherein the network comprises a further plurality of graph storage instances, each graph storage instance in the further plurality of graph storage instances comprising a partition that is duplicated from a graph storage instance in the plurality of graph storage instances, the method comprising:
- 10 determining, for one or more new actions, that the corresponding graph storage instance is broken;
- 10 updating the Partition ID for the one or more new actions, based on a graph storage instance, from the further plurality of graph storage instances, the partition of which is a duplicate of the partition of the broken corresponding graph storage instance.
- 15 9. A method for storing data, comprising:
- receiving a plurality of entities;
- forming a graph from the plurality of entities, the graph comprising a vertex for each entity, edges connecting pairs of vertices, and adjacency relations;
- 20 dividing the graph into a plurality of non-overlapping subgraphs; and allocating each subgraph to a respective graph storage instance, each graph storage instance comprising a partition comprising a vertex set of all vertices in the respective subgraph, an edge set of all edges between vertexes in the respective vertex set, and the adjacency relations for
- 25 the edges in the respective edge set and for each edge connecting a vertex in the vertex set to a vertex in another partition, each graph storage instance further comprising an executor for executing actions over the respective subgraph and an action monitor for supplying actions to the respective executor.

30