US 20210182315A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2021/0182315 A1**

Haprian et al. (43) **Pub. Date: Jun. 17, 2021**

(54) **HYBRID IN-MEMORY BFS-DFS APPROACH FOR COMPUTING GRAPH QUERIES AGAINST HETEROGENEOUS GRAPHS INSIDE RELATIONAL DATABASE SYSTEMS**

(71) Applicant: **Oracle International Corporation**, Redwood Shores, CA (US)

(72) Inventors: **Vlad Haprian**, Zürich (CH); **Laurent Daynes**, Saint-Ismier (FR); **Shasank K. Chavan**, Menlo Park, CA (US); **Jean-Pierre Lozi**, Zürich (CH); **Vasileios Trigonakis**, San Jose, CA (US); **Sungpack Hong**, Palo Alto, CA (US); **Marco Arnaboldi**, Zürich (CH); **Ciprian Baetu**, Zürich (CH)

(21) Appl. No.: **16/710,719**

(22) Filed: **Dec. 11, 2019**

Publication Classification

(51) **Int. Cl.**

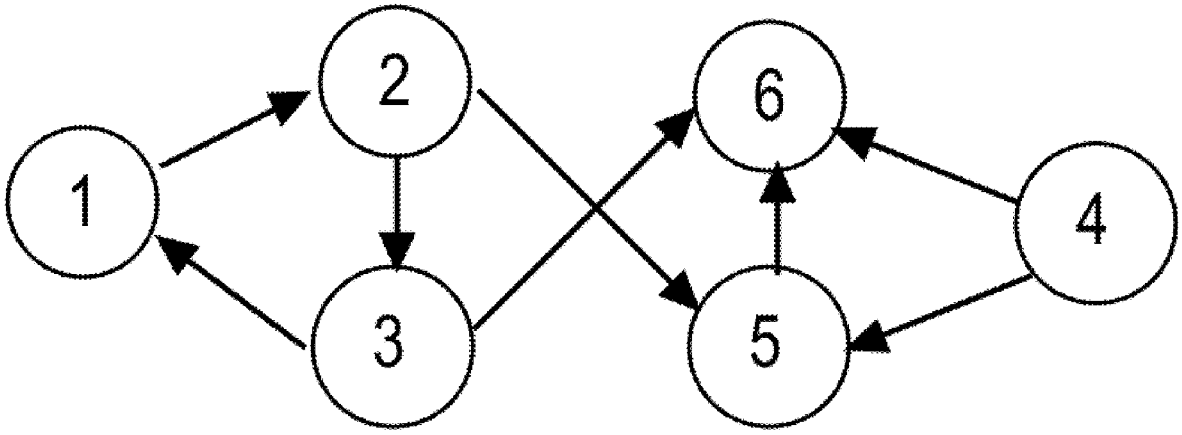
G06F 16/28	(2006.01)
G06F 16/22	(2006.01)
G06F 16/901	(2006.01)
G06F 16/242	(2006.01)

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

CPC ***G06F 16/285*** (2019.01); ***G06F 16/242*** (2019.01); ***G06F 16/9024*** (2019.01); ***G06F 16/2246*** (2019.01)

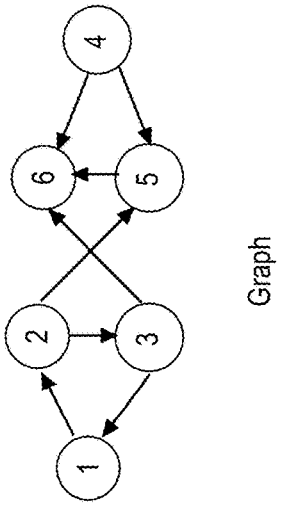
(57) **ABSTRACT**

An in-memory graph query runtime is integrated inside a database management system and is capable of performing simple patter-matching queries against homogeneous graphs. The runtime efficiently combines breadth-first (BFS) and depth-first (DFS) neighbor traversal algorithms to achieve a hybrid runtime that takes the best from both sides. As a result, the hybrid runtime is able to process arbitrarily large queries with a fixed amount of memory, optimizing for memory locality.



Graph

Fig. 1



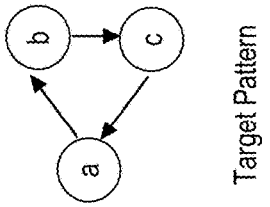


Fig. 2A

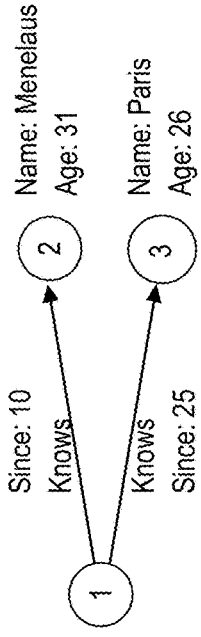


Fig. 2B

Persons Table

Vid	Name	Age
1	Helen	25
2	Menelaus	31
3	Paris	26

Knows Table

Eid	scrId	dstId	Since
1	1	2	10
2	1	3	25

Fig. 3

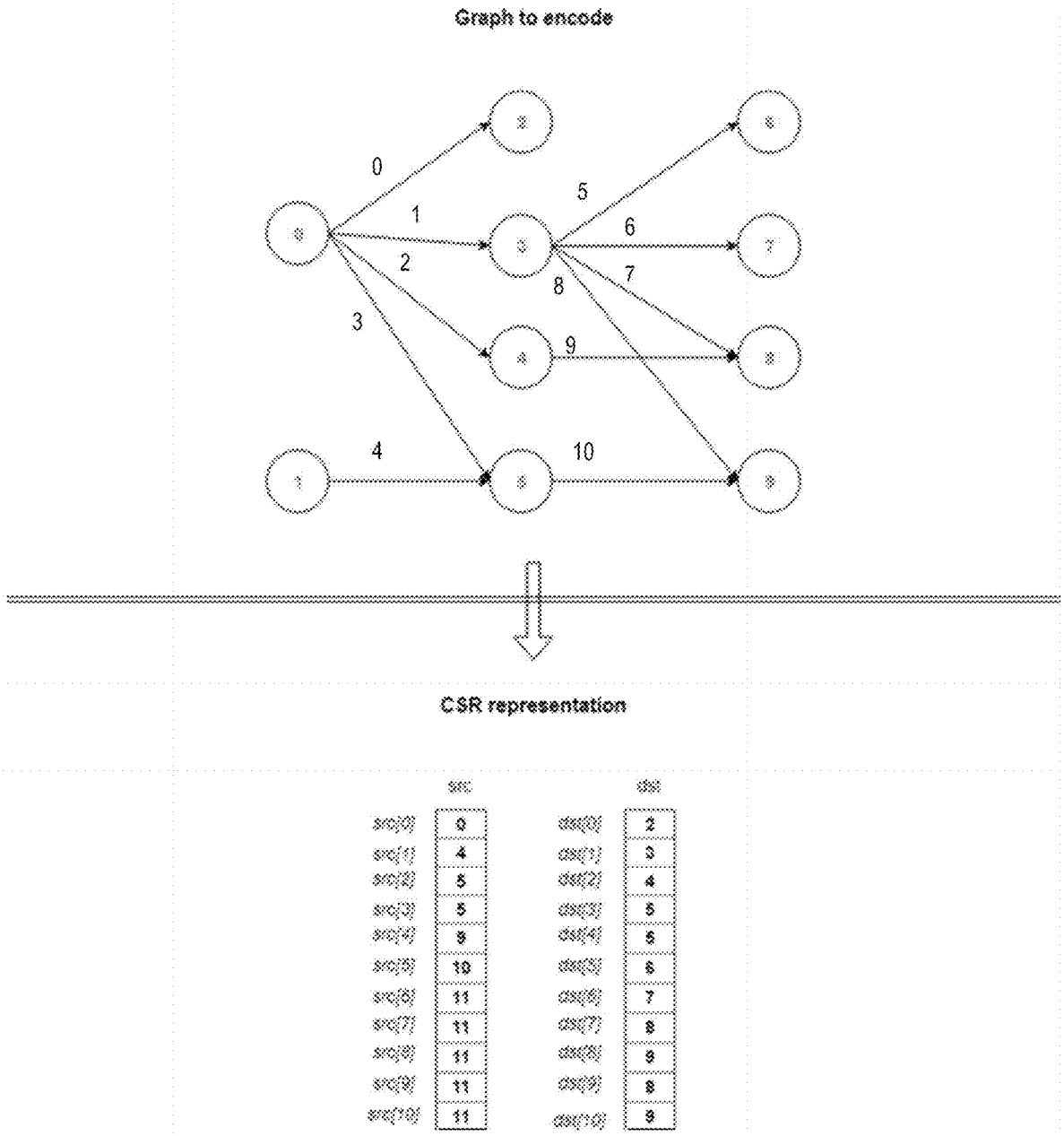

Graph to encode
0
1
2
3
4
5
6
7
8
9
10
CSR representation
src
dst
src[0] 0
src[1] 4
src[2] 5
src[3] 5
src[4] 9
src[5] 10
src[6] 11
src[7] 11
src[8] 11
src[9] 11
src[10] 11
dst[0] 2
dst[1] 3
dst[2] 4
dst[3] 5
dst[4] 5
dst[5] 6
dst[6] 7
dst[7] 8
dst[8] 9
dst[9] 8
dst[10] 9

Fig. 4

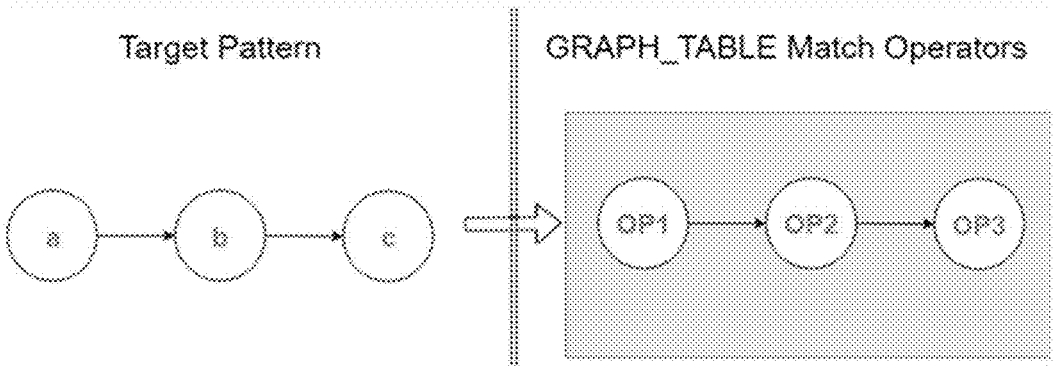
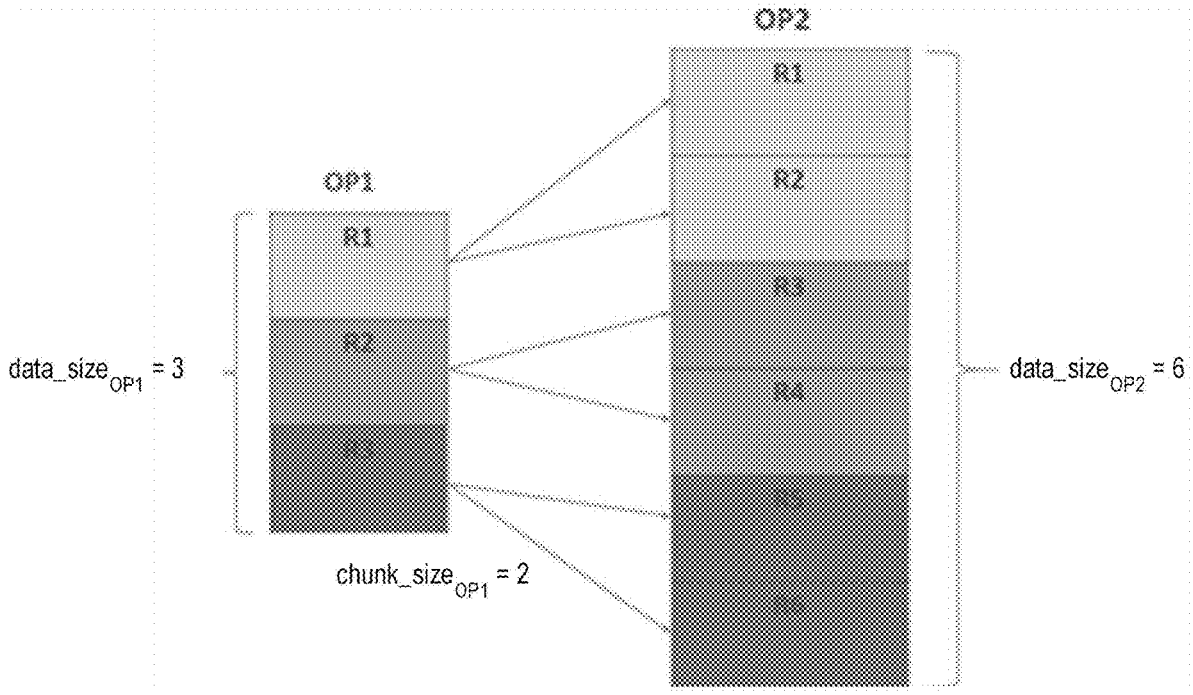

Target Pattern
GRAPH_TABLE Match Operators
a
b
c
OP1
OP2
OP3

Fig. 5


OP2
R1
R2
R3
R4
OP1
R1
R2
R3
data_size OP1 = 3
chunk_size OP1 = 2
data_size OP2 = 6

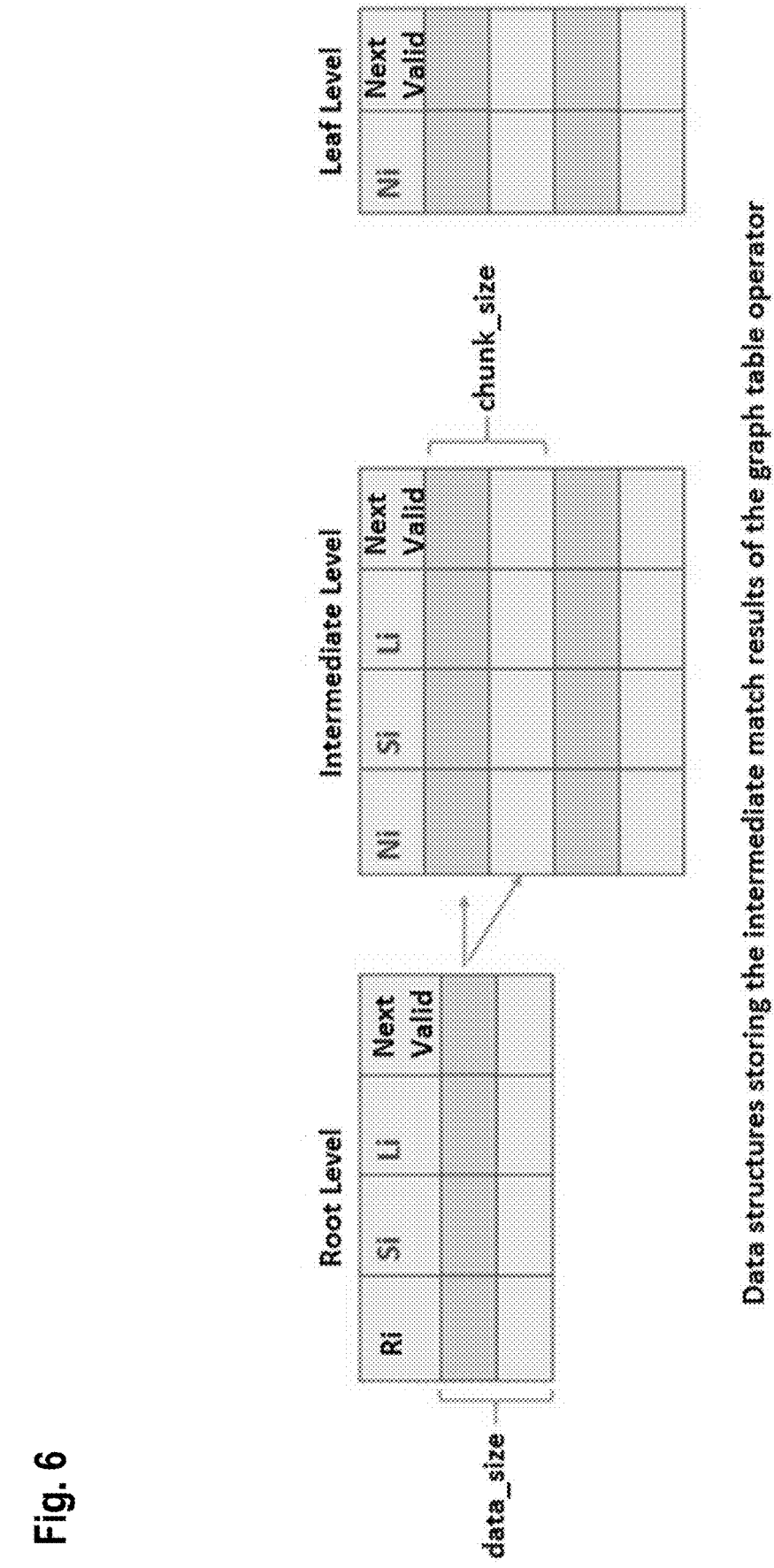

Root Level
Ri
Si
Li
Next Valid
data_size
Intermediate Level
Ni
Si
Li
Next Valid
chunk_size
Leaf Level
Ni
Next Valid
Data structures storing the intermediate match results of the graph table operator

Fig. 6

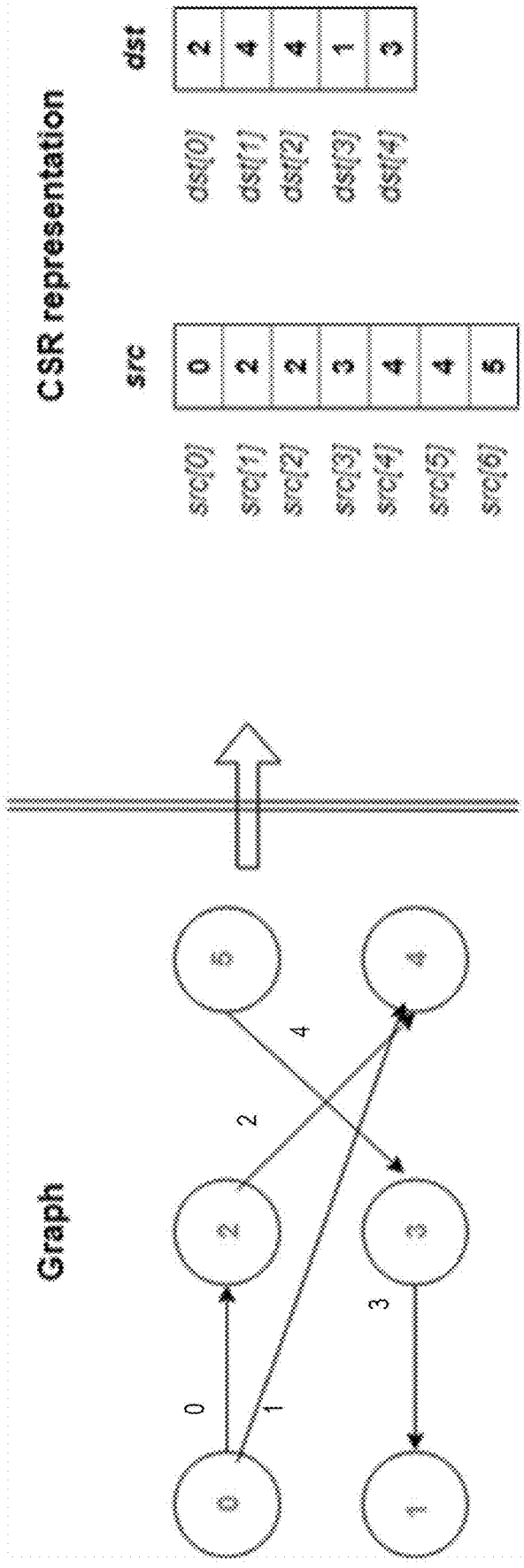

CSR representation
dst
2
4
4
1
3
dst[0]
dst[1]
dst[2]
dst[3]
dst[4]
src
0
2
2
3
4
4
5
src[0]
src[1]
src[2]
src[3]
src[4]
src[5]
src[6]
Graph
6
4
2
3
0
1
0
1
2
4
3

Fig. 7

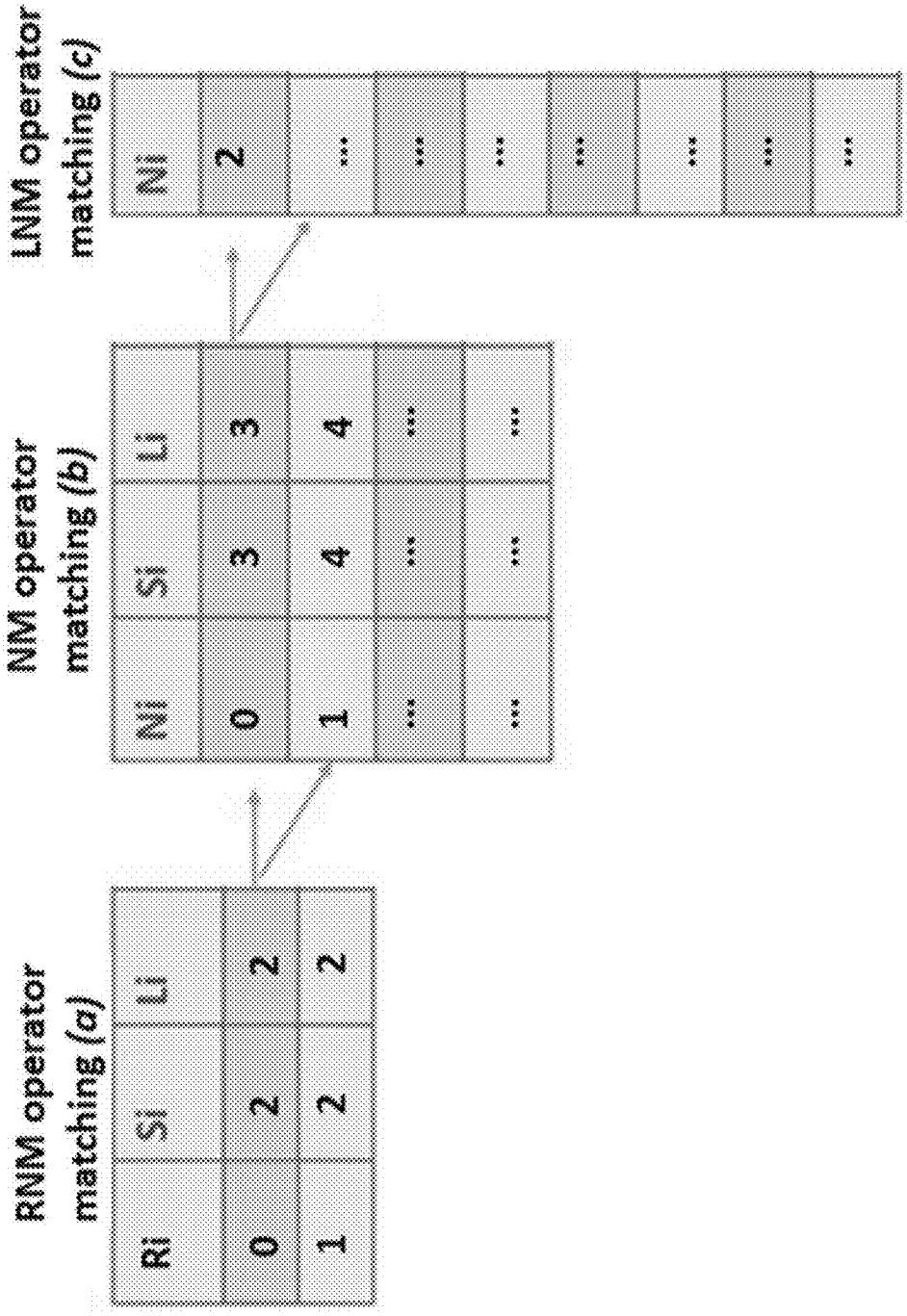

RNM operator matching (a)
Ri
Si
Li
0
2
2
1
2
2
NM operator matching (b)
Ni
Si
Li
0
3
3
1
4
4
...
...
...
LNM operator matching (c)
Ni
2
...
...
...
...
...
...
...

Fig. 8A

Fig. 8B

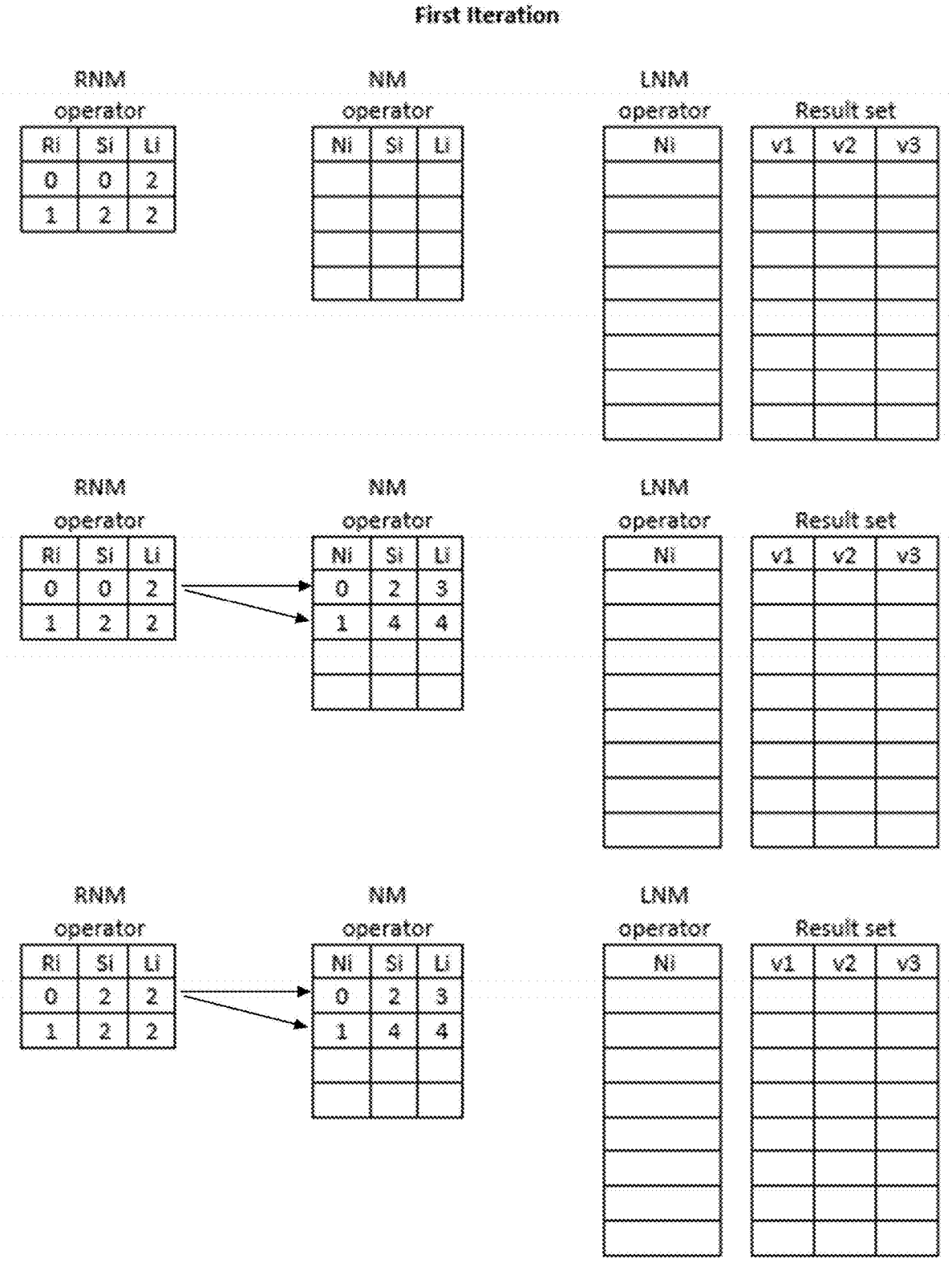

First Iteration
RNM operator
Ri Si Li
0 0 2
1 2 2
NM operator
Ni Si Li
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
0 0 2
1 2 2
NM operator
Ni Si Li
0 2 3
1 4 4
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
0 2 2
1 2 2
NM operator
Ni Si Li
0 2 3
1 4 4
LNM operator
Ni
Result set
v1 v2 v3

Fig. 8C

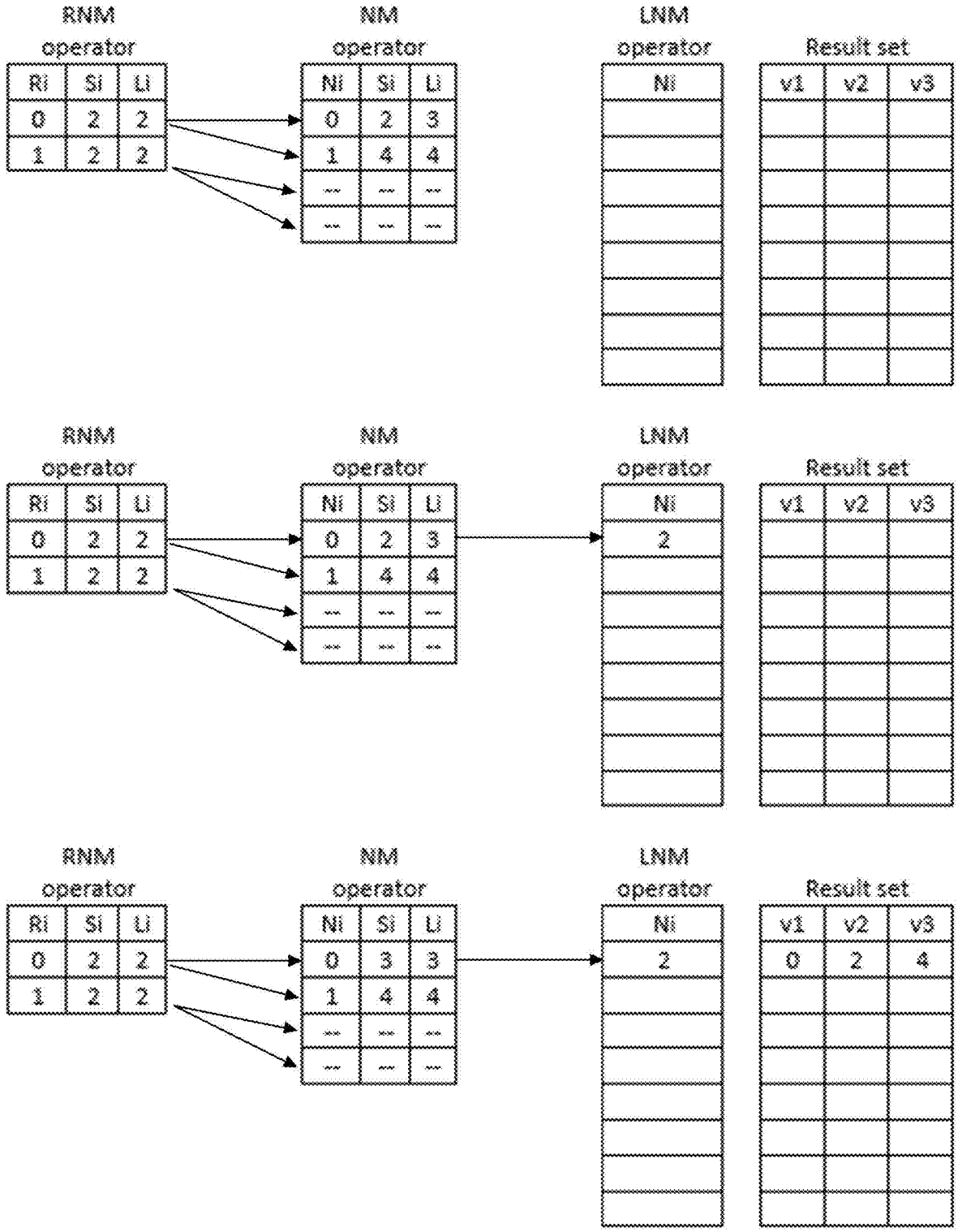

RNM operator
Ri Si Li
0 2 2
1 2 2
NM operator
Ni Si Li
0 2 3
1 4 4
... ...
... ...
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
0 2 2
1 2 2
NM operator
Ni Si Li
0 2 3
1 4 4
... ...
... ...
LNM operator
Ni
2
Result set
v1 v2 v3
RNM operator
Ri Si Li
0 2 2
1 2 2
NM operator
Ni Si Li
0 3 3
1 4 4
... ...
... ...
LNM operator
Ni
2
Result set
v1 v2 v3
0 2 4

Fig. 8D

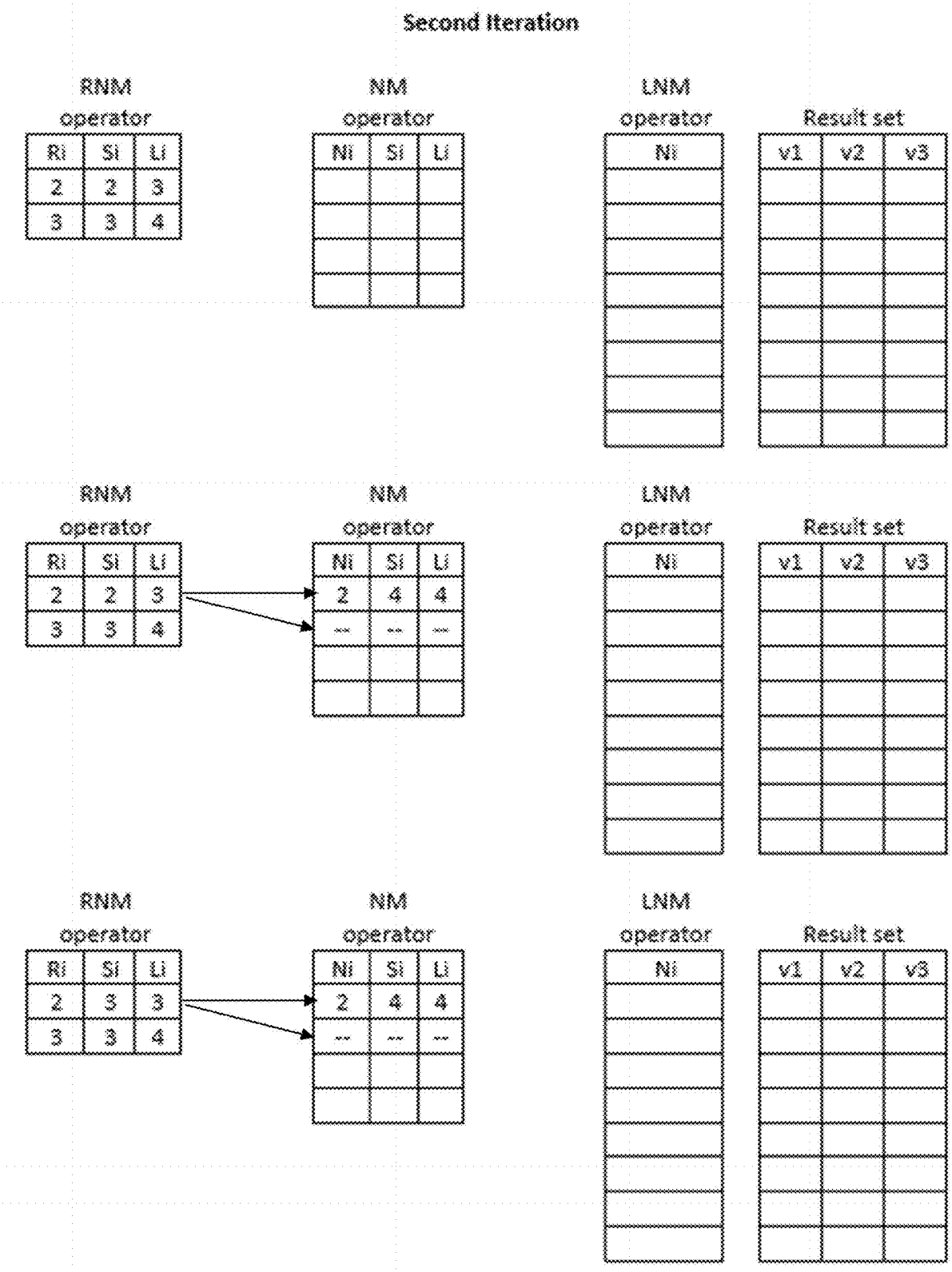

Second Iteration
RNM operator
Ri Si Li
2 2 3
3 3 4
NM operator
Ni Si Li
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
2 2 3
3 3 4
NM operator
Ni Si Li
2 4 4
-- -- --
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
2 3 3
3 3 4
NM operator
Ni Si Li
2 4 4
-- -- --
LNM operator
Ni
Result set
v1 v2 v3

Fig. 8E

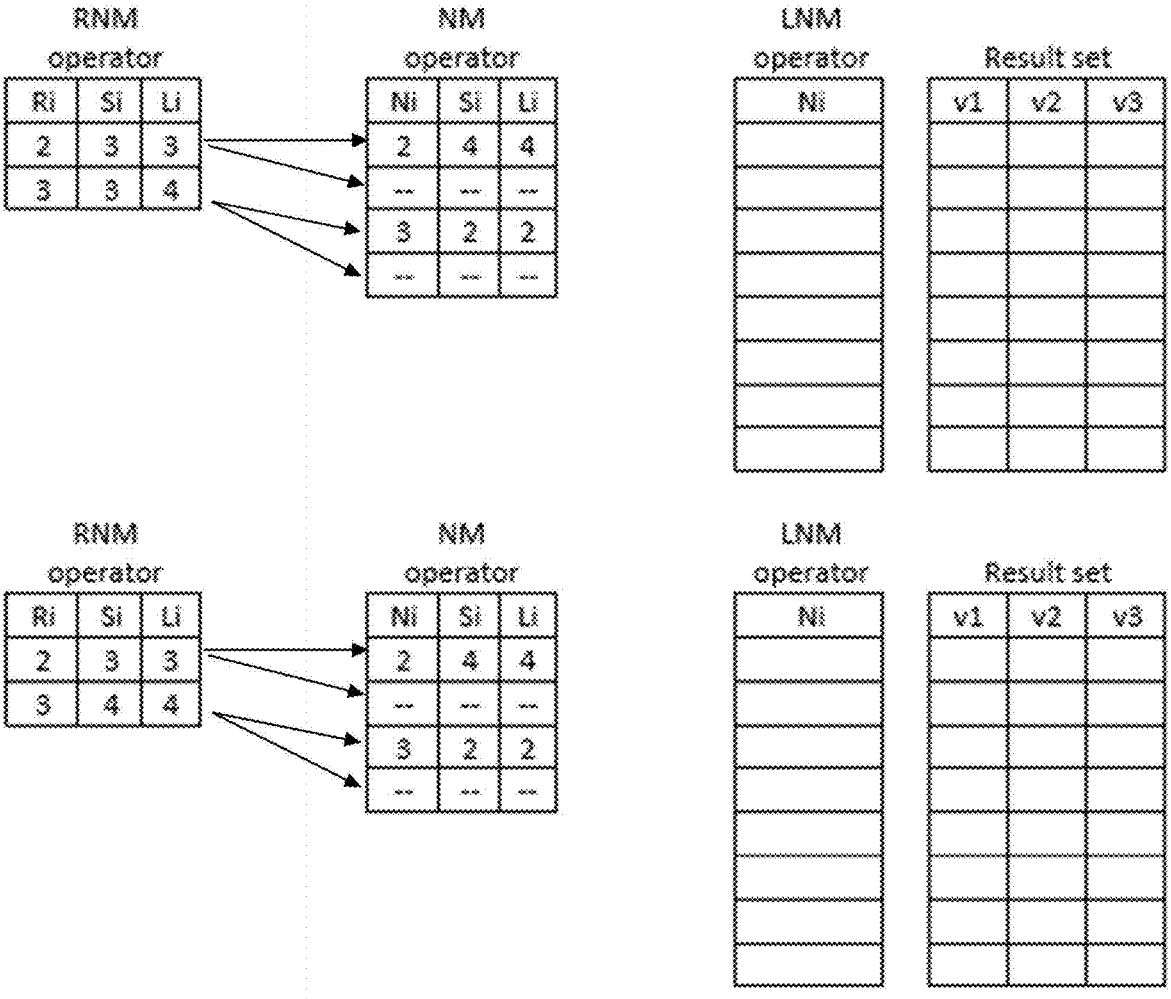

RNM operator
Ri Si Li
2 3 3
3 3 4
NM operator
Ni Si Li
2 4 4
-- -- --
3 2 2
-- -- --
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
2 3 3
3 4 4
NM operator
Ni Si Li
2 4 4
-- -- --
3 2 2
-- -- --
LNM operator
Ni
Result set
v1 v2 v3

Fig. 8F

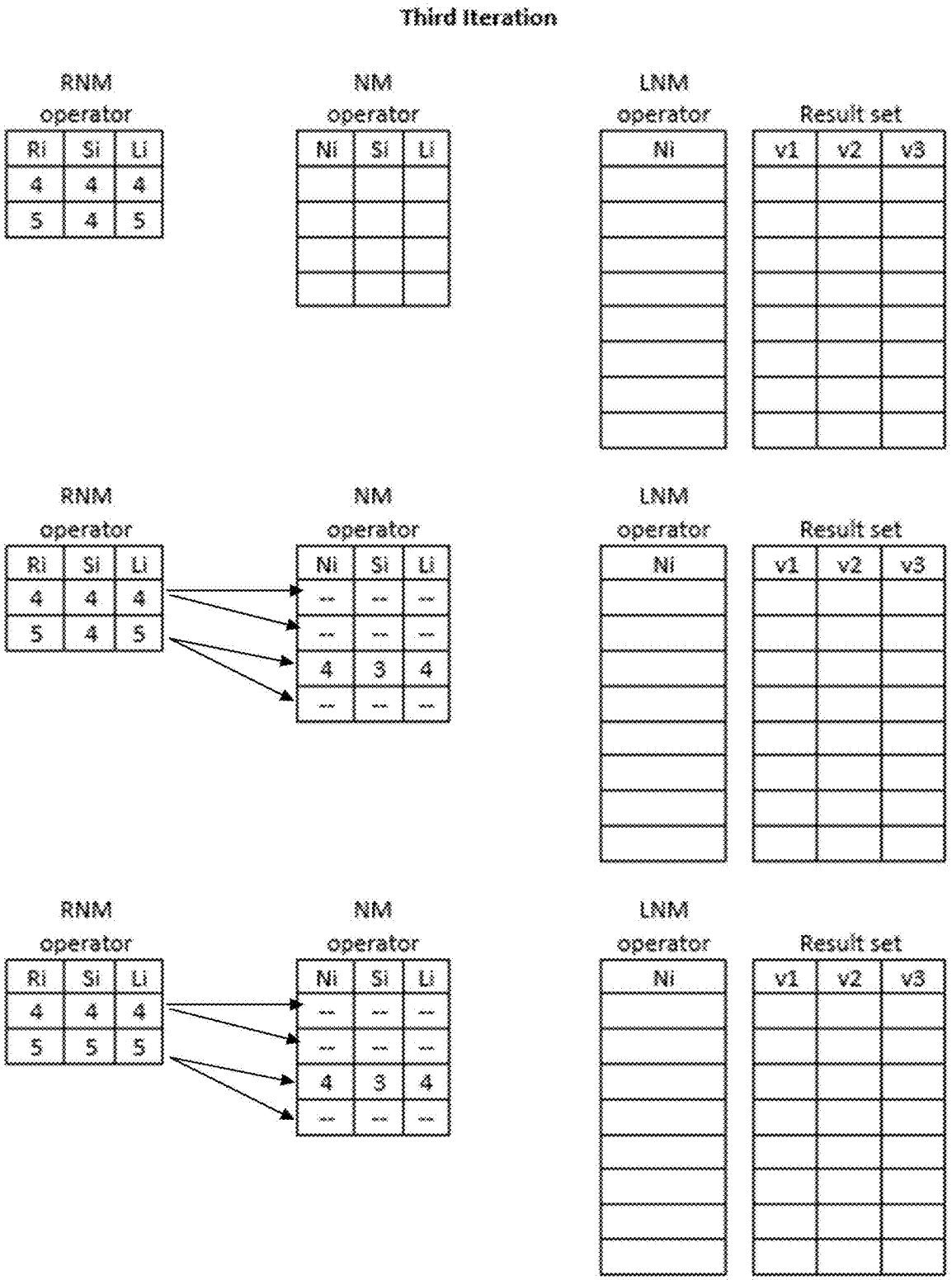

Third Iteration
RNM operator
Ri Si Li
4 4 4
5 4 5
NM operator
Ni Si Li
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
4 4 4
5 4 5
NM operator
Ni Si Li
-- -- --
-- -- --
4 3 4
-- -- --
LNM operator
Ni
Result set
v1 v2 v3
RNM operator
Ri Si Li
4 4 4
5 5 5
NM operator
Ni Si Li
-- -- --
-- -- --
4 3 4
-- -- --
LNM operator
Ni
Result set
v1 v2 v3

Fig. 8G

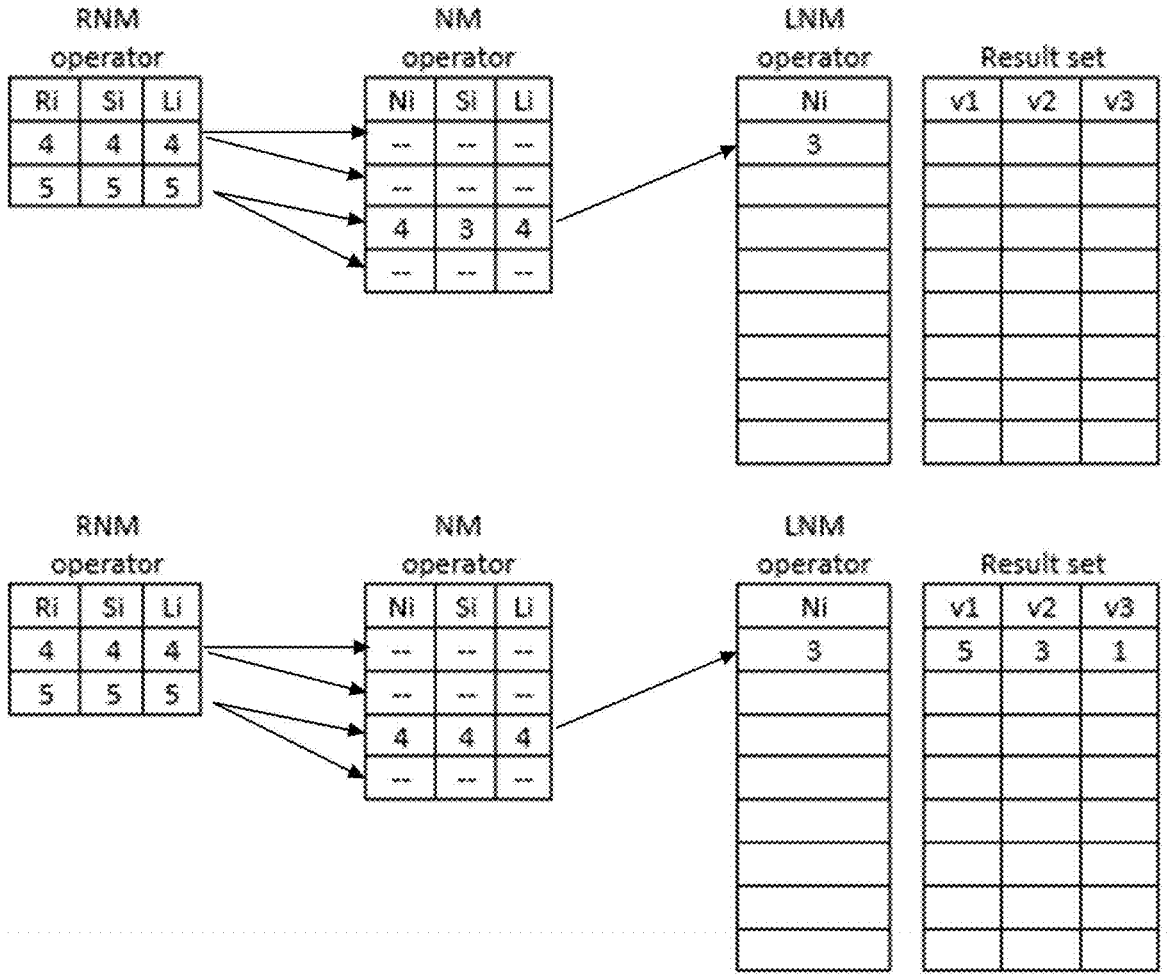

RNM operator
Ri Si Li
4 4 4
5 5 5
NM operator
Ni Si Li
... ...
... ...
4 3 4
... ...
LNM operator
Ni
3
Result set
v1 v2 v3
RNM operator
Ri Si Li
4 4 4
5 5 5
NM operator
Ni Si Li
... ...
... ...
4 4 4
... ...
LNM operator
Ni
3
Result set
v1 v2 v3
5 3 1

Fig. 9

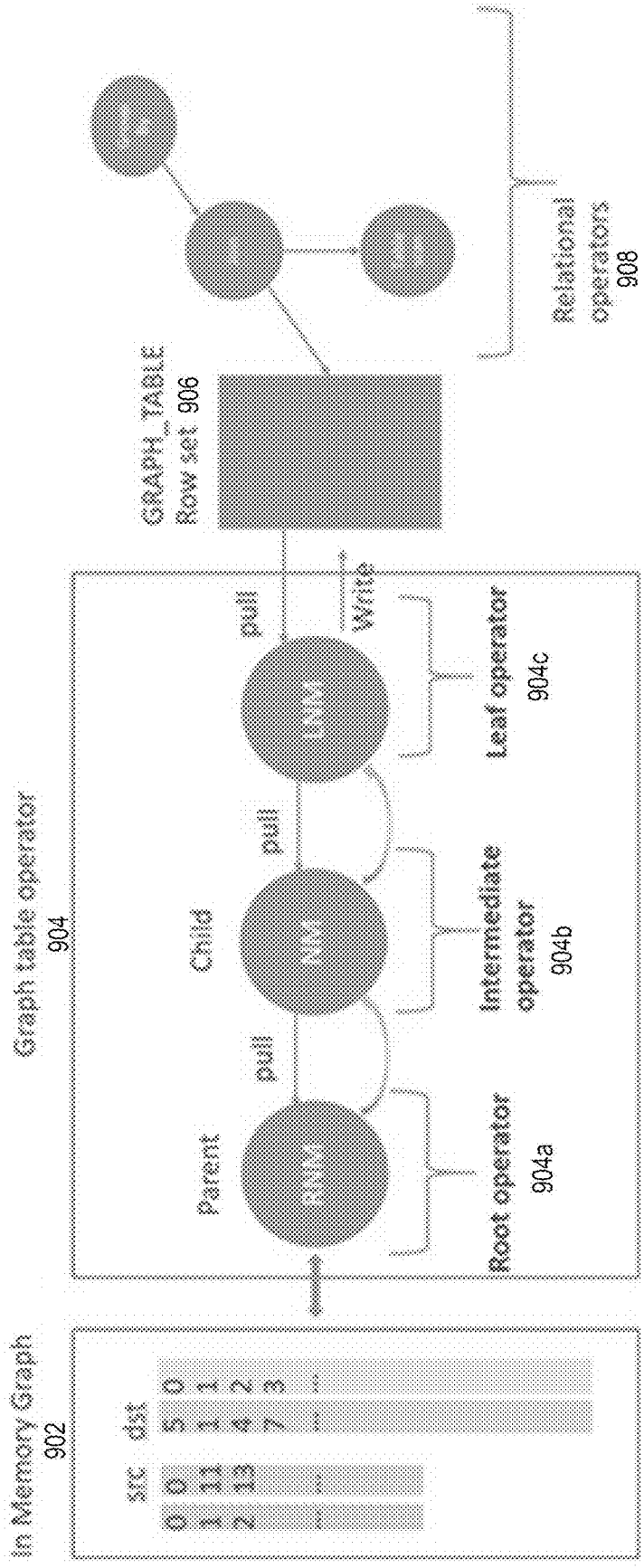

PULL based control flow
In Memory Graph
902
src dst
0 0
1 11
2 13
5 0
1 1
4 2
7 3
Graph table operator
904
Parent
Child
pull
pull
pull
Write
Root operator
904a
Intermediate
operator
904b
Leaf operator
904c
GRAPH_TABLE
Row set 906
Relational
operators
908

Fig. 10

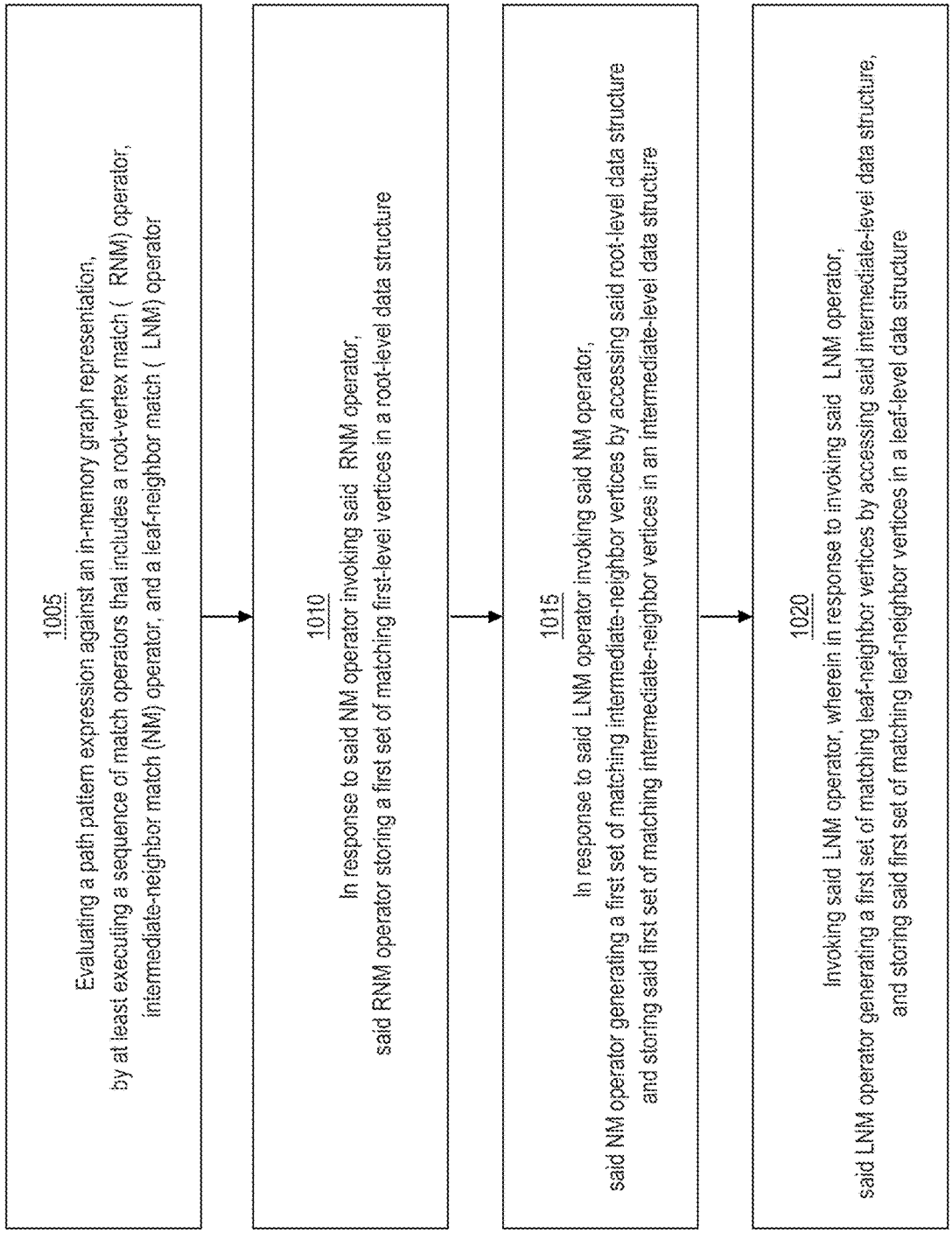

1000
1005
Evaluating a path pattern expression against an in-memory graph representation,
by at least executing a sequence of match operators that includes a root-vertex match (RNM) operator,
intermediate-neighbor match (NM) operator, and a leaf-neighbor match (LNM) operator
1010
In response to said NM operator invoking said RNM operator,
said RNM operator storing a first set of matching first-level vertices in a root-level data structure
1015
In response to said LNM operator invoking said NM operator,
said NM operator generating a first set of matching intermediate-neighbor vertices by accessing said root-level data structure
and storing said first set of matching intermediate-neighbor vertices in an intermediate-level data structure
1020
Invoking said LNM operator, wherein in response to invoking said LNM operator,
said LNM operator generating a first set of matching leaf-neighbor vertices by accessing said intermediate-level data structure,
and storing said first set of matching leaf-neighbor vertices in a leaf-level data structure

Fig. 11A

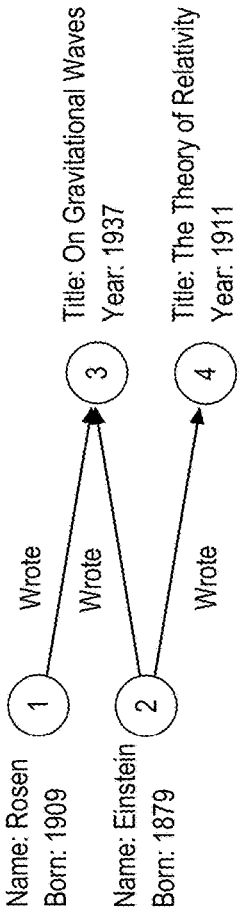


Fig. 11B

Authors Table

Vid	Name	Born
1	Rosen	1909
2	Einstein	1879

Papers Table

Vid	Title	Born
3	On Gravitational Waves	1937
4	The Theory of Relativity	1911

Wrote Table

Eid	AuthorId	PaperId
1	1	3
2	2	3
3	2	4

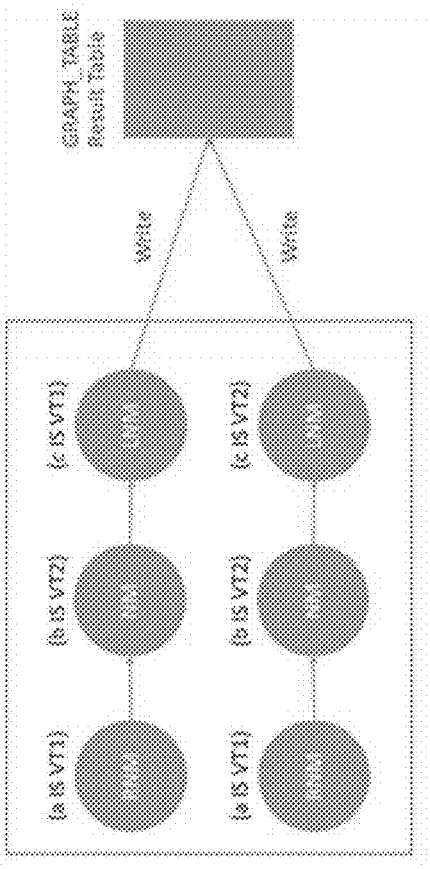


Fig. 12A

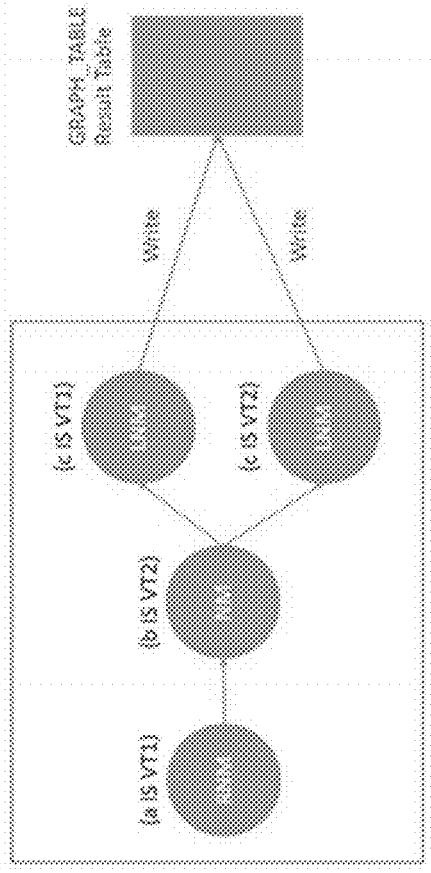


Fig. 12B

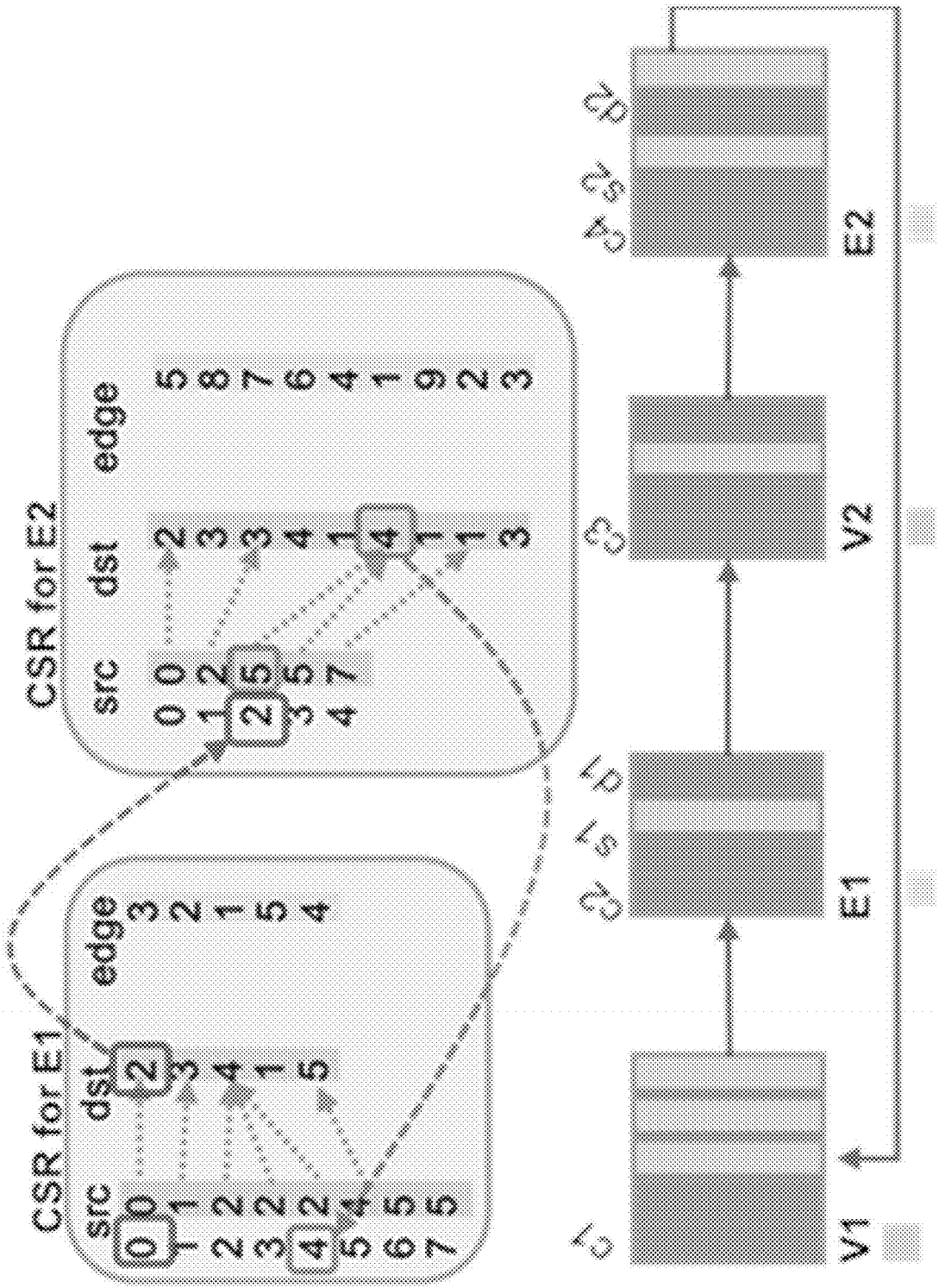

CSR for E1
src
dst
edge
0 1 2 2 2 4 5 5
2 3 4 1 5
3 2 1 5 4
CSR for E2
src
dst
edge
0 0 1 2 3 4 5 6 7
2 5
2 3 3 4 1 4 1 1 3
5 8 7 6 4 1 9 2 3
V1
E1
V2
E2

Fig. 13

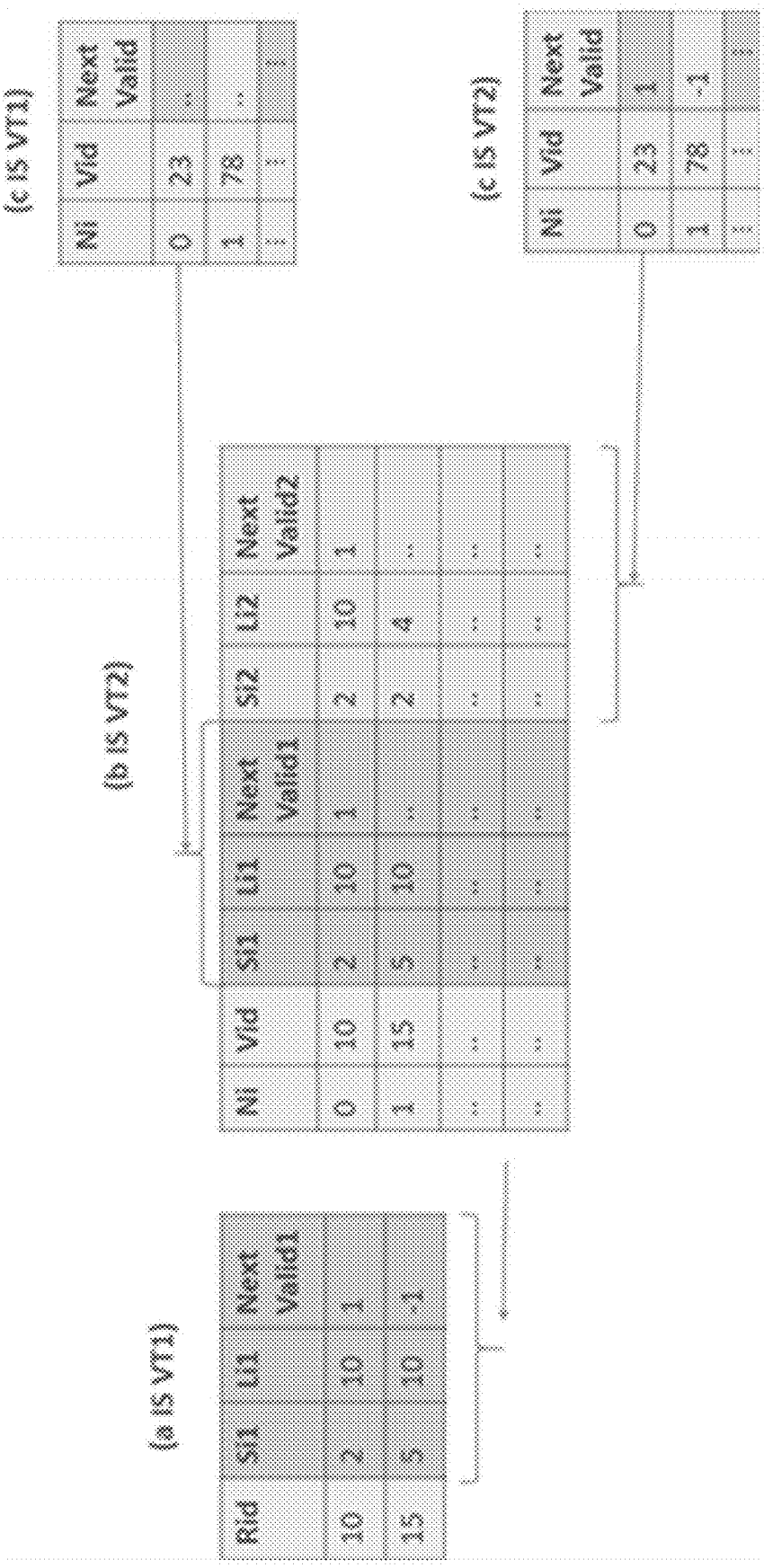

(a IS VT1)
Rid
S1
L1
Next Valid1
10
2
10
1
15
5
10
-1
(b IS VT2)
Ni
Vid
S11
L11
Next Valid1
S12
L12
Next Valid2
0
10
2
10
1
1
15
5
10
-1
2
4
-1
(c IS VT1)
Ni
Vid
Next Valid
0
23
1
1
78
-1
(c IS VT2)
Ni
Vid
Next Valid
0
23
1
1
78
-1

Fig. 14

Fig. 15

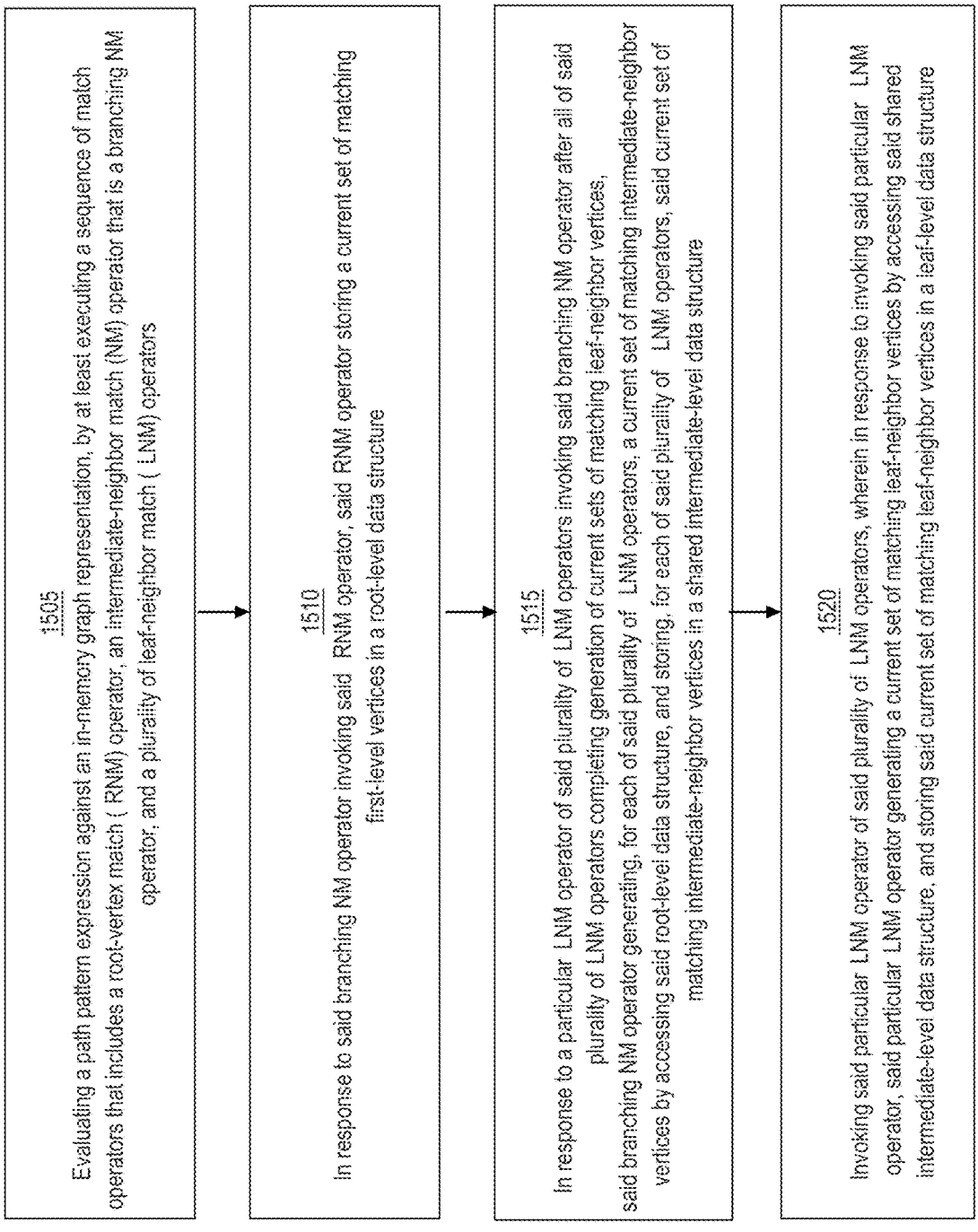
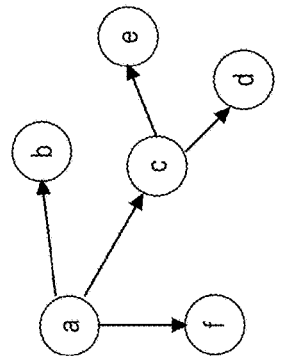

1500
1505
Evaluating a path pattern expression against an in-memory graph representation, by at least executing a sequence of match operators that includes a root-vertex match (RNM) operator, an intermediate-neighbor match (NM) operator that is a branching NM operator, and a plurality of leaf-neighbor match (LNM) operators
1510
In response to said branching NM operator invoking said RNM operator, said RNM operator storing a current set of matching first-level vertices in a root-level data structure
1515
In response to a particular LNM operator of said plurality of LNM operators invoking said branching NM operator after all of said plurality of LNM operators completing generation of current sets of matching leaf-neighbor vertices, said branching NM operator generating, for each of said plurality of LNM operators, a current set of matching intermediate-neighbor vertices by accessing said root-level data structure, and storing, for each of said plurality of LNM operators, said current set of matching intermediate-neighbor vertices in a shared intermediate-level data structure
1520
Invoking said particular LNM operator of said plurality of LNM operators, wherein in response to invoking said particular LNM operator, said particular LNM operator generating a current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure, and storing said current set of matching leaf-neighbor vertices in a leaf-level data structure

Fig. 16A



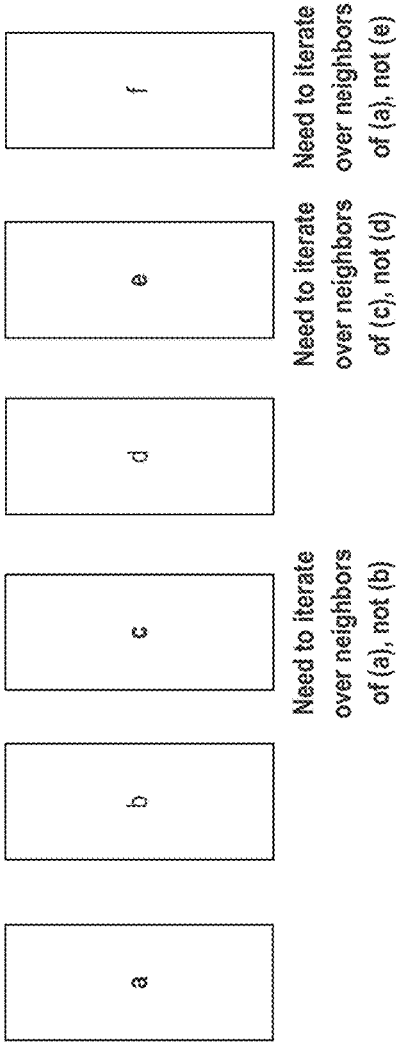


Fig. 16B

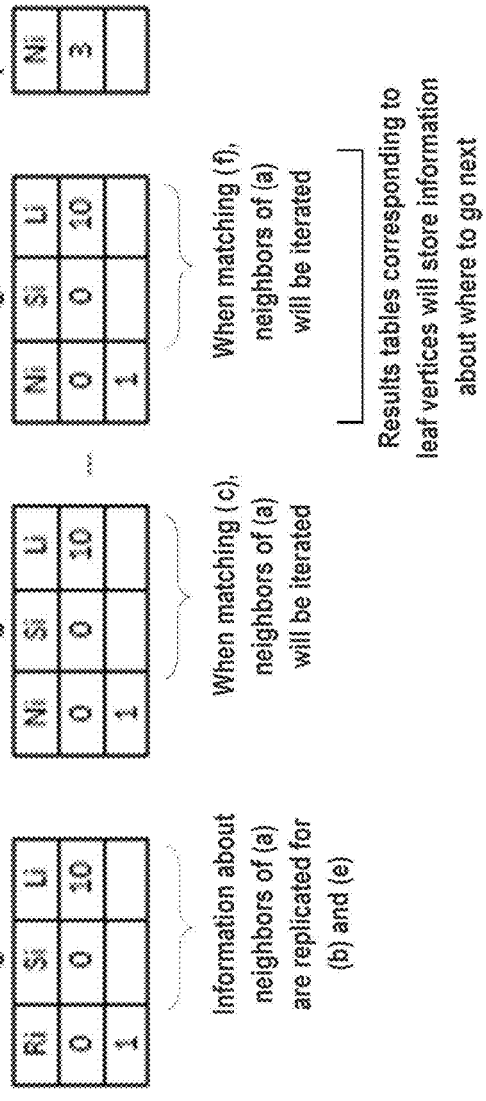


Fig. 17

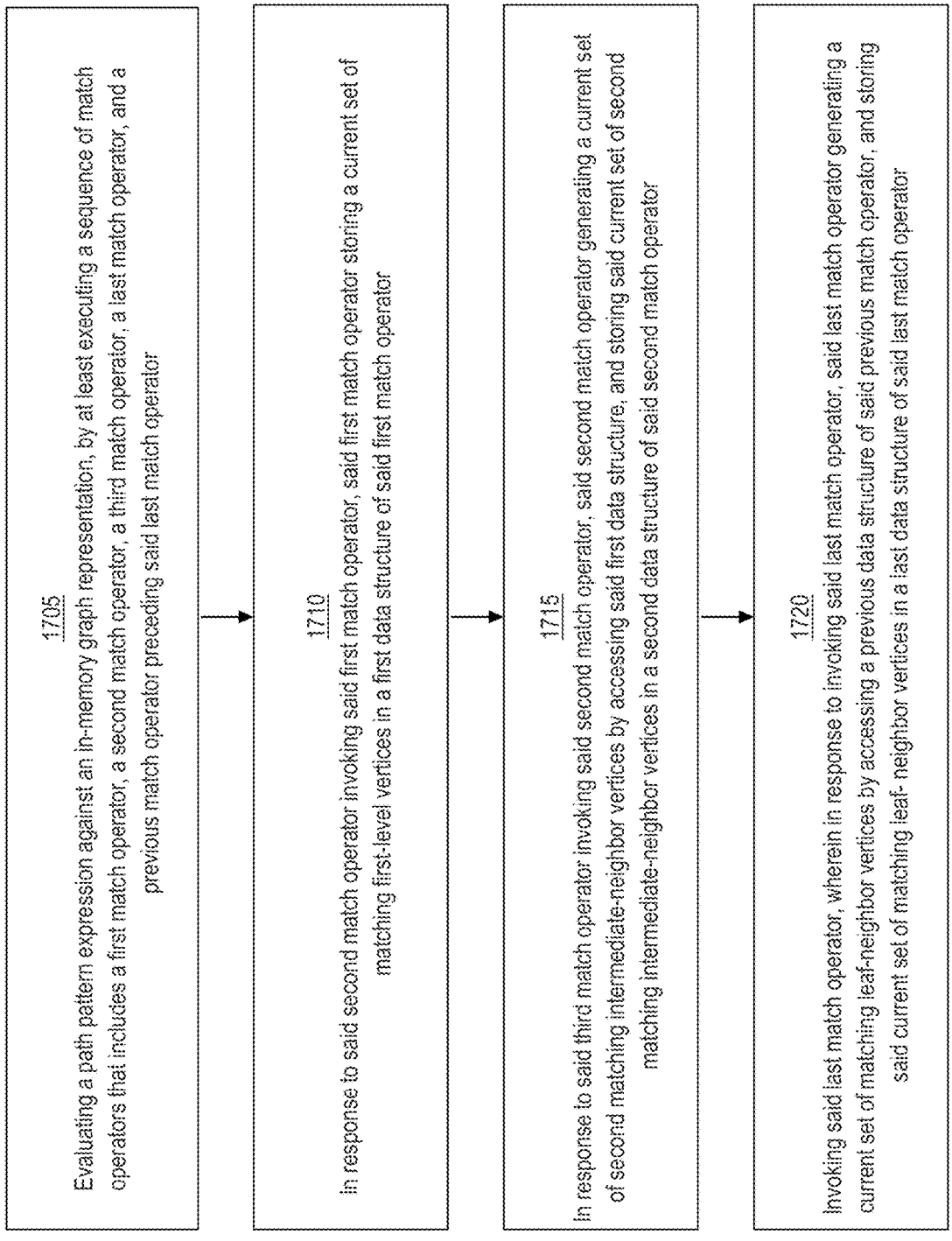

1700
1705
Evaluating a path pattern expression against an in-memory graph representation, by at least executing a sequence of match operators that includes a first match operator, a second match operator, a third match operator, a last match operator, and a previous match operator preceding said last match operator
1710
in response to said second match operator invoking said first match operator, said first match operator storing a current set of matching first-level vertices in a first data structure of said first match operator
1715
in response to said third match operator invoking said second match operator, said second match operator generating a current set of second matching intermediate-neighbor vertices by accessing said first data structure, and storing said current set of second matching intermediate-neighbor vertices in a second data structure of said second match operator
1720
invoking said last match operator, wherein in response to invoking said last match operator, said last match operator generating a current set of matching leaf-neighbor vertices by accessing a previous data structure of said previous match operator, and storing said current set of matching leaf- neighbor vertices in a last data structure of said last match operator

Fig. 18

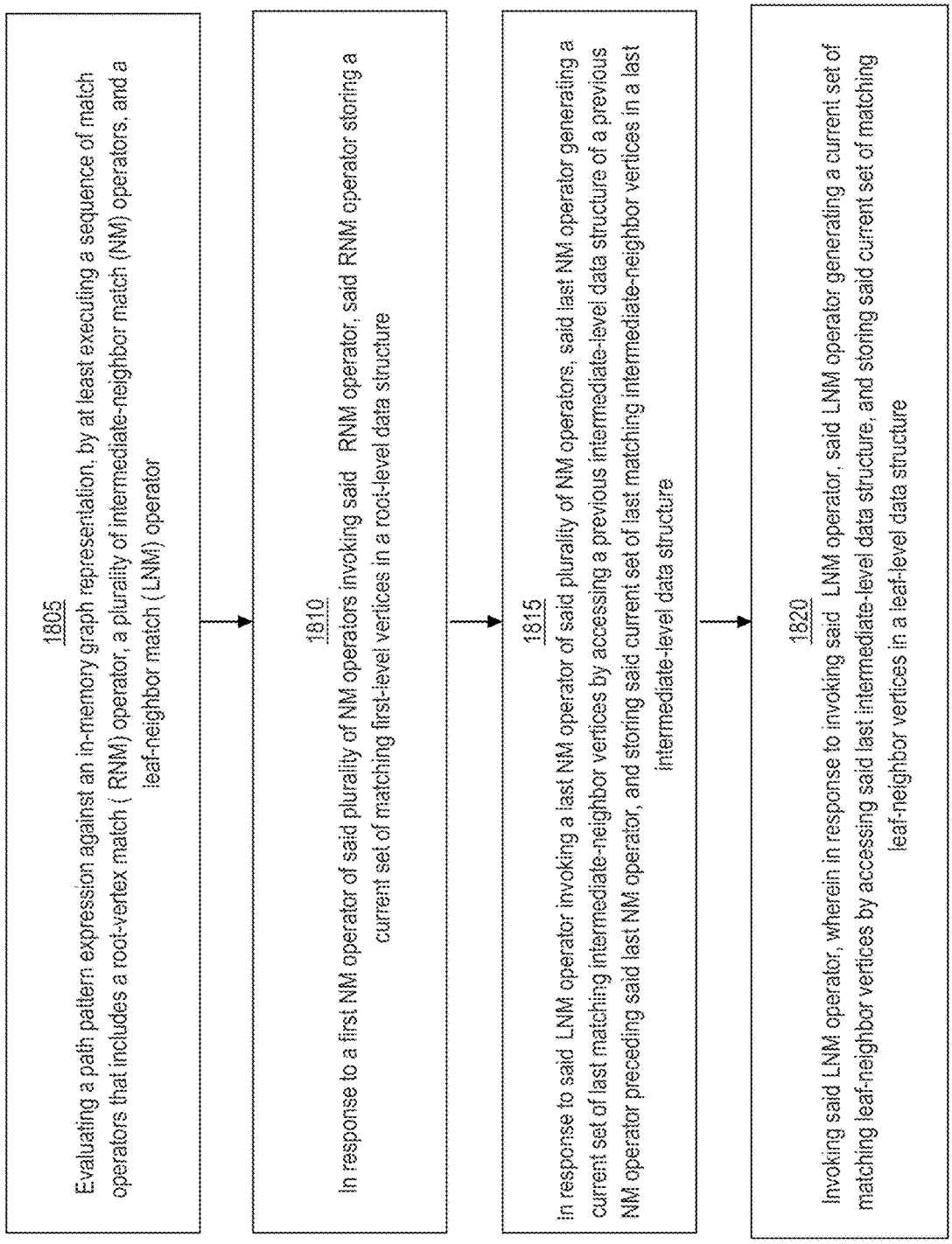
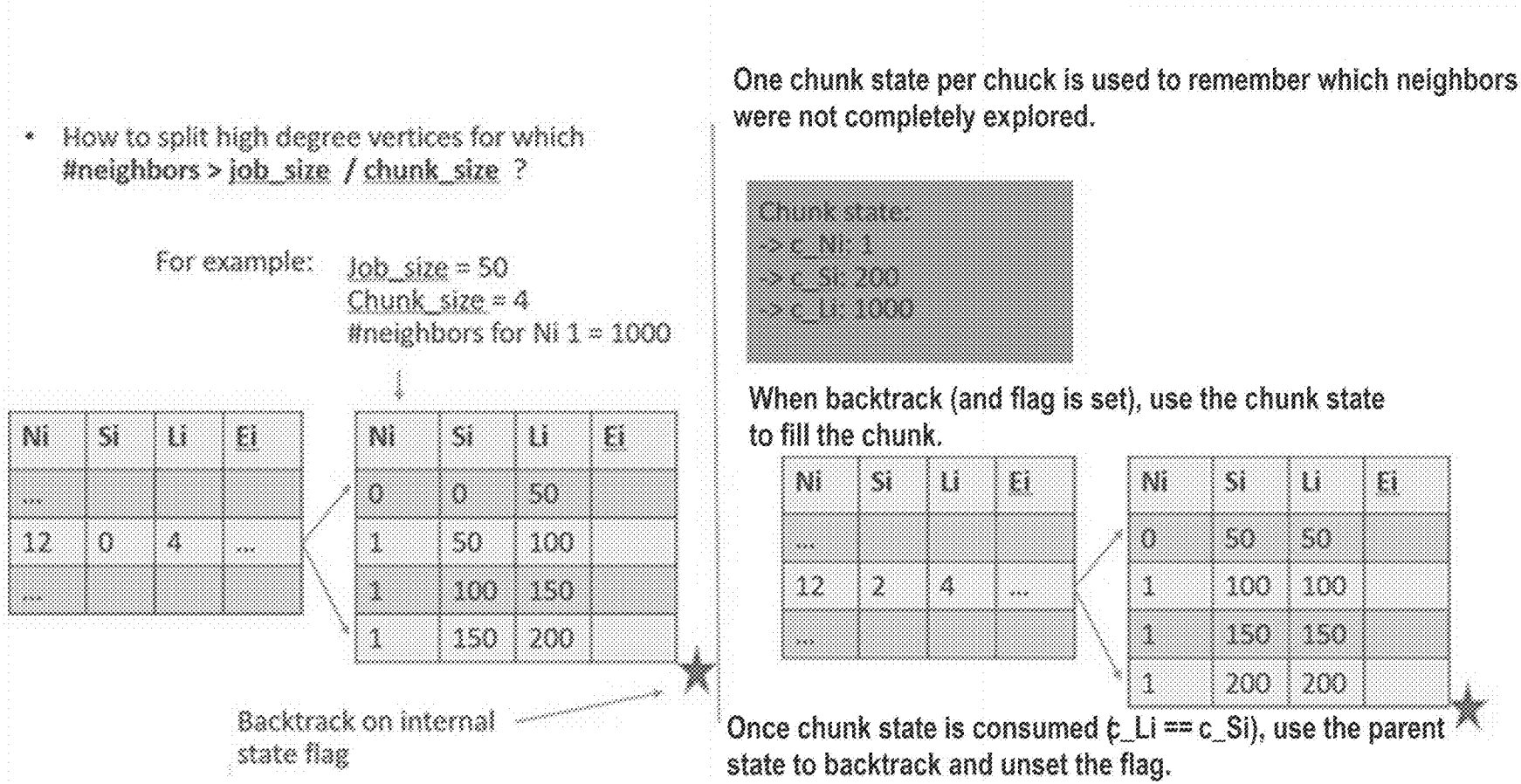

1800
1805
Evaluating a path pattern expression against an in-memory graph representation, by at least executing a sequence of match operators that includes a root-vertex match (RNM) operator, a plurality of intermediate-neighbor match (NM) operators, and a leaf-neighbor match (LNM) operator
1810
In response to a first NM operator of said plurality of NM operators invoking said RNM operator, said RNM operator storing a current set of matching first-level vertices in a root-level data structure
1815
In response to said LNM operator invoking a last NM operator of said plurality of NM operators, said last NM operator generating a current set of last matching intermediate-neighbor vertices by accessing a previous intermediate-level data structure of a previous NM operator preceding said last NM operator, and storing said current set of last matching intermediate-neighbor vertices in a last intermediate-level data structure
1820
Invoking said LNM operator, wherein in response to invoking said LNM operator, said LNM operator generating a current set of matching leaf-neighbor vertices by accessing said last intermediate-level data structure, and storing said current set of matching leaf-neighbor vertices in a leaf-level data structure

Fig. 19


How to split high degree vertices for which #neighbors > job_size / chunk_size ?
For example:
Job_size = 50
Chunk_size = 4
#neighbors for Ni 1 = 1000
Ni Si Li Ei
... ...
12 0 4 ...
...
Ni Si Li Ei
0 0 50
1 50 100
1 100 150
1 150 200
Backtrack on internal state flag
One chunk state per chuck is used to remember which neighbors were not completely explored.
Chunk state:
c_Ni: 1
c_Si: 200
c_Li: 1000
When backtrack (and flag is set), use the chunk state to fill the chunk.
Ni Si Li Ei
... ...
12 2 4 ...
...
Ni Si Li Ei
0 50 50
1 100 100
1 150 150
1 200 200
Once chunk state is consumed (c_Li == c_Si), use the parent state to backtrack and unset the flag.

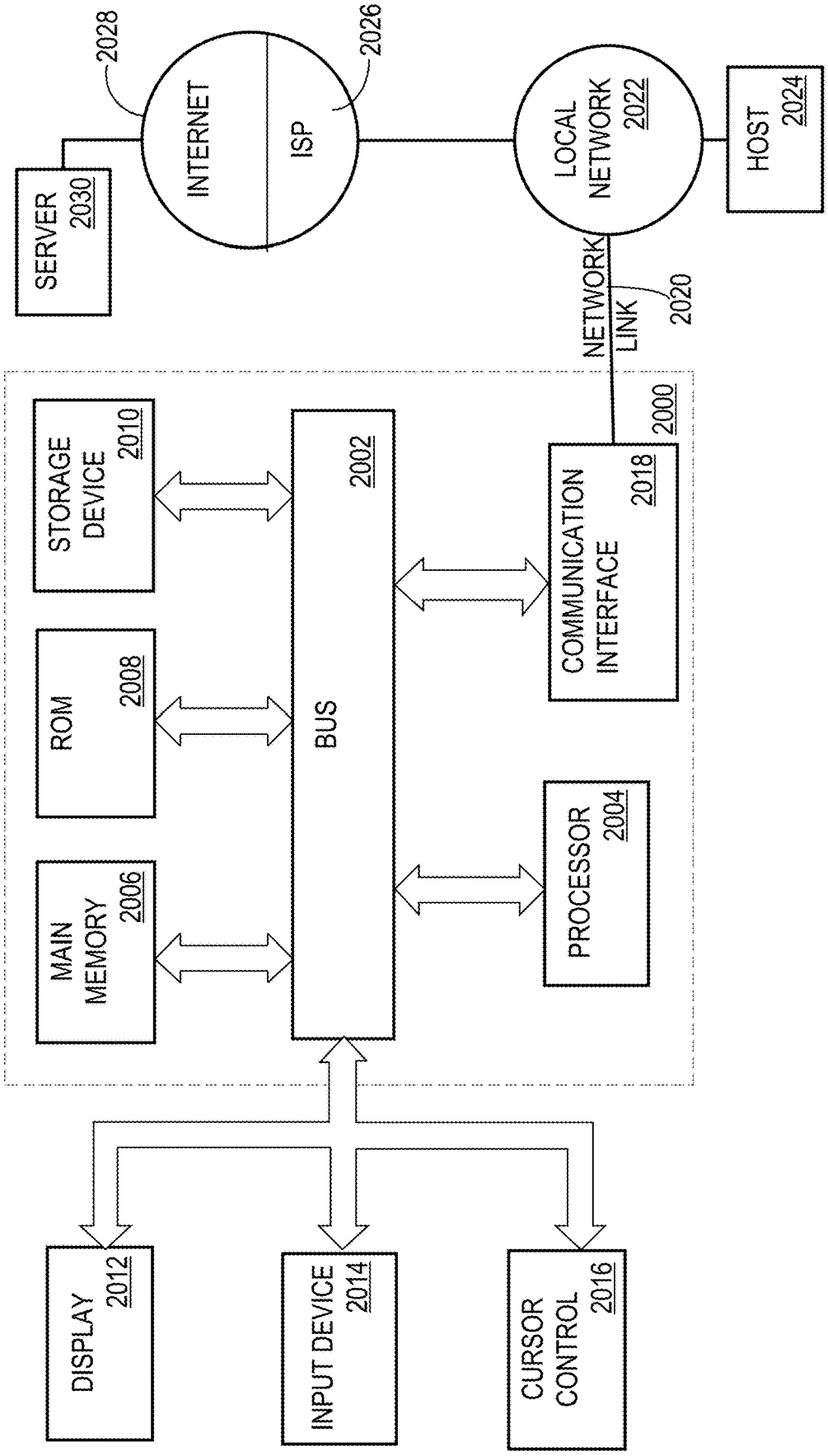
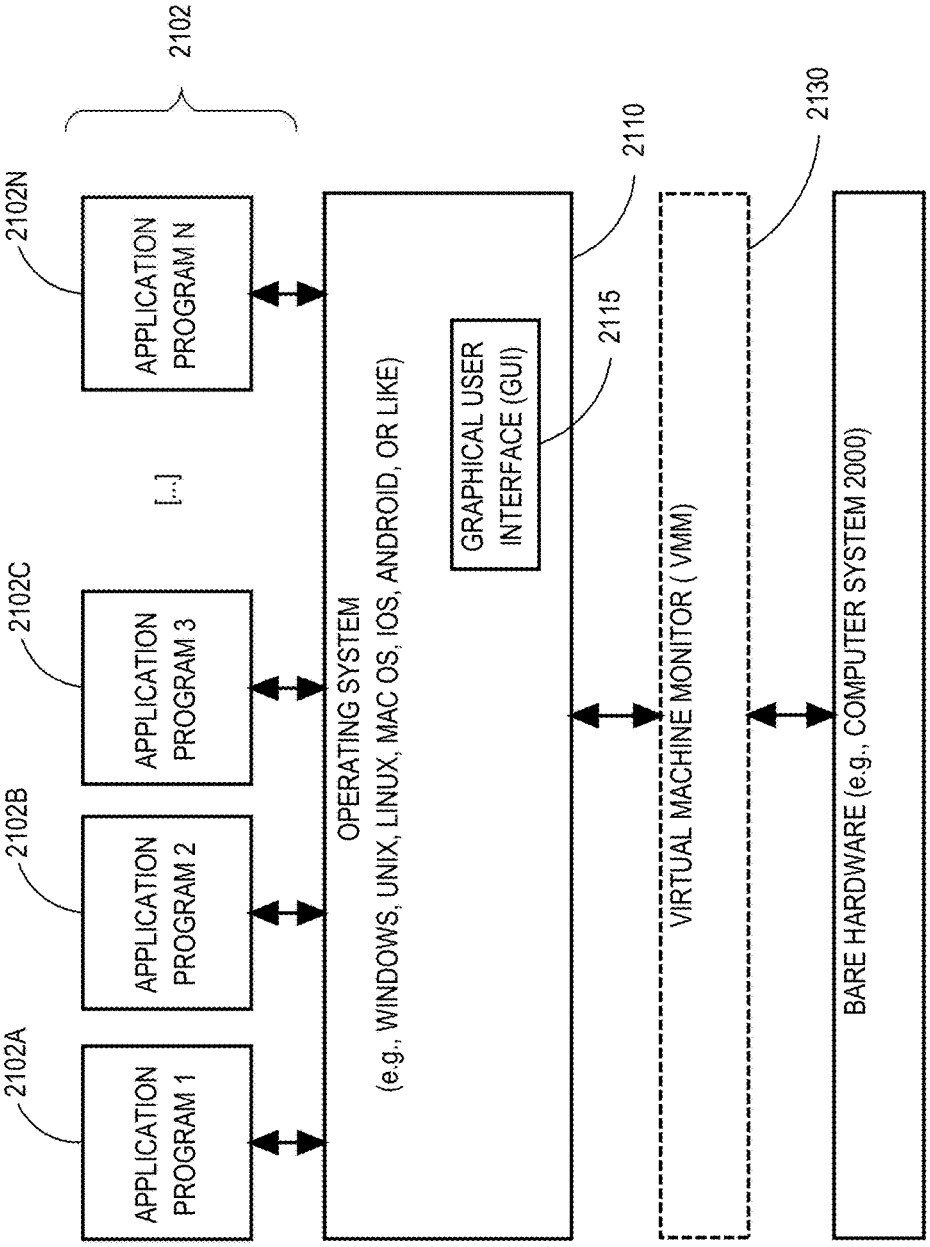

SERVER 2030
2028
INTERNET
ISP
2026
LOCAL NETWORK 2022
HOST 2024
NETWORK LINK 2020
2000
STORAGE DEVICE 2010
ROM 2008
MAIN MEMORY 2006
BUS 2002
COMMUNICATION INTERFACE 2018
PROCESSOR 2004
DISPLAY 2012
INPUT DEVICE 2014
CURSOR CONTROL 2016

Fig. 20

Fig. 21


2100
2102
2102A
APPLICATION PROGRAM 1
2102B
APPLICATION PROGRAM 2
2102C
APPLICATION PROGRAM 3
[...]
2102N
APPLICATION PROGRAM N
OPERATING SYSTEM
(e.g., WINDOWS, UNIX, LINUX, MAC OS, IOS, ANDROID, OR LIKE)
GRAPHICAL USER INTERFACE (GUI)
2115
2110
VIRTUAL MACHINE MONITOR (VMM)
2130
BARE HARDWARE (e.g., COMPUTER SYSTEM 2000)

HYBRID IN-MEMORY BFS-DFS APPROACH FOR COMPUTING GRAPH QUERIES AGAINST HETEROGENEOUS GRAPHS INSIDE RELATIONAL DATABASE SYSTEMS

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is related to U.S. patent application Ser. No. _____, entitled "Hybrid In-Memory BFS-DFS Approach for Computing Graph Queries inside Relational Database Systems," and U.S. patent application Ser. No. _____, entitled "Hybrid In-Memory BFS-DFS Approach for Computing Graph Queries inside Relational Database Systems," wherein the entire contents of which are hereby incorporated by reference as if fully set forth herein.

FIELD OF THE INVENTION

[0002] The present disclosure relates to techniques for graph searches. More specifically, the disclosure relates to a hybrid in-memory BFS-DFS approach for computing graph queries inside relational database systems.

BACKGROUND

[0003] The approaches described in this section are approaches that could be pursued, but not necessarily approaches that have been previously conceived or pursued. Therefore, unless otherwise indicated, it should not be assumed that any of the approaches described in this section qualify as prior art merely by virtue of their inclusion in this section.

[0004] Graph processing is an important tool for data analytics. Relational DBMSs increasingly allow their users to define property graph from relational tables and to query them using graph pattern matching queries. Most products limit users to define a graph out of a single vertex table and edge table (e.g., Microsoft SQL Server, SAP Hana). These graphs are called homogeneous graphs. The most advanced systems may define a graph out of multiple vertex and edge tables. For example, a heterogeneous graph may be created out of the existing tables in a database by mapping every dimension table to a vertex table and every fact table to an edge table. The only constraint is that vertex tables should have a primary key column and the edge tables should associate foreign keys corresponding to the primary keys into two different/same vertex tables.

[0005] Graph querying and pattern matching enables interactive exploration of graphs, similar to how SQL interacts with databases. Pattern matching refers to finding patterns in the graph that are homomorphic to a target pattern, such as a triangle. In addition to the structural pattern, the user can add projections, filters, etc., similar to SQL.

[0006] Graph queries, which refer to "graph querying and pattern matching," are a very challenging workload as they put the focus on edges, i.e., the connections in the data. Therefore, executing graph queries might explore immense amounts of intermediate results and queries can quickly explode in terms of memory usage. Additionally, graph queries exhibit very irregular access patterns with limited memory locality, as the query patterns and the connections in the data dictate the accesses.

[0007] One way that relational DBMSs implement graph queries is by translating the graph query into an SQL join query and processing it with their existing SQL engine. This approach is suboptimal in two ways.

[0008] In the case of homogeneous graphs, SQL engines do not leverage the graph structure when doing neighbor traversals. In other words, when iterating over the neighbors of a vertex V, a join operator will have to scan through all possible vertices and check if they are connected to vertex V. More advanced join operators may use a hash-join operation that will build in memory a hash table from the smallest of the table and, then, probe that table for matching element when scanning the largest table once. As such, the "neighbor" scan is often not performed. However, the construction of the hash table is repeatedly performed for every join in the query and for every query.

[0009] In the case of heterogeneous graphs, most SQL engines do not take advantage of sharing computation on common overlapping prefixes across paths belonging to multiple instantiations of a given path pattern.

[0010] Most of the existing specialized graph processing engines leverage the graph structure encoded in the data and use one of the two main traversal approaches when dealing with graph queries: breath-first traversal (BFS) and depth-first traversal (DFS). In BFS, all the vertices at a given level are matched before processing the next level. In DFS, a vertex at one level is processed at one level before going to the next level. Once the whole pattern is matched, the path is backtracked to iterate over more vertices.

[0011] These traversal techniques are not directly suitable for being integrated into a relational database systems for several reasons. First, most query execution plan operators ("plan operator") work in a pipeline. As soon as a plan operator produces a set of tuples, it sends them to the next plan operator in order to be processed to materialize the whole result before going able to go to the next pipeline stage. From this point of view, the graph processing runtime should be pipeline friendly and the results should be produced as fast as possible. BFS execution mode would not satisfy this as it needs to produce all the results before being able to ship them to the next plan operator.

[0012] In addition, as databases are deployed in multi-user scenarios, memory footprint of every query is essential for maximizing the number of concurrent clients. Thus, the graph processing runtime should have a low memory footprint. BFS execution mode would not satisfy this as it materializes all intermediate results, possibly monopolizing the memory of the machine.

[0013] These problems could be solved by using a DFS exploration approach, which produces results as soon as the first path is matched and has a very small memory footprint. However, the issue with DFS exploration is that it breaks locality when iterating over neighbors as it looks only at a single neighbor at a time.

[0014] Therefore, there is a need for a solution to address these challenges found inside a single machine relational database system when processing graph queries.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] The example embodiment(s) of the present invention are illustrated by way of example, and not in way by limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0016] FIG. **1** illustrates an example graph and an example target pattern.

[0017] FIG. **2**A illustrates an example property graph.
[0018] FIG. **2**B illustrates example tables associated with the graph of FIG. **2**A.
[0019] FIG. **3** illustrates an example graph and a corresponding CSR representation thereof.
[0020] FIG. **4** illustrates an example sequence of match operators.
[0021] FIG. **5** illustrates example data structures of two consecutive match operators.
[0022] FIG. **6** illustrates example data structures for storing match results of the graph table operator.
[0023] FIG. **7** illustrates another example graph and corresponding CSR representation thereof.
[0024] FIG. **8**A illustrates the state of match operators after one iteration.
[0025] FIGS. **8**B-**8**G illustrate the state of match operators during each iteration.
[0026] FIG. **9** is an example for a pull-based control flow.
[0027] FIG. **10** illustrates another example flow diagram that depicts a process **1000** for supporting graph pattern matching queries inside a relational database system.
[0028] FIG. **11**A illustrates another example property graph.
[0029] FIG. **11**B illustrates example tables associated with the graph of FIG. **11**A.
[0030] FIGS. **12**A-**12**B illustrate different instantiations of a heterogeneous operator path.
[0031] FIG. **13** illustrates an example of a heterogeneous graph in-memory representation.
[0032] FIG. **14** illustrates an example of a possible state of the graph table operators.
[0033] FIG. **15** illustrates another example flow diagram that depicts a process for supporting graph pattern matching queries inside a relational database system.
[0034] FIG. **16**A illustrates an example tree pattern to compute and a set of data structures corresponding to vertices in the tree pattern.
[0035] FIG. **16**B illustrates example data structures for storing match results for the tree pattern of FIG. **16**A.
[0036] FIG. **17** illustrates another example flow diagram that depicts a process for supporting graph pattern matching queries inside a relational database system.
[0037] FIG. **18** illustrates yet another example flow diagram that depicts a process for supporting graph pattern matching queries inside a relational database system.
[0038] FIG. **19** illustrates a parallel execution example.
[0039] FIG. **20** illustrates a block diagram of a computing device in which the example embodiment(s) of the present invention may be embodiment.
[0040] FIG. **21** illustrates a block diagram of a basic software system for controlling the operation of a computing device.

DETAILED DESCRIPTION

[0041] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

GENERAL OVERVIEW

[0042] An in-memory graph query runtime is integrated inside a database management system (DBMS) and is capable of performing pattern-matching queries against graphs of different complexities. The runtime efficiently combines breadth-first (BFS) and depth-first (DFS) neighbor traversal algorithms to achieve a hybrid runtime that takes the best from both algorithms. As a result, the hybrid runtime is able to process arbitrarily large queries with a fixed amount of memory, optimizing for memory locality. A tremendous advantage of being integrated in the relational database is that the runtime can be pipelined with the rest of already-existing operators. Another advantage is that data need not be exported from databases in external engines in order to execute graph queries.

BACKGROUND IN GRAPH PROCESSING SYSTEMS

[0043] A graph is a mathematical structure used to model relationships between entities. A graph consists of a set of vertices (corresponding to entities) and a set of edges (corresponding to relationships). When data for a specific application has many relevant relationships, the data may be represented by a graph.
[0044] Graph processing systems can be split in two classes: graph analytics and graph querying. Graph analytics systems have a goal of extracting information hidden in the relationships between entities, by iteratively traversing relevant subgraphs or the entire graph. Graph querying systems have a different goal of extracting structural information from the data, by matching patterns on the graph topology.

BACKGROUND IN GRAPH PATTERN MATCHING

[0045] Graph pattern matching refers to finding subgraphs, in a given directed graph, that are homomorphic to a target pattern. FIG. **1** illustrates an example graph and an example target pattern. If the target pattern is (a)→(b)→(c)→(a), then corresponding graph walks or paths of FIG. **1** are the following:
[0046] (1)→(2)→(3)→(1),
[0047] (2)→(3)→(1)→(2), and
[0048] (3)→(1) →(2)→(3).
One hop corresponds to a graph walk consisting of a single edge. A walk with n edges is considered as a n-hop pattern.
[0049] As discussed above, three approaches may be used for performing pattern matching in graph systems. A first approach includes operations similar to JOINs in relational DBMSs. To match a hop in the pattern, the system performs a JOIN with the previous intermediate results. The disadvantage of this approach is that it does not exploit the graph structure by iterating only over the neighbors of a node but iterates over all possible nodes when performing the JOIN.
[0050] A second approach is a BFS neighbor traversal. This approach generates all the matches for all possible pattern prefixes, starting from one single node until reaching the desired pattern. This algorithm works in an iterative manner: having all the matches for a prefix pattern, it computes all the matches for the prefix extended with an extra hop. The advantage over the first approach is that the system keeps track of the neighbors of each node and performs a scan only over the neighbors of a node when it

is needed. However, it still has the disadvantage of possibly using a big amount of memory for the intermediate results.

[0051] A third approach is a DFS neighbor traversal. With this approach, the system matches full paths and does not generate intermediate results. This algorithm does not need to store all the matches for each pattern prefixes but has the drawback in that it does not optimize for memory locality while traversing over the neighbors of each node because only one node neighbor is considered at each step.

[0052] Embodiments of the present invention relate to a hybrid traversal to perform pattern matching. The hybrid traversal is similar to the BFS neighbor traversal, but the hybrid traversal only explores a limited number of neighbors in one iteration, following to resume later to process the remaining neighbors. This algorithm has the advantage in that it optimizes for memory locality by processing several neighbors in one iteration, while still restricting the used memory by limiting the number of neighbors processed at a specific step.

INTEGRATING GRAPH AND RELATIONAL DATABASES

[0053] Techniques, described herein, use RAM memory only for storing a graph topology and keep the rest of the data on disk. In particular, the techniques use the in-memory acceleration for graph traversals, without storing redundant data in memory. Additionally, techniques use existing database tables as vertex or edge tables and can process different types of graphs, including homogeneous graphs and heterogeneous graphs.

TRANSLATION FROM RELATIONAL MODEL TO PROPETY GRAPH MODEL

[0054] A property graph data model allows vertices and edges in a graph to have arbitrary properties as key-value pairs. FIG. **2**A illustrates an example property graph. The property graph in FIG. **2**A represents contact relationships between persons. The persons have name and age properties, while the relationships have a property indicting the age at which the subject knew the other person.

[0055] Conceptually, a relational DBMS may be interpreted in a graph model. For example, all vertices and their properties may be grouped in a vertex table, while all relationships and their properties may be grouped in an edge table. As an example, the graph illustrated in FIG. **2**A may be stored in a RDBMS as tables, as illustrated FIG. **2**B. Each vertex has a unique identifier (i.e., Vid) in the vertex table (i.e., Persons Table) where it is stored. Similarly, each edge has a unique identifier (i.e., Eid) in the edge table (i.e., Knows Table) where it is stored. The edge table can be used for matching patterns; the vertex and edge tables can then be used for accessing the desired information when the pattern is matched. The RDBMS keeps track of the referenced tables by each edge table, in order to be able to reconstruct the graph structure.

REPRESENTING GRAPHS IN MEMORY

[0056] In an embodiment, a graph may be represented by compressed sparse rows (CSR). This representation may be the most useful in graph processing engines because of its compact size and its neighbor caching behavior. The CSR compression is performed over the adjacency matrix of the graph.

[0057] The CSR representation for a graph G with n vertices labeled from 0 to n−1, and m edges, labeled from 0 to m−1, includes two arrays, src and dst. FIG. **3** illustrates an example graph and a corresponding CSR representation thereof. The graph in FIG. **3** includes 10 vertices and 11 edges.

[0058] Each vertex in the graph is associated with a unique identifier allocated from a set of numbers. The unique identifier behaves as the src array index. The vertex id is referred to herein as Vid. The src array will have size n+1. For every $i \in \{0, 1, \ldots, n-1\}$, the value stored in src[i] will be the position in the dst array for the first neighbor of vertex i. The last position of src array stores the value m, i.e., src[n]=m.

[0059] Similarly, each edge in the graph is associated with a unique identifier that behaves as the dst array index. The edge identifier is referred to herein as Ni (neighbor index). The dst array will have size m. For every $i \in \{0, 1, \ldots, n-1\}$, the neighbors of vertex i are stored at the positions between src[i] and src[i+1], meaning that all the neighbors of a vertex are stored consequently, which helps while traversing all the neighbors of a specific vertex.

[0060] As an illustration, in the example of FIG. **3**, to find the first neighbor of vertex **1**, the value at src array index **1** of the src array (e.g., src[**1**]) is obtained. The value at src[**1**], which is four (4), identifies edge **4** as being the first edge from vertex **1**. Referring to dst array, the value at dst array index **4** of the dst array (e.g., dst[**4**]) is then obtained. The value dst[**4**], which is five (5), identifies vertex **5** as being the vertex from edge **4**. Accordingly, vertex **5** is the first neighbor of vertex **1**.

[0061] Also, as an illustration, in the example of FIG. **3**, to find all neighbors of vertex **3**, the value at src array index **3** and the value at scr array index **4** of the scr array (e.g., scr[**3**] and scr[**4**]) are obtained. The difference between the two values at scr[**3**], which is five (5), and at scr[**4**], which is nine (9), indicates the number of neighbors vertex **3** has. In this illustration, vertex **3** has a total of four (4) neighbors. The value at scr[**3**] identifies edge **5** as being the first edge from vertex **3**. Referring to dst array, four (4) values starting at dst array index **5** to dst array index **8** of the dst array (e.g., dst[**5**], dst[**6**], dst[**7**], and dst[**8**]) are then obtained. The values at dst[**5**] to dst[**8**], which are six (6) to nine (9), identifies vertex **6** to vertex **9** as being vertices from edges **5** to **8**. Accordingly, vertex **6** to vertex **9** are the four (4) neighbors of vertex **3**.

[0062] Since the source vertex for edges that have the same source is stored once, the CSR representation uses about half the space needed by representations operating on edge lists. Thus, CSR encoding is a good candidate for in-memory graph representations because of the small memory footprint required. In addition, by storing all the neighbors of a vertex in consecutive memory locations, it is a strong fit for tasks which need to iteratively process a vertex's neighbors, such as pattern matching or graph algorithms.

IN-MEMORY GRAPH QUERY RUNTIME

[0063] In an embodiment, an in-memory graph query runtime may be integrated inside a RDBMS to improve efficiency of graph queries. The query runtime may be a simplified version of an execution engine that includes a set of operators, a set of data structures that hold corresponding states and execution mode, including control flow (e.g., how

the operators call each other), and data flow (e.g., how data flows between the operators. The runtime has access to a CSR that is in memory for optimizing a graph pattern matching operation and to graph properties that are on disk for reducing memory consumption. In an embodiment, the graph query runtime supports simple path expressions, such as line-shaped path patterns (e.g., $(a_1)\rightarrow(a_2)\rightarrow \ldots \rightarrow(a_n)$, for any $n\geq 2$).

[0064] The graph query runtime is represented in the RDBMS as a graph table operator, which allows pattern matching queries containing filters and projections on graph properties to be written. The graph table operator transforms result of a graph pattern matching operation into a table, which will have the same number of rows as the number of subgraphs matching the pattern.

[0065] The graph table operator can be pipelined with other execution plan operators already existing into the database, thus combining pattern matching with SQL-based querying. The result of the graph table operation can therefore be used by other execution plan operators. For example, the result of a graph table operation may be joined with the result of another subquery: SELECT*FROM G GRAPH_TABLE (MATCH (a)→(b) COLUMNS (a.city as takeoff, b.city as landing)) JOIN FAVORITE_HOTELS f ON takeoff=f.city.

[0066] The control flow of the graph query runtime is pull-based. The runtime control flow is further discussed elsewhere herein. In an embodiment, a graph table operator is made of a sequence of match operators that each consumes rows from its children in the plan. When used against the CSR, the graph table operation compiles into a physical operator that takes the CSR as input and produces a row-set, which includes a sequence of rows. The term row source may also be used for a physical operator, such as a graph table operator and/or match operator.

[0067] The graph query runtime uses multiple match operators, one for each vertex or node label in a target pattern to match. The vertex or node may also be referred as a pattern node. The sequence of match operators is isomorphic to the target pattern. FIG. **4** illustrates a sequence of match operators that follows a target pattern of (a)→(b)→(c). In this example, match operator OP**1** corresponds to vertex or node (a), match operator OP**2** to vertex or node (b), and match operator OP**3** corresponds to vertex or node (c).

TOP LEVEL DESIGN

[0068] A graph table operator computes all paths matching a given target pattern on a given graph and returns a table with number of rows equal to the number of paths matched. The graph table operator includes a sequence of match operators for evaluating a path pattern expression. Each match operator in the sequence of match operators receives as input the result of a previous match operator. Each match operator processes a single hop in a path (e.g., edge to follow+destination). Each match operator is specialized to a specific task. Assembling match operators into a path pattern expression is performed by a path pattern match compiler during query compilation. The entire path pattern expression produces paths according to the target pattern.

[0069] In an embodiment, match operators perform their logic over multiple vertex/edges in a pipelined manner and produce a number of matches that they store in a result set. The first match operator of a path pattern expression always feeds on a CSR and fills a result set that identifies a number of matching vertices. This result set serves as input to the next match operator in the path pattern expression which will produce a set of matching vertices corresponding to that stage (and edges that lead to these vertices). When the last match operator has filled its result set, the graph table operator can then fill its output row-set with entire path match. Once the row-set is filled, control is returned to parent of the graph table operator which then consumes the rows.

SPECIALIZED MATCH OPERATORS

[0070] Path pattern match processing with graph table operator is performed using multiple match operators, which are level-specialized, computing units. Each match operator is responsible for computing vertices for a specific hop of a target pattern. In an embodiment, there are three types of match operators.

[0071] One type of match operator is a Root Node Match (RNM) operator. The RNM operator is responsible for computing the vertices which match the first or root node of the pattern. The RNM operator finds the first or root level vertices that match the pattern.

[0072] Another type of match operator is an Intermediate Neighbor Match (NM) operator. The NM operator will start from a set of input vertices and produces a fixed number of neighbors for each of them. The NM operator is only used for intermediate nodes in the pattern and is not for the first or last node in the pattern. In other words, the NM operator is only used for non-leaf neighbor traversals to find intermediate or non-leaf level vertices that match the pattern.

[0073] Yet another type of match operator is a Leaf Neighbor Match (LNM) operator. The LNM operator is similar to the NM operator, but it is only used for matching the last node in the pattern. The LNM operator finds the last or leaf level vertices that match the pattern.

[0074] Each type of match operators has a different control flow. All match operators have access to a CSR on which they are run. Each type of match operator has additional specialization, depending on whether a vertex predicate, an edge predicate, or both vertex and edge predicate must be evaluated.

[0075] Each match operator is associated with a data structure where it stores the current state, such as the matched vertices by the match operator, the number of valid rows, etc. The control flow enables match operators to move data through the different level data structures and interact with the graph table operator.

[0076] Some goals of the data structures include:

- **[0077]** avoiding duplicating vertices along a path: For example, if multiple paths start with the same source, the source should be stored only once. In other words, paths are stored in a prefix encoded way that provides memory savings.
- **[0078]** providing enough information for path reconstruction: When results are moved to the output row-set, full paths can be reconstructed using the information collected in each match operator in the path pattern expression.
- **[0079]** controlling memory footprint: The amount of memory used by each match operator can be tuned.
- **[0080]** supporting pipelined execution and memory locality: The number of neighbors explored in one iteration can be limited.

[0081] To achieve the last two properties, each data structure represents the state of a match operator in a unique way. In particular, each data structure has columns of size data_size, which is a parameter of the match operator. The value of data_size corresponds with the number of rows in a data structure. Each row of the data structure holds information about one vertex matched in a different path. For root and intermediate data structures, for each vertex stored in the data structures, the runtime will explore chunk_size neighbors, where chunk_size is also a parameter of the match operator. Therefore, the data structure of the next match operator in a path pattern expression will store data_size× chunk_size rows.

[0082] FIG. **5** illustrates example data structures of two consecutive match operators. In this example, the data structure corresponding to first operator OP**1** stores information for three (3) vertices (e.g., $data_size_{OP1}=3$), and each vertex can process maximum 2 neighbors (e.g., $chunk_size_{OP1}=2$), thus the data structure of the next operator OP**2** stores information for six (6) vertices (e.g., $data_size_{OP2}=6$).

[0083] In order to compute the pattern matches for a graph, vertices are matched starting at the root node and finishing at the leaf node of the pattern. The first or root match operator—RNM operator—of a path pattern expression uses a CSR and fills its first or root level data structure storing a set of matching vertices. This first or root level data structure acts as input for the next operator (intermediate match operator-NM operator) in the path pattern expression which, using the input and the CSR, computes the vertices matching the path until that point, and stores the results in its intermediate level data structure.

[0084] In an embodiment, the pattern matching uses a batch-DFS algorithm, which means that in the process of matching the nodes in the next operator, a limited number of neighbors for each vertex in previous operator is matched.

[0085] This process continues until reaching the leaf node of the pattern. After the last or leaf match operator—LNM operator—fills its last or leaf level data structure, the graph table operator returns the matching patterns to the parent operator, which consumes the information.

[0086] As discussed above, there are three different match operator types, and each match operator type corresponds with a different data structure to store the state of a match operator. Vertices and edges matched by a match operator are stored in the result set. Each match operator uses a custom result set data structure.

[0087] FIG. **6** illustrates example data structures for storing match results of the graph table operator. Each of the different data structures is described below.

ROOT LEVEL DATA STRUCTURE

[0088] The root level data structure stores information about vertices matched by the RNM operator in a plurality of columns, which are treated as arrays during compile time and are referred to as arrays. The root level data structure includes the following arrays: Ri (Root Index) array, Si (Start Index) array, Li (Last Index) array, and Next Valid array, each of them storing on row j.

[0089] The Ri array stores the vertex id of the vertices matched by the RNM operator.

[0090] The Si array stores the index of the first unprocessed neighbor of Ri[j]. This is the absolute index in the CSR dst array, which allows for fast neighbor information retrieval, such as vertex id or edge id. Si[j] only stores indices of neighbors of Ri[j] in the dst array of the CSR.

[0091] The Li array stores the index of the last neighbor of Ri[j]. Like the Si array, it stores absolute indices in the CSR dst array. Li[j] stores the maximum value of Si[j]. When Si[j]=Li[j], it means that all the neighbors for vertex Ri[j] have been processed; as such, the current row can be skipped in the next iteration.

[0092] The Next Valid array stores the index k of the next valid row in the data structure. A row k is valid if and only if Si[k]<Li[k], meaning that there are still some neighbors to process for that row. Initially, all rows are marked as invalid.

INTERMEDIATE LEVEL DATA STRUCTURE

[0093] The intermediate level data structure stores information about vertices matched by the NM operator in a plurality of columns, which are treated as arrays during compile time and are referred to as arrays. The intermediate level data structure includes the following arrays: Ni (neighbor index) array, Si array, Li array, and Next Valid array. The last three arrays—Si, Li, and Next Valid—are similarly configured as the Si array, Li array, and Next Valid array the root level data structure. The Ri array is replaced by the Ni array.

[0094] On row j, the Ni array stores the index in the CSR dst array of the edge matched by this vertex with the corresponding vertex in previous operator. The neighbor/edge index, and not the neighbor id, is stored because edge properties may be used in a filtering or projection phase. In such as case, both vertex and edge properties with the Ni value can be retrieved.

[0095] In an embodiment, the decision whether to store the Vid (vertex id) or the Ni information, depending on what the filters or projected columns are, can be taken up at compile time. For example, if a query projects an edge, then neighbor index is stored; otherwise, neighbor id is stored.

LEAF LEVEL DATA STRUCTURE

[0096] The leaf level data structure stores information about vertices matched by the LNM operator in a plurality of columns, which are treated as arrays during compile time and are referred to as arrays. In an embodiment, the leaf level data structure contains only the Ni and Next Valid arrays. Since the leaf match operator does not have a next operator, the Si and Li arrays are undefined. Similar to the intermediate level, the Ni array can be replaced with a Neighbor Id array if no information from the edge leading to the last vertex of the path pattern is needed (e.g., no edge predicate, or no projection of any properties of that edge).

EXAMPLE MATCH OPERATORS GRAPH

[0097] FIG. **7** illustrates another example graph and corresponding CSR representation thereof. In FIG. **7**, a direct graph is shown on the left and its corresponding CSR representation is shown on the right. Assume the target pattern to match is (a)→(b)→(c). In that case, considering FIG. **4**, the graph table operator will consist of three match operators: one RNM operator, one NM operator, and one LNM operator. The paths in FIG. **7** matching the target pattern include:

[0098] (0)→(2)→(4), and

[0099] (5)→(3)→(1)

[0100] FIG. **8**A illustrates the state of match operators after one iteration. The RNM operator is for matching (a). The NM operator is for matching (b). The LNM operator is for matching (c). The Next Valid arrays of the data structures are not shown for purposes of clarity and simplicity. A result set includes the path comprising vertex **0** (e.g., src[**0**]), vertex **2** (e.g., dst[**0**]), and vertex **4** (e.g., dst[**2**]) after this first iteration of the NM operator having no more valid rows in its data structure to be processed and before accessing the RNM operator for more data. As can been seen, the path in the result set corresponds to the first path determined above.

[0101] FIGS. **8**B-**8**G illustrate the state of match operators during each iteration. In particular, FIGS. **8**B-**8**C illustrate the state of match operators in the first iteration. FIGS. **8**D-**8**E illustrate the state of match operators in the second iteration. FIGS. **8**F-**8**G illustrate the state of match operators in the third iteration. As illustrated, the result set includes a path comprising vertex **0**, vertex **2**, and vertex **4**, and another path comprising vertex **5**, vertex **3**, and vertex **1**, both of which correspond to the paths determined above.

SIZING THE DATA STRUCTURE

[0102] As discussed above, each data structure uses sizing variables to control its memory footprint and to implement the batch DFS execution. The data_size variable specifies the number of elements stored in a data structure and is configurable for every level. The chunk_size variable specifies how many neighbors (e.g., vertices matched by (b)) are explored for every parent (e.g., vertices matched by (a)) at every pull operation.

[0103] Each data structure is designed in such a way that the number of rows in a match operator is equal to the number of chunks in its child match operator. For example, in FIG. **8**A, the RNM operator has data_size =2, and chunk_size=2.

[0104] If the chunk size is greater than 1 for each hop, than the LNM operator stores a number of rows exponential in the number of hops of the path. This exponential growth can be stopped by setting chunk size to 1 for various match operators.

[0105] For chunk_size=1, then the way of traversing the neighbors is similar to DFS algorithm, and for chunk_size=∞, then the traversal algorithm is similar to BFS. Varying chunk_size allows for the tradeoff between achieving a high-degree of parallelism and being pipeline-friendly.

INDEXING

[0106] Every row, at a given level, has a corresponding chunk at the next level, where its neighbors will be stored. In order to find the parent of index i, (i/chunk_size) is computed to determine the index of the parent in the parent data structure. In order to find the jth child of index i, (i*child→chunk_size+j) is computed to determine the index of the child in the child chunk.

RUNTIME CONTROL FLOW

[0107] In an embodiment, match operators are the main computing blocks of a graph table operator. They are responsible for producing vertices corresponding to every level of a target pattern. Given a target pattern, a sequence of match operators is generated at compile time. Match operators are chained according to a pull-based control flow. When requested rows from its parent operator in a query plan, the graph table operator pulls results from its chain of match operators and construct its output row-set from these results. When rows are accumulated to the capacity of the row-set, control is returned to the graph table operator's parent.

[0108] Every match operator has access to a CSR defining the graph. The result of the LNM operator is written to the graph table operator row source that can then be pipelined with other database relational operators.

[0109] FIG. **9** is an example for a pull-based control flow for a target pattern of form (a)→(b)→(c). A graph table operator **904** is pipelined with existing database relational operators **908** and references an in-memory graph **902**. Note that the control flow and the data flow are in opposite directions. The control flow is from relational operators **908** to CSR **902**, while the data flow is from CSR **902** to relational operators **908**.

[0110] When the parent of the graph table operator **904** requests rows to process, the newly created operator will call a pull function on the LNM operator **904***c* to request rows from the graph table operator **904**. If the LNM operator **904***c* still has valid rows in its data structure, then it writes them to the graph table operator's result table **906**. If the LNM operator **904***c* does not have valid rows in its data structure, then the LNM operator **904***c* invokes the previous match operator **904***b* to fetch more neighbors from the previous operator **904***b*. This is done by calling a pull function on the previous match operator **904***b* and, then, processing the neighbors of the new vertices and storing results in its data structure. This process happens for each match operator in the path until reaching the RNM operator **904***a*, for which a pull function is simply loading the next chunk of vertices existing in a graph, by referencing the CSR **902** which represents the graph. Accordingly, each match operator in the chain will either consume more neighbors from its predecessor in the chain or call its predecessor's pull function to fetch the next batch of neighbors.

[0111] In an embodiment, a data structure at a level may not be filled with neighbors from a parent. The control flow can be modified to return only when the data structure at the level is full. For example, the match operator at the level may pull from the parent until either the level data structure is full or the parent runs out of data.

[0112] In an embodiment, iteration over valid entries in a parent data structure can be improved. For example, rather than iterating over all entries in the parent structure and jumping over invalid ones, an index of valid entries in a list is kept and referenced to iterate only over the indices present in the list. When an index becomes invalid (e.g., all neighbors for that entry were consumed), it is removed from the list.

EXAMPLE HOMOGENEOUS GRAPH SEARCH PROCESS

[0113] FIG. **10** illustrates an example flow diagram that depicts a process **1000** for supporting graph pattern matching queries inside a relational database system. The process **1000** assumes that a homogeneous property graph is defined from relational tables, and an in-memory representation of the homogeneous graph uses a CSR format.

[0114] At step **1005**, a path pattern expression is evaluated against an in-memory graph representation by at least executing a sequence of match operators. The sequence of match operators includes a root vertex match (RNM) opera-

tor, an intermediate neighbor match (NM) operator, and a leaf neighbor match (LNM) operator.

[0115] The match operators perform their logic over multiple vertex/edges in a pipelined manner and produce a number of matches that are stored as result sets in respective data structures. Results to a query are pulled from the sequence of match operators, and an output table is constructed from these results. When the parent of a graph table operator that is associated with the path pattern expression, requests rows to process, the graph table operator invokes the LNM operator, which in turn invokes the NM operator, which in turn invokes the RNM operator.

[0116] At step **1010**, in response to the NM operator invoking the RNM operator, the RNM operator stores a first set of matching first level vertices in a root level data structure.

[0117] At step **1015**, in response to the LNM operator invoking the NM operator, the NM operator generates a first set of matching intermediate neighbor vertices by accessing the root level data structure and stores the first set of matching intermediate neighbor vertices in an intermediate level data structure.

[0118] In an embodiment, the NM operator explores a particular chunk size of a particular first level vertex in the first set of matching first level vertices in said root level data structure, wherein this particular chunk size specifies a number of neighbors of the particular first-level vertex to explore.

[0119] In an embodiment, in response to the NM operator generating the first set of matching intermediate neighbor vertices by accessing the root level data structure, the RNM updates data stored in the root level data structure to indicate the next unprocessed neighbor.

[0120] At step **1020**, the LNM operator is invoked and, in response to invoking the LNM operator, the LNM operator generates a first set of matching leaf neighbor vertices by accessing the intermediate level data structure and stores the first set of matching leaf neighbor vertices in a leaf level data structure.

[0121] In an embodiment, the LNM operator explores a particular chunk size of a particular intermediate neighbor vertex in the first set of matching intermediate neighbor vertices in the intermediate level data structure, wherein this particular chunk size specifies a number of neighbors of the particular intermediate neighbor vertex to explore.

[0122] In an embodiment, in response to the LNM operator generating the first set of matching leaf neighbor vertices by accessing the intermediate level data structure, the NM updates data stored in the intermediate level intermediate level data structure.

[0123] When the leaf level data structure is filled with data, at least part of a result for the path pattern expression is generated. In an embodiment, the at least part of a result for the path pattern expression is generated based on the leaf level data structure, the intermediate level data structure, and the root level data structure. In an embodiment, the result for the path pattern expression may be pipelined with one or more database relational operators.

[0124] As shown, when an operator, in a sequence of match operators, is asked for tuples or rows by a child operator, if the operator has tuples, then the child operator will directly iterate over the neighbors of its parent operator. However, if the operator does not have tuples, then the operator will ask the same request from its parent operator.

HETEROGENEOUS GRAPHS

[0125] Most existing graph processing systems only provide support for handling homogeneous graphs, which means that the graph is stored by using only one vertex table and one single edge table, as in the example of FIGS. **2**A-**2**B. However, this model can be extended such that the vertices and edges are split across multiple tables with different semantics and properties. This model would be beneficial when trying to add a graph processing runtime over an existing relational database, which has an already-defined schema.

[0126] FIG. **11**A shows an example graph structure of authors, papers and the author-paper relationships between them. In the homogeneous graph paradigm, all the authors and the papers vertices would be stored in the same table, following that the authors would have no values for the papers' properties and, similarly, that the papers would have no values for the authors' properties. The heterogeneous paradigm models better intuition, by having two vertices tables, one for authors and another for papers, each with their own specific properties, as shown in FIG. **11**B.

[0127] Splitting the vertices and edges over multiple tables means also splitting the CSR, when using that graph representation. Therefore, if the edges are split over n tables, then n CSRs are constructed in order to define the graph, one for each edge table.

[0128] When running a pattern-matching query on a heterogeneous graph, a given pattern can correspond to multiple specializations. As an example, consider a graph that has two different vertex tables VT**1** and VT**2**. When matching the pattern (aIS VT**1**)→(b IS VT**2**)→(c), two pattern specializations can be generated:

[0129] (aIS VT**1**)→(b IS VT**2**)→(c IS VT**1**)

[0130] (aIS VT**1**)→(b IS VT**2**)→(c IS VT**2**)

[0131] The specializations are generated at compile time, based on the metadata tracking the vertex and edge tables of the graph. Specializations are part of the same specialization tree if they have an overlapping common prefix (e.g., (aIS VT**1**)→(b IS VT**2**)). In an embodiment, a graph table operator computes the paths matching the pattern for each of these specializations.

MODIFIED RUNTIME DESIGN

[0132] There are different approaches to support multiple specializations. One approach to support multiple specializations is to create a different chain of match operators for every specialization. However, this may result in computing a sequence multiple times. For the example above, the common prefix (aIS VT**1**)→(b IS VT**2**) would be computed twice. FIG. **12**A visually reflects this approach.

[0133] Another approach is to compute common prefixes in overlapping specializations, thereby decreasing pattern-matching queries run time for heterogeneous graphs. As further discussed below, this can be done by extending the match operators and control flow.

EXTENSIONS TO THE IN-MEMORY GRAPH REPRESENTATION

[0134] An in-memory representation of a heterogeneous graph includes as many CSRs as there are edge tables in a graph. A key difference to the homogeneous graph representation described above is that the indices stored in the

destination array of a CSR are indices to a destination vertex table that is different from the source array of the CSR.

[0135] An example of a heterogeneous graph in-memory representation is illustrated in FIG. **13**. In the example of FIG. **13**, The graph includes two vertex tables (e.g., V**1**, V**2**) and two edges table (e.g., E**1**, E**2**). The corresponding in-memory representation includes two CSRs: one for each edge tables. The src array for E**1**'s CSR (respectively, E**2**'s CSR) has as many entry as there are rows in V**1** (respectively, rows in V**2**). Values stored in the dst array of E**1**'s CSR (respectively, E**2**'s CSR) are indices to src array used for V**2** (respectively, V**1**). In this example, there is only one such src array for V**1**, which is the one used in the CSR for E**2**.

[0136] It may be possible for a vertex table to be a source in multiple edge tables and, therefore, to have a src array in multiple CSRs; that array would have the same dimension and the same entry would be assigned to the same row (i.e., vertex) in all src arrays. They would, however, hold a neighbor index specific to the dst array of the specific CSR.

OPERATOR EXTENSIONS

[0137] Compared to the homogeneous implementation, every match operator involving neighbor matching (e.g., RNM operator, NM operator) may use multiple CSRs, one for each specialization that branches from that match operator. From the previous example, the match operator matching (c IS VT**1**) will use a CSR corresponding to the edge table associating VT**2** with VT**1**, and the match operator matching (c IS VT**2**) will use a CSR corresponding to the edge table associating VT**2** with VT**2**. FIG. **12**B illustrates an operators graph generated for the previous example matching the pattern (a IS VT**1**)→(b IS VT**2**)→(c).

[0138] An important aspect to observe is that the simple path pattern is now transformed into a specialization tree of operators, which might have multiple leaf nodes. Each path from a tree root to a tree leaf corresponds to a path specialization, which generates different paths. Also, every specialization has the same length. In other words, all the paths from a root operator to a leaf operator have the same length, which is equal to the length of the initial path pattern. Furthermore, each leaf node should produce results into the graph table result table.

MATCH OPERATORS DATA STRUCTURE EXTENSIONS

[0139] In heterogeneous graphs processing, match operators may have multiple children match operators. For a matching operator where a common prefix ends, referred to as a branching operator (e.g., the node matching (b) in the previous example), this branching match operator stores neighbor information for each of its children match operators. The branching match operator state stores an Si array, Li array and Next Valid array for every child match operator. However, since the Ni array only depends on the previous neighbor index in the CSR, it can be shared among all the specializations stored in that operator.

[0140] FIG. **14** illustrates an example of a possible state of the graph table operators when matching the above-defined pattern of (a IS VT**1**)→(b IS VT**2**)→(c). The branching operator (b) stores neighbor information for each of its children match operators, (c IS VT**1**) and (c IS VT**2**).

[0141] In an embodiment, the additional number of columns in the data structure corresponding to the branching match operator for the children match operators are generated at compile time.

[0142] In an embodiment, the vertex id for each row can be stored in an Vid array, for optimization, so as the CSR need not be accessed for same information multiple times. The pull function of every children acts on a corresponding pair of Si, Li and Next Valid arrays.

CONTROL FLOW EXTENSION

[0143] In heterogeneous graphs processing, a path pattern is compiled into a set of independent specialization trees. A graph table operator executes each of those specialization trees to produce its output row set. The different specialization trees can be computed sequentially or in parallel, in which case, different threads are synchronized when writing results to the output row set.

[0144] As discussed above, a specialization tree contains multiple leaf operators. In order to return paths from all possible specializations, the graph table operator sequentially calls the pull function on the leaf match operators and writes the matches to the graph table result table. The change at the row set level is one change to the control flow execution when there are multiple leaf operators in a specialization tree.

[0145] Sharing the computation of prefix of a path pattern in the specialization tree requires the graph table operator to implement additional control flow to synchronize all leaf operators feeding on the same branching operators in the specialization tree. In an embodiment, the graph table operator may maintain a map <NODE, STATE> that keeps track of the state of every node in the specialization tree, wherein STATE can be ACTIVE or WAITING. The initial state for all nodes in the specialization tree may be ACTIVE.

[0146] Another change to the control flow execution is at a branch node level. Since the Ni array is shared for all the specializations starting from a branching operator, the branching operator must first process all the neighbors for every specialization before calling the pull function of its parent match operator to fetch more data. In an embodiment, branching operators have an additional pull method that their children will call instead of the original pull method. In addition, the original pull methods of NM and LNM operators need extensions.

EXAMPLE HETEROGENEOUS GRAPH SEARCH PROCESS

[0147] FIG. **15** illustrates another example flow diagram that depicts a process **1500** for supporting graph pattern matching queries inside a relational database system. The process **1500** assumes that a heterogeneous property graph is defined from relational tables, and an in-memory representation of the heterogeneous graph uses a CSR format. A path pattern expression is compiled into multiple specializations, with each specialization rooted by a starting vertex table in the graph.

[0148] At step **1505**, a path pattern expression is evaluated against an in-memory graph representation by at least executing a sequence of match operators. The sequence of match operators includes a root vertex match (RNM) operator, an intermediate neighbor match (NM) operator that is a branching NM operator, and a plurality of leaf neighbor

match (LNM) operators. The plurality of LNM operators includes at least two LNM operators of the plurality that are children match operators of the branching NM operator. The branching NM operator is the only operator with more than one child match operators.

[0149] Each path from the RNM operator to a LNM operator corresponds to a specialization, which generates different paths. Every specialization is the same length.

[0150] At step **1510**, in response to the branching NM operator invoking the RNM operator, the RNM operator stores a current set of matching first level vertices in a root level data structure.

[0151] At step **1515**, in response to the particular LNM operator of the plurality of LNM operators invoking the branching NM operator after all of the plurality of LNM operators have completed generation of current sets of matching leaf neighbor vertices, the branching NM operator generates, for each of the plurality of LNM operators, a current set of matching intermediate neighbor vertices by accessing the root level data structure, and stores, for each of the plurality of LNM operators, the current set of matching intermediate neighbor vertices in a shared intermediate level data structure.

[0152] In an embodiment, the in-memory graph representation for the heterogeneous graph comprises multiple CSRs, one per edge table in the graph. The CSRs are referenced by the branching NM operator when generating the sets of matching intermediate-neighbor vertices.

[0153] The shared intermediate level data structure stores once a common path pattern prefix of the path pattern expression.

[0154] At step **1520**, the particular LNM operator of the plurality of LNM operators is invoked and, in response to invoking the particular LNM operator, the particular LNM operator generates a current set of matching leaf neighbor vertices by accessing the shared intermediate level data structure and stores the current set of matching leaf neighbor vertices in a leaf level data structure.

[0155] In an embodiment, the branching NM operator controls the state of its children match operators (e.g., ACTIVE/WAITING) to synchronize consumption of matches from the common path pattern prefix. A pull request from all but the last active child match operator results in the children match operators to wait. A pull request from the last active child match operator triggers pulling more matches from the parent match operator (e.g., RNM operator) of the branching NM operator and reactivating all of the children match operators of the branching NM operator if additional matches are available.

[0156] Each of the LNM operators is sequentially invoked and produces results into a graph table result table. Result for the path pattern expression may be pipelined with one or more database relational operators.

[0157] In an embodiment, sharing the processing of the common path pattern prefix reduces processing and memory usage and turns the specializations into a specialization tree of match operators including the branching NM operator.

[0158] As shown, when an operator, in a sequence of match operators, is asked for tuples or rows by a child operator, if the operator has tuples, then the child operator will directly iterate over the neighbors of its parent operator. However, if the operator does not have tuples, then the operator will ask the same request from its parent operator.

EXTENSIONS TO SUPPORT COMPLEX PATH PATTERNS

[0159] The graph query runtime is extendible to support path patterns that are more complex than line-shaped path patterns discussed above.

[0160] Example complex path patterns include:

[0161] trees: An example of a query pattern including trees which are not line-shaped is (a)→(b)→(c), (a)→(d), where (c) is not a leaf node. After (c) is computed, (d) is then computed. This can be done by adding in the data structure holding (c) neighbor information for (a), such that the runtime knows it has to return to continue matching other paths.

[0162] cycles: An example of a query pattern including a cycle is (a)→(b)→(c)→(a). A naive initial version would be to convert the cycle in a cross-variable filter at query compilation time. The resulting query pattern would be: (a)→(b)→(c)→(d), where (a)=(d).

COMPLEX PATH PATTERNS INVOLVING TREES

[0163] For complex path patterns involving trees, a data structure must store results matched by every pattern variable. FIG. **16**A illustrates an example tree pattern to compute and a set of data structures corresponding to vertices in the tree pattern. The tree pattern can be decomposed as line-shaped path patterns including:

[0164] (a)→(b),

[0165] (a)→(c)→(d),

[0166] (c)→(e), and

[0167] (a)→(f).

[0168] From the tree pattern in FIG. **16**A, for example, when matching (f), it is the neighbors of (a), not the neighbors of the preceding data structure of (e), that need to be iterated over because (e) is not (f)'s neighbor. Similarly, when matching (c), it is the neighbors of (a), not the neighbors of the preceding data structure of (b), that need to be iterated over because (b) is not (c)'s neighbor. Unlike (c) and (f), the neighbors of the preceding data structure of (b) are iterated over since (a) is indeed (b)'s neighbor, in the example of FIG. **16**A.

[0169] FIG. **16**B illustrates example data structures for storing match results for the tree pattern of FIG. **16**A. As illustrated in FIG. **16**B, the Si and Li arrays in the data structure corresponding to (b) store neighbor information of (a). Similarly, the Si and Li arrays in the data structure corresponding to (e) store neighbor information of (a). Unlike (c) and (f), (b) has direct access to neighbor information of (a) since the data structure of (b) is sequentially adjacent to the data structure of (a). Accordingly, the following observations are identified:

[0170] Each of (c) and (f) requiring preceding (b) and (e), respectively, to store neighbor information of (a), is a child of (a); and

[0171] Data structures of (c) and (f) are not sequentially adjacent to the data structure of (a). (c) and (f) do not have direct access to (a).

[0172] In an embodiment, when computing next neighbor information of (a), the neighbor information is copied or replicated in the data structures corresponding to (b) and (e). Although Si and Li arrays give information about which neighbors to iterate next, they might not be neighbors of the current vertex.

EXAMPLE GRAPH SEARCH PROCESS OF A COMPLEX PATH PATTERN INVOLVING A TREE

[0173] FIG. **17** illustrates another example flow diagram that depicts a process **1700** for supporting graph pattern matching queries inside a relational database system. The process **1700** assumes that a property graph is defined from relational tables, an in-memory representation of the graph uses a CSR format, and a path pattern expression involving a tree.

[0174] At step **1705**, a path pattern expression is evaluated against an in-memory graph representation by at least executing a sequence of match operators. The sequence of match operators includes a first match operator, a second match operator, a third match operator, a last match operator, and a previous match operator preceding said last match operator.

[0175] In an embodiment, the path pattern expression comprises a plurality of nodes that includes a root node and a plurality of parent nodes. Each parent node of the plurality of parent nodes has at least one child node. Each child node is a child of one of the plurality of parent nodes. In an embodiment, each of the plurality of nodes is associated with a match operator of the sequence of match operators.

[0176] In an embodiment, neighbor information associated with a parent node of said plurality of parent nodes is stored in a data structure for a preceding match operator that is sequentially before a subsequent match operator that (1) corresponds to a node that is a child of said parent node and (2) is not sequentially adjacent to a match operator that corresponds to said parent node. As an illustration, in the example of FIGS. **16**A and **16**B, neighbor information of (a) is stored only in the data structures corresponding to (b) and (e), which are sequentially before (c) and (f). Although (b) is a child of (a), neighbor information of (a) is not stored in the preceding match operator that is sequentially before (b) because (b) has direct access to (a)'s neighbor information as it is sequentially adjacent to (a).

[0177] At step **1710**, in response to the second match operator invoking the first match operator, the first match operator stores a current set of matching first-level vertices in a first data structure of the first match operator.

[0178] At step **1715**, in response to the third match operator invoking the second match operator, the second match operator generates a current set of second matching intermediate-neighbor vertices by accessing the first data structure and stores the current set of second matching intermediate-neighbor vertices in a second data structure of the second match operator.

[0179] At step **1720**, the last match operator is invoked and, in response to invoking the last match operator, the last match operator generates a current set of matching leaf-neighbor vertices by accessing a previous data structure of the previous match operator and stores the current set of matching leaf-neighbor vertices in a last data structure of the last match operator.

[0180] When the last data structure is filled with data, at least part of a result for the path pattern expression is generated. In an embodiment, the result for the path pattern expression may be pipelined with one or more database relational operators.

[0181] As shown, when an operator, in a sequence of match operators, is asked for tuples or rows by a child operator, if the operator has tuples, then the child operator will directly iterate over the neighbors of its parent operator. However, if the operator does not have tuples, then the operator will ask the same request from its parent operator.

COMPLEX PATH PATTERNS INVOLVING CYCLES

[0182] For complex path patterns involving cycles, there are different options. A naive option, as described above, is to transform (a)→(b)→(c)→(a) to (a)→(b)→(c)→(d), where a=d, and execute the pattern via the current runtime.

[0183] Alternatively, a specialized operator, which is referred to as a common neighbor match operator, may be used to iterate over common neighbors when there are two matched nodes (for example, (a) and (c)). In this regard, (a) and (c) are first matched and (b) is then matched.

EXAMPLE GRAPH SEARCH PROCESS OF A COMPLEX PATH PATTERN INVOLVING A CYCLE

[0184] FIG. **18** illustrates yet another example flow diagram that depicts a process **1800** for supporting graph pattern matching queries inside a relational database system. The process **1800** assumes that a property graph is defined from relational tables, an in-memory representation of the graph uses a CSR format, and a path pattern expression involving a cycle.

[0185] At step **1805**, a path pattern expression is evaluated against an in-memory graph representation by at least executing a sequence of match operators. The sequence of match operators includes a root-vertex match (RNM) operator, a plurality of intermediate-neighbor match (NM) operators, and a leaf-neighbor match (LNM) operator.

[0186] In an embodiment, the path pattern expression comprises a plurality of nodes that includes a particular node and a plurality of parent nodes. Each parent node of the plurality of parent nodes has one child node. Each child node is a child of one of the plurality of parent nodes.

[0187] In an embodiment, a number of match operators in said sequence of match operators is one more than a number of nodes in said plurality of nodes. In an embodiment, each of the plurality of nodes other than the particular node is associated with a match operator of a NM operator of the plurality NM operators. The particular node is associated with both of the RNM operator and the LNM operator.

[0188] At step **1810**, in response to a first NM operator of the plurality of NM operators invoking the RNM operator, the RNM operator stores a current set of matching first-level vertices in a root-level data structure.

[0189] At step **1815**, in response to the LNM operator invoking a last NM operator of the plurality of NM operators, the last NM operator generates a current set of last matching intermediate-neighbor vertices by accessing a previous intermediate-level data structure of a previous NM operator preceding the last NM operator, and stores the current set of last matching intermediate-neighbor vertices in a last intermediate-level data structure.

[0190] At step **1820**, the LNM operator is invoked and, in response to invoking the LNM operator, the LNM operator generates a current set of matching leaf-neighbor vertices by accessing the last intermediate-level data structure and stores the current set of matching leaf-neighbor vertices in a leaf-level data structure.

[0191] When the leaf-level data structure is filled with data, at least part of a result for the path pattern expression is generated. In an embodiment, the result for the path pattern expression may be pipelined with one or more database relational operators.

[0192] As shown, when an operator, in a sequence of match operators, is asked for tuples or rows by a child operator, if the operator has tuples, then the child operator will directly iterate over the neighbors of its parent operator. However, if the operator does not have tuples, then the operator will ask the same request from its parent operator.

PARALLEL EXECUTION

[0193] In an embodiment, the query engine may exploit parallelism to efficiently process a path pattern. One consideration is to split work across execution threads. The main unit of work to parallelize is the neighbor iteration. Level sync parallelism entails parallelizing neighbor iteration for every operator. Every thread will pick a number of rows from an operator (e.g., at level i) such that the total number of neighbors to iterate on is equal to a predefined job_size constant, and it will write its results at the corresponding chunk of its child operator (e.g., at level i+1).

[0194] Another consideration is to achieve work balance (fair distribution of work) when high-degree nodes are present in the data set. In order to achieve work balance, vertices whose number of neighbors is greater than the job_size parameter is split. A high degree vertex in an operator is split it by writing its Ni multiple times in the corresponding chunk in the child operator, such that for every row it holds Li−Si<job_size. For vertices whose number of neighbors is greater than chunk_size*job_size, its remaining neighbor information is written in a chunk state, from which the operator can consume on backtrack. This ensures that every thread iterates over a similar number of neighbors, achieving thus work balance.

[0195] FIG. **19** illustrates a parallel execution example. By exploiting parallelism, work balance is achieved since each thread iterates over the same number of neighbors. No synchronization is needed because different threads write to different result chunks. In an embodiment, the job_size constant is configurable such that the compute time limit per job inside the DB is DB runtime compliant.

ADAPTIVE OPERATORS

[0196] In an embodiment, operators (control flow and data structures) may be tuned in order to adapt them to different input data. The choice of a tuned operator can be done at compile time based on various data statistics, such as average degree of neighbors, diameter, connectivity metrics, etc., on the input graph.

[0197] When RNM and NM operators consume neighbors, it is checked, for every newly generated neighbor, if it has next neighbors or not. In the case that it has neighbors, then the vertex and its neighbor information are written to the underlying data structure; otherwise, the next neighbor is checked. This control flow is very effective when the path pattern has a very high degree of selectivity as it avoids materializing useless data (neighbors that do not have further neighbors). For example, if (a)(b)(c) is matched, then writing to the data structure storing vertices matching (b) all the vertices that do not have neighbors (so they cannot have a (c)) can be avoided.

[0198] On the other hand, this control flow can be suboptimal, as every time a value from CSR dst array (the neighbor id) is read, CSR src array must be accessed to read neighbor information. This can break locality and reduce the potential of using SIMD instructions. In this regard, the control flow can be modified to first store the neighbor id (regardless if it has more neighbors or not) and only then access neighbor information. In the case of non-selective path patterns, this modification can bring big benefits.

[0199] The same modification can be applied in the case of heterogeneous graphs. In addition to that, in the case of branching nodes (that store neighbor information for multiple children), Ni array may be replicated for every <Si, Li> pair, allowing for independent iteration over neighbors in different child specializations. This can be very useful when the two specializations have different degrees of selectivity.

DIFFERENCES AND ADVANTAGES

[0200] Compared to traditional database systems, the runtime leverages a main-memory graph representation in a relational execution engine to achieve graph pattern matching performance, which is significantly better than using state of the art join operators. In the case of heterogeneous graphs, the runtime allows sharing computation of common prefixes across multiple instantiations of a path pattern. This approach brings savings both in terms of compute time and memory footprint as the overlapping prefixes need to be computed and stored only once.

[0201] Compared to traditional graph processing systems, the runtime efficiently combines BFS and DFS exploration in order to efficiently address challenges found inside a single machine relational database system when processing graph queries (e.g., pipeline friendly, small memory footprint and locality on neighbor iteration). The result is a batch-DFS exploration where, at every level, only a batch of neighbors is explored. The batch size is configurable at every level.

[0202] The batch-DFS exploration addresses the above-mentioned graph processing challenges. For example, the batch-DFS exploration is pipeline friendly. It is suitable for usage in relational DBMSs. The batch allows for exploration of only a small number of neighbors at a time, thus allowing to stream tuples to the next match/relational operator much faster than a classical BFS approach. The size of a batch can be adapted just-in-time to available resource (number of available core/available memory).

[0203] For another example, the batch-DFS exploration has a small memory footprint. The total memory footprint of the runtime can be controlled by configuring the batch size at every level. This does not limit the size of queries that can be processed. Generated paths can be stored in a prefix encoded way by using smart data structures.

[0204] For yet another example, the batch-DFS exploration performs locality on neighbor iteration. It can achieve the performance of the more classic BFS exploration by iterating over a batch of neighbors at a time, increasing thus probability for cache hits.

DATABASE OVERVIEW

[0205] Embodiments of the present invention are used in the context of database management systems (DBMSs). Therefore, a description of an example DBMS is provided.

[0206] Generally, a server, such as a database server, is a combination of integrated software components and an allocation of computational resources, such as memory, a node, and processes on the node for executing the integrated software components, where the combination of the software and computational resources are dedicated to providing a particular type of function on behalf of clients of the server. A database server governs and facilitates access to a particular database, processing requests by clients to access the database.

[0207] A database comprises data and metadata that is stored on a persistent memory mechanism, such as a set of hard disks. Such data and metadata may be stored in a database logically, for example, according to relational and/or object-relational database constructs.

[0208] Users interact with a database server of a DBMS by submitting to the database server commands that cause the database server to perform operations on data stored in a database. A user may be one or more applications running on a client computer that interact with a database server. Multiple users may also be referred to herein collectively as a user.

[0209] A database command may be in the form of a database statement. For the database server to process the database statements, the database statements must conform to a database language supported by the database server. One non-limiting example of a database language that is supported by many database servers is SQL, including proprietary forms of SQL supported by such database servers as Oracle, (e.g. Oracle Database 11g). SQL data definition language ("DDL") instructions are issued to a database server to create or configure database objects, such as tables, views, or complex types. Data manipulation language ("DML") instructions are issued to a DBMS to manage data stored within a database structure. For instance, SELECT, INSERT, UPDATE, and DELETE are common examples of DML instructions found in some SQL implementations. SQL/XML is a common extension of SQL used when manipulating XML data in an object-relational database.

[0210] Generally, data is stored in a database in one or more data containers, each container contains records, and the data within each record is organized into one or more fields. In relational database systems, the data containers are typically referred to as tables, the records are referred to as rows, and the fields are referred to as columns. In object-oriented databases, the data containers are typically referred to as object classes, the records are referred to as objects, and the fields are referred to as attributes. Other database architectures may use other terminology. Systems that implement the present invention are not limited to any particular type of data container or database architecture. However, for the purpose of explanation, the examples and the terminology used herein shall be that typically associated with relational or object-relational databases. Thus, the terms "table", "row" and "column" shall be used herein to refer respectively to the data container, record, and field.

[0211] Query Optimization and Execution Plans

[0212] Query optimization generates one or more different candidate execution plans for a query, which are evaluated by the query optimizer to determine which execution plan should be used to compute the query.

[0213] Execution plans may be represented by a graph of interlinked nodes, each representing an plan operator or row sources. The hierarchy of the graphs (i.e., directed tree) represents the order in which the execution plan operators are performed and how data flows between each of the execution plan operators.

[0214] An operator, as the term is used herein, comprises one or more routines or functions that are configured for performing operations on input rows or tuples to generate an output set of rows or tuples. The operations may use interim data structures. Output set of rows or tuples may be used as input rows or tuples for a parent operator.

[0215] An operator may be executed by one or more computer processes or threads. Referring to an operator as performing an operation means that a process or thread executing functions or routines of an operator are performing the operation.

[0216] A row source performs operations on input rows and generates output rows, which may serve as input to another row source. The output rows may be new rows, and or a version of the input rows that have been transformed by the row source.

[0217] A match operator of a path pattern expression performs operations on a set of input matching vertices and generates a set of output matching vertices, which may serve as input to another match operator in the path pattern expression. The match operator performs logic over multiple vertex/edges to generate the set of output matching vertices for a specific hop of a target pattern corresponding to the path pattern expression.

[0218] An execution plan operator generates a set of rows (which may be referred to as a table) as output and execution plan operations include, for example, a table scan, an index scan, sort-merge join, nested-loop join, filter, and importantly, a full outer join.

[0219] A query optimizer may optimize a query by transforming the query. In general, transforming a query involves rewriting a query into another semantically equivalent query that should produce the same result and that can potentially be executed more efficiently, i.e. one for which a potentially more efficient and less costly execution plan can be generated. Examples of query transformation include view merging, subquery unnesting, predicate move-around and push-down, common subexpression elimination, outer-to-inner join conversion, materialized view rewrite, and star transformation.

HARDWARE OVERVIEW

[0220] According to one embodiment, the techniques described herein are implemented by one or more special-purpose computing devices. The special-purpose computing devices may be hard-wired to perform the techniques, or may include digital electronic devices such as one or more application-specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs) that are persistently programmed to perform the techniques, or may include one or more general purpose hardware processors programmed to perform the techniques pursuant to program instructions in firmware, memory, other storage, or a combination. Such special-purpose computing devices may also combine custom hard-wired logic, ASICs, or FPGAs with custom programming to accomplish the techniques. The special-purpose computing devices may be desktop computer systems, portable computer systems, handheld devices, networking devices or any other device that incorporates hard-wired and/or program logic to implement the techniques.

[0221] For example, FIG. **20** is a block diagram that illustrates a computer system **2000** upon which an embodiment of the invention may be implemented. Computer system **2000** includes a bus **2002** or other communication mechanism for communicating information, and a hardware processor **2004** coupled with bus **2002** for processing information. Hardware processor **2004** may be, for example, a general purpose microprocessor.

[0222] Computer system **2000** also includes a main memory **2006**, such as a random access memory (RAM) or other dynamic storage device, coupled to bus **2002** for storing information and instructions to be executed by processor **2004**. Main memory **2006** also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor **2004**. Such instructions, when stored in non-transitory storage media accessible to processor **2004**, render computer system **2000** into a special-purpose machine that is customized to perform the operations specified in the instructions.

[0223] Computer system **2000** further includes a read only memory (ROM) **2008** or other static storage device coupled to bus **2002** for storing static information and instructions for processor **2004**. A storage device **2010**, such as a magnetic disk, optical disk, or solid-state drive is provided and coupled to bus **2002** for storing information and instructions.

[0224] Computer system **2000** may be coupled via bus **2002** to a display **2012**, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device **2014**, including alphanumeric and other keys, is coupled to bus **2002** for communicating information and command selections to processor **2004**. Another type of user input device is cursor control **2016**, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor **2004** and for controlling cursor movement on display **2012**. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

[0225] Computer system **2000** may implement the techniques described herein using customized hard-wired logic, one or more ASICs or FPGAs, firmware and/or program logic which in combination with the computer system causes or programs computer system **2000** to be a special-purpose machine. According to one embodiment, the techniques herein are performed by computer system **2000** in response to processor **2004** executing one or more sequences of one or more instructions contained in main memory **2006**. Such instructions may be read into main memory **2006** from another storage medium, such as storage device **2010**. Execution of the sequences of instructions contained in main memory **2006** causes processor **2004** to perform the process steps described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions.

[0226] The term "storage media" as used herein refers to any non-transitory media that store data and/or instructions that cause a machine to operate in a specific fashion. Such storage media may comprise non-volatile media and/or volatile media. Non-volatile media includes, for example, optical disks, magnetic disks, or solid-state drives, such as storage device **2010**. Volatile media includes dynamic memory, such as main memory **2006**. Common forms of storage media include, for example, a floppy disk, a flexible disk, hard disk, solid-state drive, magnetic tape, or any other magnetic data storage medium, a CD-ROM, any other optical data storage medium, any physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, NVRAM, any other memory chip or cartridge.

[0227] Storage media is distinct from but may be used in conjunction with transmission media. Transmission media participates in transferring information between storage media. For example, transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus **2002**. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications.

[0228] Various forms of media may be involved in carrying one or more sequences of one or more instructions to processor **2004** for execution. For example, the instructions may initially be carried on a magnetic disk or solid-state drive of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system **2000** can receive the data on the telephone line and use an infra-red transmitter to convert the data to an infra-red signal. An infra-red detector can receive the data carried in the infra-red signal and appropriate circuitry can place the data on bus **2002**. Bus **2002** carries the data to main memory **2006**, from which processor **2004** retrieves and executes the instructions. The instructions received by main memory **2006** may optionally be stored on storage device **2010** either before or after execution by processor **2004**.

[0229] Computer system **2000** also includes a communication interface **2018** coupled to bus **2002**. Communication interface **2018** provides a two-way data communication coupling to a network link **2020** that is connected to a local network **2022**. For example, communication interface **2018** may be an integrated services digital network (ISDN) card, cable modem, satellite modem, or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface **2018** may be a local area network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface **2018** sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

[0230] Network link **2020** typically provides data communication through one or more networks to other data devices. For example, network link **2020** may provide a connection through local network **2022** to a host computer **2024** or to data equipment operated by an Internet Service Provider (ISP) **2026**. ISP **2026** in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" **2028**. Local network **2022** and Internet **2028** both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link **2020** and through communication interface **2018**, which carry the digital data to and from computer system **2000**, are example forms of transmission media.

[0231] Computer system **2000** can send messages and receive data, including program code, through the network (s), network link **2020** and communication interface **2018**. In

the Internet example, a server **2030** might transmit a requested code for an application program through Internet **2028**, ISP **2026**, local network **2022** and communication interface **2018**.

[0232] The received code may be executed by processor **2004** as it is received, and/or stored in storage device **2010**, or other non-volatile storage for later execution.

[0233] A computer system process comprises an allotment of hardware processor time, and an allotment of memory (physical and/or virtual), the allotment of memory being for storing instructions executed by the hardware processor, for storing data generated by the hardware processor executing the instructions, and/or for storing the hardware processor state (e.g. content of registers) between allotments of the hardware processor time when the computer system process is not running. Computer system processes run under the control of an operating system, and may run under the control of other programs being executed on the computer system.

[0234] In the foregoing specification, embodiments of the invention have been described with reference to numerous specific details that may vary from implementation to implementation. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense. The sole and exclusive indicator of the scope of the invention, and what is intended by the applicants to be the scope of the invention, is the literal and equivalent scope of the set of claims that issue from this application, in the specific form in which such claims issue, including any subsequent correction.

SOFTWARE OVERVIEW

[0235] FIG. **21** is a block diagram of a basic software system **2100** that may be employed for controlling the operation of computing device **2000**. Software system **2100** and its components, including their connections, relationships, and functions, is meant to be exemplary only, and not meant to limit implementations of the example embodiment(s). Other software systems suitable for implementing the example embodiment(s) may have different components, including components with different connections, relationships, and functions.

[0236] Software system **2100** is provided for directing the operation of computing device **2000**. Software system **2100**, which may be stored in system memory (RAM) **2006** and on fixed storage (e.g., hard disk or flash memory) **2010**, includes a kernel or operating system (OS) **2110**.

[0237] The OS **2110** manages low-level aspects of computer operation, including managing execution of processes, memory allocation, file input and output (I/O), and device I/O. One or more application programs, represented as **2102**A, **2102**B, **2102**C . . . **2102**N, may be "loaded" (e.g., transferred from fixed storage **2010** into memory **2006**) for execution by the system **2100**. The applications or other software intended for use on device **2100** may also be stored as a set of downloadable computer-executable instructions, for example, for downloading and installation from an Internet location (e.g., a Web server, an app store, or other online service).

[0238] Software system **2100** includes a graphical user interface (GUI) **2115**, for receiving user commands and data in a graphical (e.g., "point-and-click" or "touch gesture") fashion. These inputs, in turn, may be acted upon by the system **2100** in accordance with instructions from operating system **2110** and/or application(s) **2102**. The GUI **2115** also serves to display the results of operation from the OS **2110** and application(s) **2102**, whereupon the user may supply additional inputs or terminate the session (e.g., log off).

[0239] OS **2110** can execute directly on the bare hardware **2120** (e.g., processor(s) **2004**) of device **2000**. Alternatively, a hypervisor or virtual machine monitor (VMM) **2130** may be interposed between the bare hardware **2120** and the OS **2110**. In this configuration, VMM **2130** acts as a software "cushion" or virtualization layer between the OS **2110** and the bare hardware **2120** of the device **2000**.

[0240] VMM **2130** instantiates and runs one or more virtual machine instances ("guest machines"). Each guest machine comprises a "guest" operating system, such as OS **2110**, and one or more applications, such as application(s) **2102**, designed to execute on the guest operating system. The VMI **2130** presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems.

[0241] In some instances, the VMM **2130** may allow a guest operating system to run as if it is running on the bare hardware **2120** of device **2000** directly. In these instances, the same version of the guest operating system configured to execute on the bare hardware **2120** directly may also execute on VMM **2130** without modification or reconfiguration. In other words, VMM **2130** may provide full hardware and CPU virtualization to a guest operating system in some instances.

[0242] In other instances, a guest operating system may be specially designed or configured to execute on VMM **2130** for efficiency. In these instances, the guest operating system is "aware" that it executes on a virtual machine monitor. In other words, VMM **2130** may provide para-virtualization to a guest operating system in some instances.

[0243] The above-described basic computer hardware and software is presented for purpose of illustrating the basic underlying computer components that may be employed for implementing the example embodiment(s). The example embodiment(s), however, are not necessarily limited to any particular computing environment or computing device configuration. Instead, the example embodiment(s) may be implemented in any type of system architecture or processing environment that one skilled in the art, in light of this disclosure, would understand as capable of supporting the features and functions of the example embodiment(s) presented herein.

EXTENSIONS AND ALTERNATIVES

[0244] Although some of the figures described in the foregoing specification include flow diagrams with steps that are shown in an order, the steps may be performed in any order, and are not limited to the order shown in those flowcharts. Additionally, some steps may be optional, may be performed multiple times, and/or may be performed by different components. All steps, operations and functions of a flow diagram that are described herein are intended to indicate operations that are performed using programming in a special-purpose computer or general-purpose computer, in various embodiments. In other words, each flow diagram in this disclosure, in combination with the related text herein, is a guide, plan or specification of all or part of an algorithm for programming a computer to execute the functions that are described. The level of skill in the field associated with this disclosure is known to be high, and therefore the flow

diagrams and related text in this disclosure have been prepared to convey information at a level of sufficiency and detail that is normally expected in the field when skilled persons communicate among themselves with respect to programs, algorithms and their implementation.

[0245] In the foregoing specification, the example embodiment(s) of the present invention have been described with reference to numerous specific details. However, the details may vary from implementation to implementation according to the requirements of the particular implement at hand. The example embodiment(s) are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method comprising:

evaluating a path pattern expression against an in-memory graph representation, by at least executing a sequence of match operators that includes a root-vertex match (RNM) operator, an intermediate-neighbor match (NM) operator that is a branching NM operator, and a plurality of leaf-neighbor match (LNM) operators,

wherein said branching NM operator is associated with at least two LNM operators of said plurality of LNM operators;

wherein executing said sequence of match operators includes:

in response to said branching NM operator invoking said RNM operator, said RNM operator storing a current set of matching first-level vertices in a root-level data structure;

in response to a particular LNM operator of said plurality of LNM operators invoking said branching NM operator after all of said plurality of LNM operators completing generation of current sets of matching leaf-neighbor vertices:

said branching NM operator generating, for each of said plurality of LNM operators, a current set of matching intermediate-neighbor vertices by accessing said root-level data structure;

said branching NM operator storing, for each of said plurality of LNM operators, said current set of matching intermediate-neighbor vertices in a shared intermediate-level data structure;

wherein said shared intermediate-level data structure stores once a common path pattern prefix of said path pattern expression;

invoking said particular LNM operator of said plurality of LNM operators, wherein in response to invoking said particular LNM operator:

said particular LNM operator generating a current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure;

said particular LNM operator storing said current set of matching leaf-neighbor vertices in a leaf-level data structure.

2. The method of claim **1**, further comprising generating at least part of a result for said path pattern expression based on said leaf-level data structure, said shared intermediate-level data structure, and said root-level data structure.

3. The method of claim **2**, wherein said at least part of said result for said path pattern expression is generated when said leaf-level data structure is filled with data.

4. The method of claim **3**, further comprising pipelining said result for said path pattern expression with one or more database relational operators.

5. The method of claim **1**, further comprising in response to said branching NM operator generating said current set of matching intermediate-neighbor vertices by accessing said root-level data structure, said RNM operator updating data stored in said root-level data structure.

6. The method of claim **1**, further comprising in response to said particular LNM operator generating said current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure, said branching NM operator updating data stored in said shared intermediate-level data structure.

7. The method of claim **1**, wherein said RNM operator is invoked by said branching NM operator in response to said branching NM operator being invoked.

8. The method of claim **1**, wherein said branching NM operator is invoked by said particular LNM operator in response to said particular LNM operator being invoked.

9. The method of claim **1**, wherein said branching NM operator generating a current set of matching intermediate-neighbor vertices by accessing said root-level data structure comprises exploring a particular chunk size of a particular first-level vertex in said current set of matching first-level vertices in said root-level data structure, wherein said particular chunk size specifies a number of neighbors of said particular first-level vertex to explore.

10. The method of claim **1**, wherein said particular LNM operator generating said current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure comprises exploring a particular chunk size of a particular intermediate-neighbor vertex in said current set of matching intermediate-neighbor vertices in said shared intermediate-level data structure, wherein said particular chunk size specifies a number of neighbors of said particular intermediate-neighbor vertex to explore.

11. The method of claim **1**, wherein said in-memory graph representation comprises graph topologies referenced by said branching NM operator when generating said current set of matching intermediate-neighbor vertices.

12. One or more non-transitory computer-readable storage media storing one or more sequences of program instructions which, when executed by one or more computing devices, cause:

evaluating a path pattern expression against an in-memory graph representation, by at least executing a sequence of match operators that includes a root-vertex match (RNM) operator, an intermediate-neighbor match (NM) operator that is a branching NM operator, and a plurality of leaf-neighbor match (LNM) operators,

wherein said branching NM operator is associated with at least two LNM operators of said plurality of LNM operators;

wherein executing said sequence of match operators includes:

in response to said branching NM operator invoking said RNM operator, said RNM operator storing a current set of matching first-level vertices in a root-level data structure;

in response to a particular LNM operator of said plurality of LNM operators invoking said branching NM operator after all of said plurality of LNM operators completing generation of current sets of matching leaf-neighbor vertices:

said branching NM operator generating, for each of said plurality of LNM operators, a current set of

matching intermediate-neighbor vertices by accessing said root-level data structure;

said branching NM operator storing, for each of said plurality of LNM operators, said current set of matching intermediate-neighbor vertices in a shared intermediate-level data structure;

wherein said shared intermediate-level data structure stores once a common path pattern prefix of said path pattern expression;

invoking said particular LNM operator of said plurality of LNM operators, wherein in response to invoking said particular LNM operator:

said particular LNM operator generating a current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure;

said particular LNM operator storing said current set of matching leaf-neighbor vertices in a leaf-level data structure.

13. The one or more non-transitory computer-readable storage media of claim **12**, wherein the one or more sequences of the program instructions which, when executed by the one or more computing devices, further cause generating at least part of a result for said path pattern expression based on said leaf-level data structure, said shared intermediate-level data structure, and said root-level data structure.

14. The one or more non-transitory computer-readable storage media of claim **13**, wherein said at least part of said result for said path pattern expression is generated when said leaf-level data structure is filled with data.

15. The one or more non-transitory computer-readable storage media of claim **14**, wherein the one or more sequences of the program instructions which, when executed by the one or more computing devices, further cause pipelining said result for said path pattern expression with one or more database relational operators.

16. The one or more non-transitory computer-readable storage media of claim **12**, wherein the one or more sequences of the program instructions which, when executed by the one or more computing devices, further cause in response to said branching NM operator generating said current set of matching intermediate-neighbor vertices by accessing said root-level data structure, said RNM operator updating data stored in said root-level data structure.

17. The one or more non-transitory computer-readable storage media of claim **12**, wherein the one or more sequences of the program instructions which, when executed by the one or more computing devices, further cause in response to said particular LNM operator generating said current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure, said branching NM operator updating data stored in said shared intermediate-level data structure.

18. The one or more non-transitory computer-readable storage media of claim **12**, wherein said RNM operator is invoked by said branching NM operator in response to said branching NM operator being invoked.

19. The one or more non-transitory computer-readable storage media of claim **12**, wherein said branching NM operator is invoked by said particular LNM operator in response to said particular LNM operator being invoked.

20. The one or more non-transitory computer-readable storage media of claim **12**, wherein said branching NM operator generating a current set of matching intermediate-neighbor vertices by accessing said root-level data structure comprises exploring a particular chunk size of a particular first-level vertex in said current set of matching first-level vertices in said root-level data structure, wherein said particular chunk size specifies a number of neighbors of said particular first-level vertex to explore.

21. The one or more non-transitory computer-readable storage media of claim **12**, wherein said particular LNM operator generating said current set of matching leaf-neighbor vertices by accessing said shared intermediate-level data structure comprises exploring a particular chunk size of a particular intermediate-neighbor vertex in said current set of matching intermediate-neighbor vertices in said shared intermediate-level data structure, wherein said particular chunk size specifies a number of neighbors of said particular intermediate-neighbor vertex to explore.

22. The one or more non-transitory computer-readable storage media of claim **12**, wherein said in-memory graph representation comprises graph topologies referenced by said branching NM operator when generating said current set of matching intermediate-neighbor vertices.

* * * * *