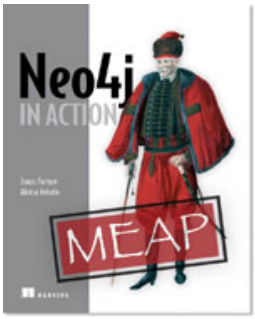
MANNING PUBLICATIONS



Neo4j in Action
By Jonas Partner and Aleksa Vukotic

Imagine you have a traversal that returns all movies that John's friends have seen but John hasn't. Neo4j Core API, allows you to use the low-level Neo4j data structures to achieve such a goal. In this article based on chapter 3 of Neo4j in Action, you'll see how you can refactor a Core API example using Neo4j Transversal Framework and implement a custom evaluator.

You may also be interested in…

Traversing Using Neo4j Traversal Framework

Neo4j Traversal Framework is a callback-based framework with fluent builder API, which allows you to expressively build the traversal rules in a single line of code. The main part of the Traversal Framework is the org.neo4j.graphdb.traversal.TraversalDescription interface, which defines builder methods for describing traverser behavior. You can think of traverser as a robot that jumps from node to node via relationships, with a well-defined set of rules about order of traversal, a relationship to follow, nodes and relationships to include in the result, and so on. `TraversalDescription` is immutable object that is used to define traversal rules. Adding any new traversal rule (by invoking one of the builder methods on `TraversalDescription` interface) always returns a new instance of `TraversalDescription` instance.

In order to illustrate how Travesal API works, let's refactor a Neo4j Core API traversal example to use Neo4j Traversal Framework. Let's take a look at listing 1, where we implement the traversal that returns movies that have been seen by John Johnson friends, without checking for duplicates among movies seen by John himself.

Listing 1 Using Traversal Framework to find movies seen by friends

```
Node userJohn = graphDb.getNodeById(JOHN_JOHNSON_NODE_ID);
RelationshipType isFriendOfRelationshipType =
DynamicRelationshipType.withName("IS_FRIEND_OF");
RelationshipType hasSeenRelationshupType = DynamicRelationshipType.withName("HAS_SEEN");

TraversalDescription traversalMoviesFriendsLike =
          Traversal.description()                                          #1
             .relationships(isFriendOfRelationshipType)                    #2
             .relationships(hasSeenRelationshupType, Direction.OUTGOING)   #3
                      .uniqueness(Uniqueness.NODE_GLOBAL)                  #4
                      .evaluator(Evaluators.atDepth(2));                   #5
Traverser traverser = traversalMoviesFriendsLike.traverse(userJohn);       #6
Iterable<Node> moviesFriendsLike = traverser.nodes();                      #7

for (Node movie : moviesFriendsLikeList) {
     logger.info("Found movie: " + movie.getProperty("name"));
}
```

#1 Instantiating TraversalDescription
#2 Adds IS_FRIEND_OF relationships to the list of relationships to follow
#3 Adds HAS_SEEN relationships to the list of relationships to follow
#4 Sets uniqueness rule so that each node in the result is unique
#5 Creates traverser from the created TraversalDescription starting from the node representing user John Johnson
#6 Gets the all nodes visited as a result

Neo4j provides default implementation of the `TraversalDescription` interface, which you can instantiate using the static factory method `Traversal.description()` (#1). This will typically be a starting point when building a `TraversalDescription` in most cases, as you would rarely need to provide your own `TraversalDescription`

implementation. Next, we define the relationships we want to include in our traversal (#2, #3). `TraversalDescription` maintains the list of relationships added using `TraversalDescrition.relationships(..)` method, and only relationships that this list contains will be followed by the traverser. You can add relationship type without specifying direction, in which case both directions will be allowed (#2), or you can specify allowed direction, relative to the starting node (#3). In the next line, we specify how the traverser should behave in perspective of uniqueness of the nodes and relationships it encounters during traversal. We want each node to be visited exactly once, so we set uniqueness to `Uniqueness.NODE_GLOBAL` (#4). Other allowed values are, for example, `Uniqueness.NODE_PATH`, which allows multiple traversal through the same node while the path from start node to the current node is unique, or `Uniqueness.RELATIONSHIP_GLOBAL`, which allows traversal through each relationship only once.

Finally, we are going to add an evaluator to our `TraversalDescription` (#5). The evaluator is responsible for two features of Neo4j Traversal Framework. It determines whether the current node should be returned as part of traversal result. It also determines if the traversal should continue further down the current path of the graph, or if it should be abandoned and another path tried instead. The evaluators in Neo4j are defined using `org.neo4j.graphdb.traversal.Evaluator` interface. Neo4j provides a number of convenient implementations that you can use out of the box. The provided implementations are accessible via static factory methods in the `org.neo4j.graphdb.traversal.Evaluators` class. We are using the `Evaluators.atDepth(int depth)` evaluator, which simply accepts all nodes at the specified depth, counting from the start node. In addition, this evaluator stops any traversal at depth higher than specified.

Evaluators are one of the key concepts of the Traversal Framework, and it is likely that you will need to implement your own custom evaluators often. We will have a look at custom implementation of `Evaluator` interface next.

Implementing a custom evaluator

We need to improve the code from listing 1 to exclude from the result movies that John has seen already. In order to do that, we need to add new rule to the traversal description. Neo4j's `Evaluator` implementation defines which nodes to keep in the result and which to discard. In addition, it defines when the traverser should stop the traversal altogether. Based on that, we can implement additional custom evaluator that will make sure to exclude the movies already seen by the user. The `Evaluator` interface defines a single method that we need to implement, `public Evaluation evaluate(Path path)`. This method accepts a single argument of type `org.neo4j.graphdb.Path`, which represents all nodes and relationships that were traversed until the current node. This interface defines a number of convenient methods to collect information about current state of traversal, such as all nodes traversed, all relationships traversed, the path start node, the path end node, and so on. Table 1 lists all methods that are available from the Path interface.

Table 1 Methods defined on org.neo4j.graphdb.Path interface

METHOD SIGNATURE	DESCRIPTION
`Node startNode();`	Start node of the path, not to be confused with start node of the relationship.
`Node endNode();`	Path end node, which is the current node of the traversal. Not to be confused with relationship end node.
`Relationship lastRelationship();`	Last relationship traversed
`Iterable<Relationship> relationships();`	All relationship traversed until current node, in traversal order
`Iterable<Node> nodes();`	All nodes in the oath, in the traversal order
`int length();`	Returns the length of the path, which is actually the same number of relationships traversed (or number of

nodes minus one)

NOTE It is important not to be confused by the path start and end nodes with the start and end nodes of relationships. Relationship start and end nodes depend on the direction of the relationship. Path start and end nodes depend on the traversal order. Since with Neo4j you can traverse relationships in any direction, this can lead to the confusion to anyone new to Neo4j.

The `evaluate(..)` method has the return type `org.neo4j.graphdb.traversal.Evaluation`, which is Java enumeration with four possible values. Based on the returned `Evaluation`, the traverser decides whether to stop (or prune in Neo4j terminology) or continue with the traversal. In addition, the returned evaluation is used to determine whether to keep the current node in the result (include) or to discard it (exclude). A combination of these four variables defines four values of the `Evaluation` enumeration. Table 2 explains the different `Evaluation` values.

Table 2 Possible values of the Evaluation enumeration

METHOD SIGNATURE	DESCRIPTION
`INCLUDE_AND_CONTINUE`	Includes the current node in the result and continues traversing further
`INCLUDE_AND_PRUNE`	Includes the current node in the result but stops going further down this path
`EXCLUDE_AND_CONTINUE`	Discards the current node and continues traversing.
`EXCLUDE_AND_PRUNE`	Discards the current node and stops traversing further

Now that we understand the `Path` interface and the Evaluation enumeration, it's time to implement our custom `Evaluator`. Listing 2 shows the code implementation.

Listing 2 Custom Evaluator to exclude movie nodes already seen by the user

```
public class CustomFilteringEvaluator implements Evaluator {     #A
    private RelationshipType hasSeenRelationshupType =
[CA]DynamicRelationshipType.withName("HAS_SEEN");
    private final Node userNode;

    public CustomFilteringEvaluator(Node userNode) {                    #B
        this.userNode = userNode;
    }

    public Evaluation evaluate(Path path) {
        Node currentNode = path.endNode();                              #C
        if (!currentNode.hasProperty("type") ||
[CA]currentNode.getProperty("type").equals("Movie")) {
            return Evaluation.EXCLUDE_AND_PRUNE;                        #D
        }
        for (Relationship r : [CA]currentNode.getRelationships(Direction.INCOMING,
hasSeenRelationshupType)) {                                             #E
            if (r.getStartNode().equals(userNode)) {
                return Evaluation.EXCLUDE_AND_CONTINUE;                 #F
            }
        }
        return Evaluation.INCLUDE_AND_CONTINUE;                         #G
    }
}
```

#A Our class implements org.neo4j.graphdb.traversal.Evaluator interface
#B The constructor that takes one argument, the user we perform evaluation for
#C Gets reference to the current node in the traversal
#D If the current node isn't a movie, discards it and stops further traversal as we're only interested in movies
#E Iterates through all incoming HAS_SEEN relationships of the current movie node
#F If the start node of the relationship is same as user node, discards the current node (as user has already seen it) and continues
#G Otherwise, includes the current node in the result and continues

We will now include the implemented custom evaluator in the traversal definition. Listing 3 shows improved traversal definition.

Listing 3 Improved traversal definition with custom evaluator

```
Node userJohn = graphDb.getNodeById(JOHN_JOHNSON_NODE_ID);
RelationshipType isFriendOfRelationshipType =
DynamicRelationshipType.withName("IS_FRIEND_OF");
RelationshipType hasSeenRelationshupType = DynamicRelationshipType.withName("HAS_SEEN");

TraversalDescription traversalMoviesFriendsLike =
           Traversal.description()
               .relationships(isFriendOfRelationshipType)
               .relationships(hasSeenRelationshupType, Direction.OUTGOING)
.uniqueness(Uniqueness.NODE_GLOBAL)
              .evaluator(Evaluators.atDepth(2));                          #1
              .evaluator(new CustomFilteringEvaluator(userJohn));         #2
Traverser traverser = traversalMoviesFriendsLike.traverse(userJohn);
Iterable<Node> moviesFriendsLike = traverser.nodes();

for (Node movie : moviesFriendsLikeList) {
     logger.info("Found movie: " + movie.getProperty("name"));
}
```

#1 Existing Evaluator is still used
#2 Custom evaluator is added to the `TraversalDescription`

You will remember that we have one evaluator already, which includes only nodes at the depth level two (#1). Good thing is that we don't need to remove it, but simply add the new one (#2). In Neo4j Traversal Framework, multiple evaluators can be composed together, so you can add many evaluators to the single `TraversalDescription`. In case multiple evaluators are included during the traversal, the Boolean algebra applies; in order for current node to be included in the result, all evaluators must return `Evaluation` with `INCLUDE` element (`INCLUDE_AND_CONTINUE` or `INCLUDE_AND_PRUNE`). Similarly, in order to continue traversal down the same path, all evaluators must return `CONTINUE` evaluator (`INCLUDE_AND_CONTINUE` or `EXCLUDE_AND_CONTINUE`).

If you run the improved application on the same data set, you will see the expected output as before:

```
Found movie: Alien
Found movie: Heat
```

However, this time we used fluent Neo4j Traversal Framework, and managed to implement solution in a more expressive manner. Traversal Framework has declarative nature—we simply declare how we want our traverser to behave, and start it to do its job. We recommend you use Traverser Framework for graph traversals whenever possible because it allows you to produce readable, maintainable and performant code when dealing with complex graph traversals. However, in addition to Traversal Framework, Neo4j has another, older Traversal API that is more imperative in nature.

Summary

For complex traversals, the Core API implementation could quickly become complex. That's where Neo4j Traversal API comes in place. Using fluent Traversal API, you can describe the traversal query in a simple and declarative manner, without sacrificing any of the power of graph traversal and with minimal performance impact.

Both Traversal API and Core API have their strengths and should be used to solve graph problems accordingly.

Here are some other Manning titles you might be interested in:

Big Data
Nathan Marz

Hadoop in Practice
Alex Holmes

HBase in Action
Nick Dimiduk and Amandeep Khurana

Last updated: July 9, 2012