Querying graph databases

Graph traversals

NoSQL: Lecture 3

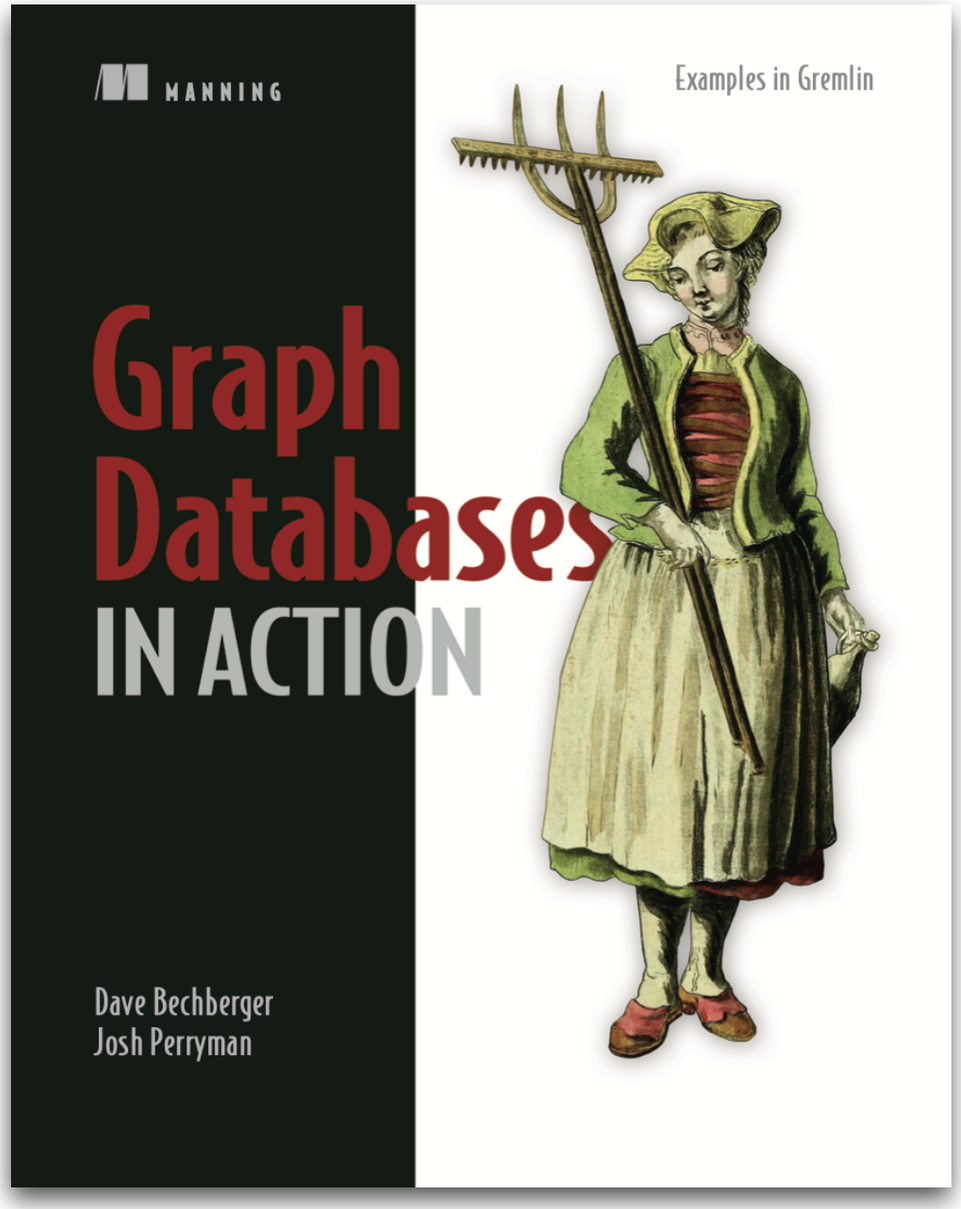
Piotr Fulmański

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
University of Lodz

Graph databases In Action

by Dave Bechberger and Josh Perryman

Manning Publications, 2020



What we need

Installation of Gremlin Console

- The easiest way to get started with Gremlin is to install the Gremlin Console. The Gremlin Console is a REPL that allows immediate feedback on the results of Gremlin traversals.
- As a prerequisite, Java 8 is required for the Gremlin Console to run.
- Download the Gremlin Console, unpack it wherever you want - no install is needed.
- Running the Gremlin Console is a simple as executing starting script from command line.

In my case

```
$TINKERPOP_DIR = ~/Desktop/tinkerpop
```

```
nosql@nosql:~$ $TINKERPOP_DIR/gremlin-console/bin$
./ gremlin.sh
[ cut some illegal reflective access warning lines
here ]
          \,,,/
          (o o)
 -----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
```

```
gremlin> Gremlin.version()
==>3.4.8

gremlin> :help

Available commands:
[ cut some lines here ]
  :exit (:x ) Exit the shell
  :quit (:q ) Alias to: :exit
[ cut some lines here ]
```

What we need

Sample data

We will use sample data of secret agents network. As they are a secret agents, we don't know their names. Instead we will use a short pseudonyms.

You can load sample data running console with script argument:

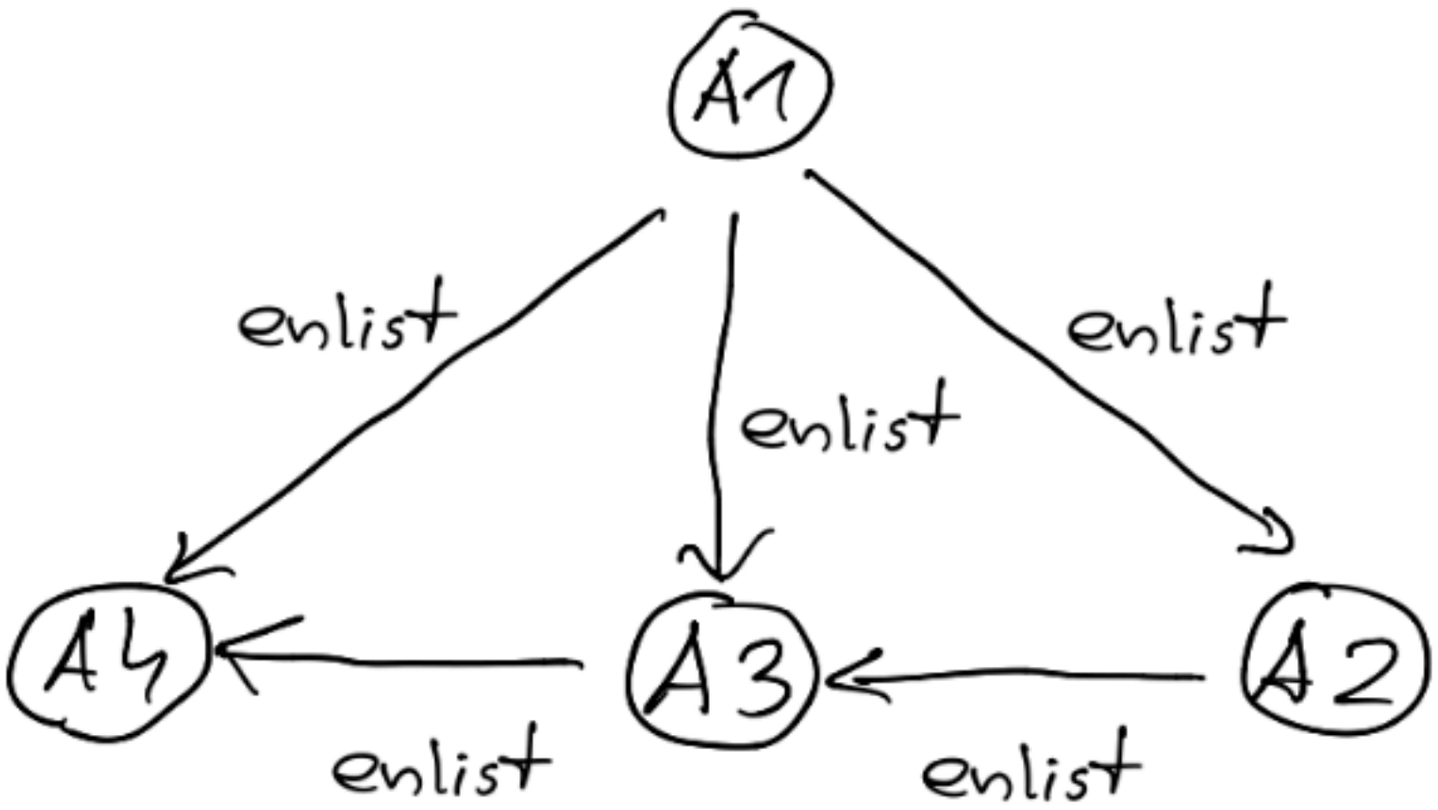
```
-i [PATH_TO_SAMPLE_DATA_FILE]agents.groovy
```

Example:

```
nosql@nosql:~$ ./$TINKERPOP_DIR/gremlin-console/bin/
gremlin.sh -i ~/Desktop/nosql_2/agents.groovy
[ cut some lines here ]
gremlin> graph
==>tinkergraph[vertices:4 edges:5]
```

Sample data

Secret agents net



Travers[e|al|al source|er]

- *Traverse* - The process of moving from vertex to edge or edge to vertex as we navigate through a graph.
- *Traversal* - A specification of one or more steps or actions to perform on a graph.
- *Traversal source* - The traversal source is a concept specific to TinkerPop. It represents the base or starting point from which steps traverse the graph. By convention, this is usually represented with the variable `g` and is required to begin any traversal.
- *Traverser* - The computing process; maintains all the metadata about the current branch of the graph it's moving through.

Graph traverse

- Querying in a graph database by traverse entails defining the series of steps for moving through the graph from one element to another.
- As we traverse through the graph **every step starts at one location and almost always ends at a different location**. Thus it is very important to exactly know where we are as different locations modeled different entities with different set of properties and relations and in consequence offer different traversal possibilities.

Graph traverse

Keep track of where we are

- **This idea is completely opposite to what we may infer from relational experience:**
  - In a relational database we may use any two tables joining them whenever we want.
  - In a graph, we're limited to using only the edges or vertices which are next to currently considered node. To be able to proceed further, we have to keep track of where we are within the structure of our graph data model.

Graph traverse

Relations directedness

- In relational word we can combine what we want with what we want. In graph databases edges directions determines the way we traverse the graph by control if we move on only incoming, outgoing, or both edge directions.

Graph traverse

Amnesia

- The last one thing we have to be aware of is a lack of backward memory. However it sounds, it means that at the and of any traversal we have an access only to ending node and data directly related to it. In contrast, in typical SQL query we may retrieve data from all columns belonging to tables joined together.

Graph traverse

Amnesia

Thus it is often useful to think traversal in terms of a stream processing:

- enters from the previous step,
- an operation is performed on current,
- and data is transmitted on to the next step.

Practical part

Graph traverse

Very basic "traversal"

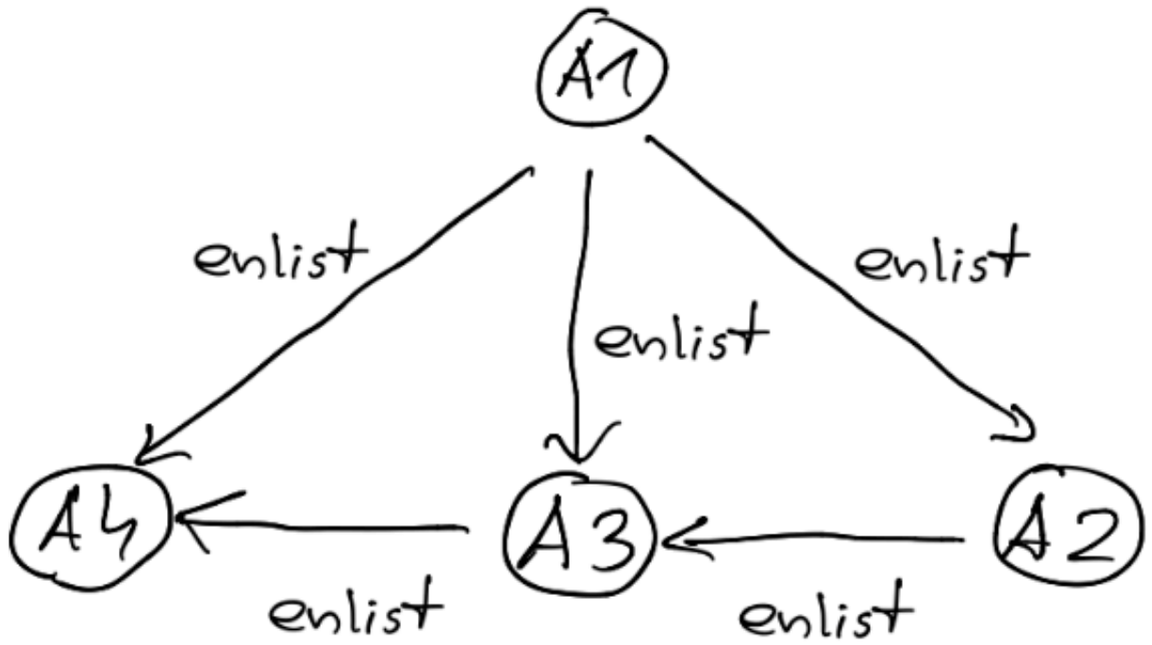
```
gremlin> graph
==>tinkergraph[vertices:4 edges:5]

gremlin> g.V()
==>v[0]
==>v[2]
==>v[4]
==>v[6]


gremlin> g.E()
==>e[8][0-enlist->2]
==>e[9][0-enlist->4]
==>e[10][0-enlist->6]
==>e[11][2-enlist->4]
==>e[12][4-enlist->6]
```

Graph traverse

Very basic "traversal"




```
gremlin> g.V().elementMap()
==>[id:0,label:agent,pseudonim:A1]
==>[id:2,label:agent,pseudonim:A2]
==>[id:4,label:agent,pseudonim:A3]
==>[id:6,label:agent,pseudonim:A4]

gremlin> g.E().elementMap()
==>[id:8,label:enlist,IN:[id:2,label:agent],OUT:[id:0,label:agent]]
==>[id:9,label:enlist,IN:[id:4,label:agent],OUT:[id:0,label:agent]]
==>[id:10,label:enlist,IN:[id:6,label:agent],OUT:[id:0,label:agent]]
==>[id:11,label:enlist,IN:[id:4,label:agent],OUT:[id:2,label:agent]]
==>[id:12,label:enlist,IN:[id:6,label:agent],OUT:[id:4,label:agent]]
```

Graph traverse

Very basic "traversal"

```
gremlin> g.V().elementMap().unfold()
==>id=0
==>label=agent
==>pseudonim=A1
==>id=2
==>label=agent
==>pseudonim=A2
==>id=4
==>label=agent
==>pseudonim=A3
==>id=6
==>label=agent
==>pseudonim=A4
```

Graph traverse

Very basic "traversal"

```
gremlin> g.E().elementMap().unfold()
==>id=8
==>label=enlist
==>IN={id=2, label=agent}
==>OUT={id=0, label=agent}
==>id=9
==>label=enlist
==>IN={id=4, label=agent}
==>OUT={id=0, label=agent}
==>id=10
==>label=enlist
==>IN={id=6, label=agent}
==>OUT={id=0, label=agent}
==>id=11
==>label=enlist
==>IN={id=4, label=agent}
==>OUT={id=2, label=agent}
==>id=12
==>label=enlist
==>IN={id=6, label=agent}
==>OUT={id=4, label=agent}
```

Graph traverse

What is *g*?

- The `g` step is always the first step in every Gremlin traversal.
- The `g` represents the *traversal source* for our graph and is the base on which all traversals are written.
- This could be called anything, but the convention with a TinkerPop graph in transactional mode is to use `g`.
- `g` is not a graph; it is a traversal source by convention, defined by `g = graph.traversal()`

Graph traverse

First real traversal

Graph traverse

First real traversal

Task: Find all agents known by agent A2 (A2 -[enlist]->??).

We start by outlining the steps we need to take through the graph:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Walk the outgoing *enlist* edges to the incident vertex.
4. Return the *pseudonim*.

Graph traverse

First real traversal

Task: Find all agents known by agent A2 (A2 -[enlist]->??).

We start by outlining the steps we need to take through the graph:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Walk the outgoing *enlist* edges to the incident vertex.
4. Return the *pseudonim*.

Next, we map these plain English steps to the corresponding steps in Gremlin.

Graph traverse

First real traversal

Task: Find all agents known by agent A2 (A2 -[enlist]->??).

1. Given all the vertices in a graph.
2.
3.
4.

```
g.V()
```

Graph traverse

First real traversal

Task: Find all agents known by agent A2 (A2 -[enlist]->??).

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3.
4.

```
g.V().
has('agent', 'pseudonim', 'A2')
```

Graph traverse

First real traversal

Task: Find all agents known by agent A2 (A2 -[enlist]->??).

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Walk the outgoing *enlist* edges to the incident vertex.
4.

```
g.V().
has('agent', 'pseudonim', 'A2').
out('enlist')
```

Graph traverse

First real traversal

Task: Find all agents known by agent A2 (A2 -[enlist]->??).

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Walk the outgoing *enlist* edges to the incident vertex.
4. Return the *pseudonim*.

```
g.V().
has('agent', 'pseudonim', 'A2').
out('enlist').
values('pseudonim')
```

Graph traverse

First real traversal

```
g.V().has('agent', 'pseudonim', 'A2').
out('enlist').values('pseudonim')
```

Traversal source
Global step
Filtering steps
Traversal steps
Values steps

```
==>A3
```

Graph traverse

Filtering steps

- `hasLabel(label)` – Yields all vertices or edges of the specified label type.

- `has(key, value)` – Yields all vertices and edges with a property matching the specified `key` and `value`.

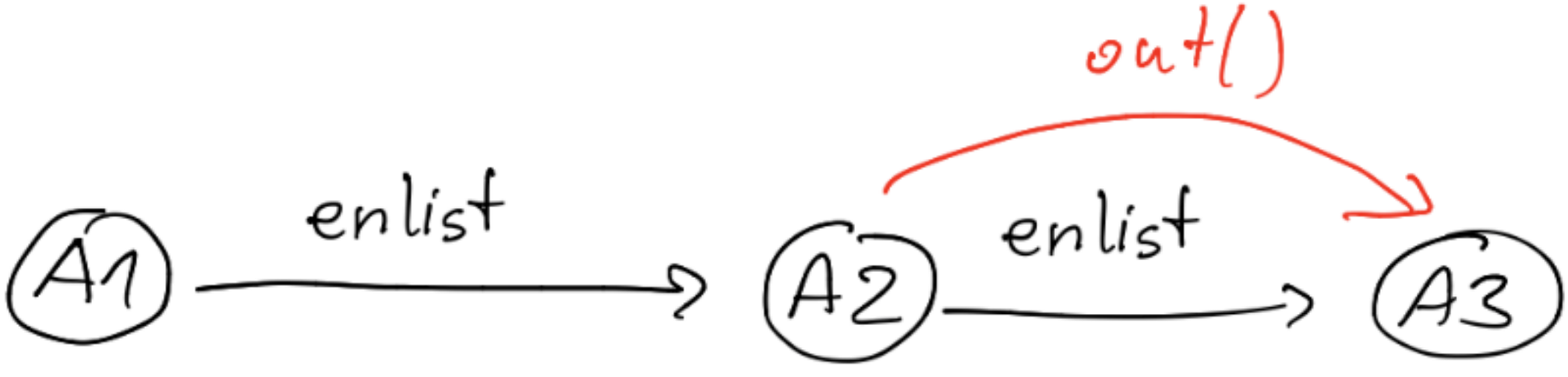
- `has(label, key, value)` – Yields all vertices and edges with both the specified `label` and with a property matching the specified `key` and `value`. This performs the same function as this combination:

  ```
  g.V().hasLabel('agent').has('pseudonim', 'A2')
  ```

Graph traverse

Traversal steps: *in*, *out*

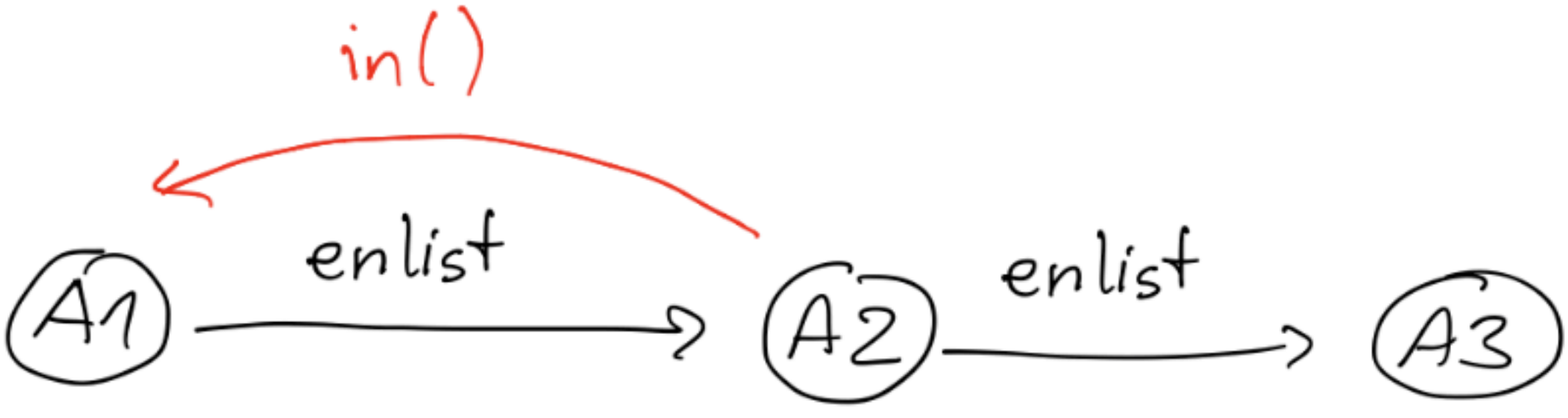
```
g.V().
has('agent', 'pseudonim', 'A2').
out('enlist').
values('pseudonim')
==>A3
```



Graph traverse

Traversal steps: *in*, *out*

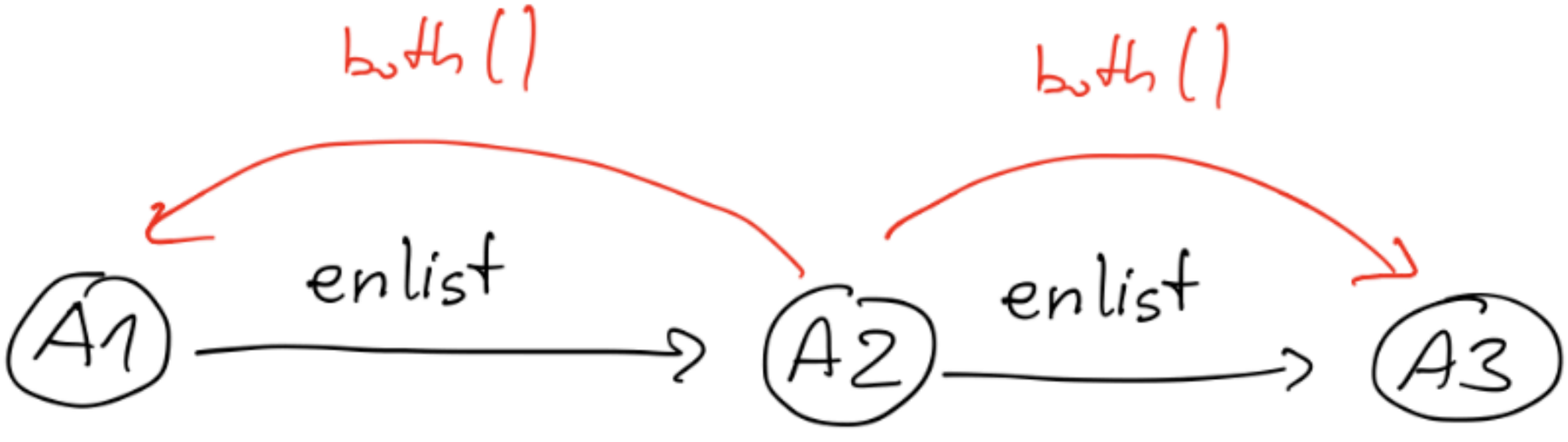
```
g.V().
has('agent', 'pseudonim', 'A2').
in('enlist').
values('pseudonim')
==>A1
```



Graph traverse

Traversal steps: *in*, *out*

```
g.V().
has('agent', 'pseudonim', 'A2').
both('enlist').
values('pseudonim')
==>A3
==>A1
```



Graph traverse

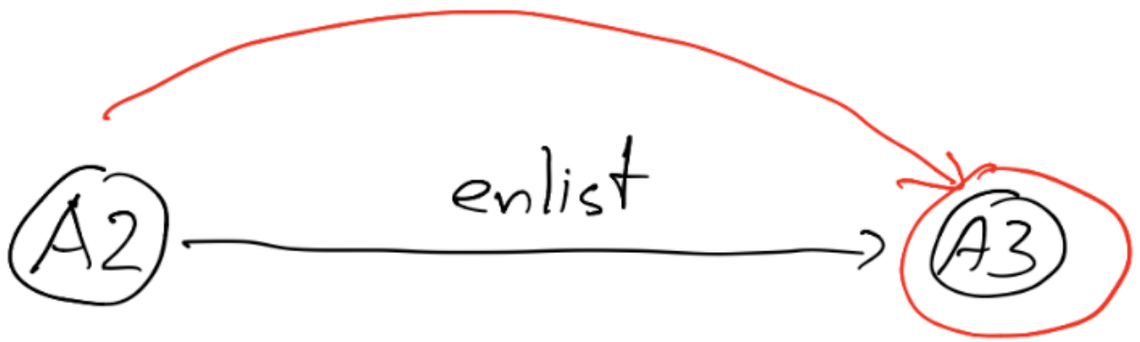
Traversal steps

- `out(string…)`: Move to the outgoing adjacent vertices given the edge labels.
- `in(string…)`: Move to the incoming adjacent vertices given the edge labels.
- `both(string…)`: Move to both the incoming and outgoing adjacent vertices given the edge labels.
- `outE(string…)`: Move to the outgoing incident edges given the edge labels. Traverses from the current vertex onto the outgoing incident edges.
- `inE(string…)`: Move to the incoming incident edges given the edge labels. Traverses from the current vertex onto the incoming incident edges.
- `bothE(string…)`: Move to both the incoming and outgoing incident edges given the edge labels. Traverses from the current vertex onto the incident edges, regardless of direction.
- `outV()`: Move to the outgoing vertex. Traverses from the current edge to the outgoing vertex.
- `inV()`: Move to the incoming vertex. Traverses from the current edge to the incoming vertex.
- `bothV()`: Move to both vertices. Traverses from the current edge to both of the incident vertices.
- `otherV()`: Move to the vertex that was not the vertex that was moved from.

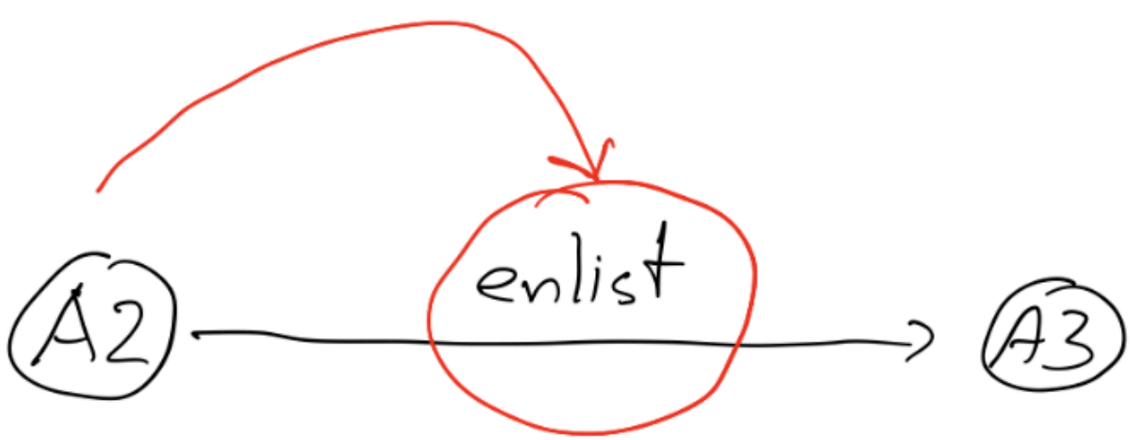
Graph traverse

Traversal steps: `out()` vs `outE()`

- `g.V().has('agent','pseudonim','A2').out()`



- `g.V().has('agent','pseudonim','A2').outE()`



Graph traverse

Traversal steps: `out()` vs `outE()`

When possible, don't traverse an edge to the other vertex.

Compare:

1. `g.V().out().count()`

2. `g.V().outE().count()`

In the **first case**, we go through each edge and count the vertices on the other side.

In the **second case**, we count the edge that we see from our vertex.

Graph traverse

Traversal steps

```
g.V().
has('agent', 'pseudonim', 'A2').
out('enlist').
values('pseudonim')
==>A3

g.V().
has('agent', 'pseudonim', 'A2').
outE('enlist').
inV().
values('pseudonim')
==>A3
```

Recursive graph traverse

The essence of graph querying

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Traverse the outgoing *enlist* edges to the incident vertex.

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Traverse the outgoing *enlist* edges to the incident vertex.
4. Traverse to the incoming vertex (at this point, we're at A?).

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Traverse the outgoing *enlist* edges to the incident vertex.
4. Traverse to the incoming vertex (at this point, we're at A?).
5. Traverse the outgoing *enlist* edges to the incident vertex.

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Traverse the outgoing *enlist* edges to the incident vertex.
4. Traverse to the incoming vertex (at this point, we're at A?).
5. Traverse the outgoing *enlist* edges to the incident vertex.
6. Traverse to the incoming vertex (at this point, we're at ??).

Recursive graph traverse

The essence of graph querying

Task: Find all agents known by agents who A2 knows (A2 -[enlist]-> A? -[enlist]-> ??).

This *friends-of-friends-type* question is a common pattern in all kind of "social" networks. If we want to accomplish this in our social network graph, we would need to execute the following steps

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of *A2*.
3. Traverse the outgoing *enlist* edges to the incident vertex.
4. Traverse to the incoming vertex (at this point, we're at A?).
5. Traverse the outgoing *enlist* edges to the incident vertex.
6. Traverse to the incoming vertex (at this point, we're at ??).
7. Return the *pseudonim* property value.

Recursive graph traverse

The essence of graph querying

Recursive graph traverse

The essence of graph querying

Using explicit query

Recursive graph traverse

The essence of graph querying

Using explicit query

```
g.V().has('agent', 'pseudonim', 'A2').
out('enlist').
out('enlist').
values('pseudonim')

==>A4
```

Recursive graph traverse

The essence of graph querying

Using explicit query

```
g.V().has('agent', 'pseudonim', 'A2').
out('enlist').
out('enlist').
values('pseudonim')

==>A4
```

This works and provides the correct answer, but it only works because we knew that we needed to repet `out()` step two times. In many cases, we don’t know how many repetitions we’ll need.

Recursive graph traverse

The essence of graph querying

Recursive graph traverse

The essence of graph querying

- `repeat(traversal)` - Repeatedly loops thorough the steps until instructed to stop. The `traversal` parameter represents the set of Gremlin steps to be repeated within the loop.

Recursive graph traverse

The essence of graph querying

- `repeat(traversal)` - Repeatedly loops thorough the steps until instructed to stop. The `traversal` parameter represents the set of Gremlin steps to be repeated within the loop.

- `times(integer)` - A modifier for a `repeat()` loop. The `integer` parameter represents the number of operations for the loop to execute.

Recursive graph traverse

The essence of graph querying

- `repeat(traversal)` - Repeatedly loops thorough the steps until instructed to stop. The `traversal` parameter represents the set of Gremlin steps to be repeated within the loop.

- `times(integer)` - A modifier for a `repeat()` loop. The `integer` parameter represents the number of operations for the loop to execute.

- `until(traversal)` - A modifier for a `repeat()` loop. The `traversal` parameter represents the set of Gremlin steps that evaluate for each loop. When the traversal evaluates to `true`, the `repeat()` step exits.

Recursive graph traverse

The essence of graph querying

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a) Walk the outgoing *enlist* edges to the incident vertex.
    b) Execute the repeated step(s) two times.

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a) Walk the outgoing *enlist* edges to the incident vertex.
    b) Execute the repeated step(s) two times.
4. Return the *pseudonim* property value.

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2.
3.
    a)
    b)
4.

```
g.V().
```

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3.
    a)
    b)
4.

```
g.V().
has('agent', 'pseudonim', 'A2').
```

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a)
    b)
4.

```
g.V().
has('agent', 'pseudonim', 'A2').
repeat(

)
```

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a)
    b) Execute the repeated step(s) two times.
4.

```
g.V().
has('agent', 'pseudonim', 'A2').
repeat(

).
times(2).
```

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a) Walk the outgoing *knows* edges to the incident vertex.
    b) Execute the repeated step(s) two times.
4.

```
g.V().
has('agent', 'pseudonim', 'A2').
repeat(
  out('enlist')
).
times(2).
```

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a) Walk the outgoing *knows* edges to the incident vertex.
    b) Execute the repeated step(s) two times.
4. Return the *pseudonim* property value.

```
g.V().
has('agent', 'pseudonim', 'A2').
repeat(
  out('enlist')
).
times(2).
values('pseudonim')
```

Recursive graph traverse

The essence of graph querying

Now all the steps expressed in plain English are as follow:

1. Given all the vertices in a graph.
2. Find all the *agent* vertices with a *pseudonim* of A2.
3. Repeat the following step(s):
    a) Walk the outgoing *knows* edges to the incident vertex.
    b) Execute the repeated step(s) two times.
4. Return the *pseudonim* property value.

```
g.V().
has('agent', 'pseudonim', 'A2').
repeat(
  out('enlist')
).
times(2).
values('pseudonim')
==>A4
```

Recursive graph traverse

Remarks

If we have no knowledge how many edges we should expect: one, two or more we can use `until()` step:

```
g.V().
has('agent', 'pseudonim', 'A2').
until(outE('enlist').count().is(0)).
repeat(
  out('enlist')
).
values('pseudonim')

==>A4
```

Recursive graph traverse

Remarks

If we have no knowledge how many edges we should expect: one, two or more we can use `until()` step:

```
g.V().
has('agent', 'pseudonim', 'A2').
until(outE('enlist').count().is(0)).
repeat(
  out('enlist')
).
values('pseudonim')

==>A4
```

BUT BE CAREFUL!

Recursive graph traverse

Remarks

If we have a knowledge about initial vertex and final vertex, our traversal is less probable to become an *unbounded traversal*.

```
g.V().
has('agent', 'pseudonim', 'A2').
until(has('agent', 'pseudonim', 'A4')).
repeat(
  out('enlist')
).
values('pseudonim')

==>A4
```

Recursive graph traverse

Remarks

To determine the intermediate steps, we need to introduce a modifier step to the `repeat()` step, known as `emit()`.

```
g.V().
has('agent', 'pseudonim', 'A2').
until(
  has('pseudonim', 'A4')
).
repeat(
  out('enlist')
).
emit().
values('pseudonim')

==>A3
==>A4
==>A4
```

Recursive graph traverse

Remarks

To determine the intermediate steps, we need to introduce a modifier step to the `repeat()` step, known as `emit()`.

```
g.V().
has('agent', 'pseudonim', 'A2').
until(
  has('pseudonim', 'A4')
).
emit().
repeat(
  out('enlist')
).
values('pseudonim')

==>A2
==>A3
==>A4
```

Bibliography

- [Bec] Dave Bechberger, Josh Perryman, *Graph Databases in Action*, Manning Publications, 2020