An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: Renesca

Felix Dietze, Johannes Karoff, André Calero Valdez, Martina Ziefle, Christoph Greven, Ulrik Schroeder

An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: renesca*

Felix Dietze[1], Johannes Karoff[2], André Calero Valdez[1], Martina Ziefle[1], Christoph Greven[3], and Ulrik Schroeder[3]

[1] Human-Computer Interaction Center, Campus Boulevard 57, RWTH Aachen University, Germany
{dietze,calero-valdez,ziefle}@comm.rwth-aachen.de
[2] RWTH Aachen University, Germany
{johannes.karoff}@rwth-aachen.de
[3] Learning Technologies Research Group, Ahornstr. 55, RWTH Aachen University, Germany
{greven,schroeder}@cs.rwth-aachen.de

Abstract. The usage and application of graph databases is increasing. Many research problems are based on understanding relationships between data entities. This is where graph databases are powerful. Nevertheless, software developers model and think in object-oriented software. Combining both approaches leads to a paradigm mismatch. This mismatch can be addressed by using object graph mappers (OGM). OGM adapt graph databases for object-oriented code, to relieve the developer. Most graph database access frameworks only support table-based result outputs. This defeats one of the strongest purposes of using graph databases. In order to harness both the power of graph databases and object-oriented modeling (e.g. type-safety, inheritance, etc.) we propose an open-source framework with two libraries: 1) *renesca*, which is a graph database driver providing graph-query-results and change-tracking. 2) *renesca-magic*, a macro-based ER-modeling domain specific language (DSL). Both were tested in a graph-based application and lead to dramatic improvements in code size (factor 10) and extensibility of the code, with no significant effect on performance.

Keywords: Graph Databases, Scala, Neo4j, REST API, Object-Graph-Mapper, OGM

1 Introduction

An increasing amount of today's applications uses graph-based data structures. Social Network Analysis [1], bibliometrics [2], biomedical data [3], recommender systems, and neural networks are just some of the research fields that use graph-based data structures (e.g. railroad-planning [4], remote sensing [5]). Even software development itself can benefit from the power of graph databases [6]. Graph

* The framework is available at https://github.com/renesca/

databases differ from other forms of databases as they rely on graph-based data storage internally. This comes with the benefit of naturally modeling data that derives meaning from structure and relations. Another benefit of graph databases is their performance in local search tasks that are based on relationships of the stored data. This is particularly helpful when using social network data [7].

On the other hand a lot of software development today is done using the object-oriented software paradigm. Object-oriented modeling and programming allows an intuitive organization of code as it allows the developer to think in *object* terms that are natural to the human mind.

Holzschuher and Peinl [8] investigated the benefits of graph databases in comparison to Apache Shinding that uses a relational database backend. They state that using graph-based databases comes with little performance overhead and more readable code.

1.1 Object-Graph Mapping

Naturally, vendors have come up with Object-Graph Mapping tools that are already in use. Neo4j comes with an OGM[4] that is currently in release 2. But also third party vendors provide OGM for Neo4j. Hibernate is a very popular example. Hibernate OGM[5] also supports Neo4j and several other databases like NoSQL but also relational Databases (ORM). A similar solution is provided by Spring Data Neo4j[6] that uses AspectJ for advanced mapping features. The NoSQL database OrientDB also comes with a type-safe property-graph model that is ensured by the database itself[7]. The framework Structr[8] provides a Graph-Database and an Object Schema, but is aimed mostly at enterprise data management and comes with graphical editing tools. To the best of our knowledge there is no OGM that supports graph-query results, hyper-graphs and multiple-inheritance at our time of development.

1.2 Neo4j and Scala

We picked Neo4j as it is the graph database that outperforms many of its competitors [9,10]. Neo4j can be used as an embedded database or over the network via a REST API. In the embedded case the data is stored in the file system and is accessed by using Neo4j as a library. This allows to work imperatively with the complete graph to traverse and modify nodes as well as relations or to declaratively query data with the Cypher query language. The REST API provides access to nodes and relations via REST calls or Cypher queries.

Traditional relational databases have table data structures and the queries always result in a table form. Query results coming from the Neo4j REST API

[4] `https://neo4j.com/docs/ogm-manual/current/`
[5] `http://hibernate.org/ogm/`
[6] `http://projects.spring.io/spring-data-neo4j/`
[7] `http://orientdb.com/docs/last/Schema.html`
[8] `https://structr.org`

can be graphs and tables. At the time of development, the existing Scala Neo4j REST libraries imitated ORMs (Object Relational Mapper) and were therefore limited to list or table structures of nodes and relations. This is neither convenient to work with nor does it fit the purpose of a graph database, especially when using hypergraph data structures.

2 Our Contribution

We present the framework renesca for handling graph database query results using a new paradigm. With renesca, the result data structure can be a graph instead of a table. Changes to the data can be cumulatively applied (c.f. *Unit of Work* [11]).

On top of this paradigm we present the framework and DSL renesca-magic for describing graph database schemata. The DSL is implemented with Scala macros which generate code for using renesca in a type-safe way. This allows to interpret graph results with regards to a schema (see Fig. 1). Furthermore, the framework allows to realize hyperrelations, which means connecting relations with nodes.

The whole renesca framework (i.e. renesca and renesca-magic) is implemented in Scala as a lightweight (approx. 3,200 LOC + 6,500 LOC tests) OGM which can be used as two separate libraries.

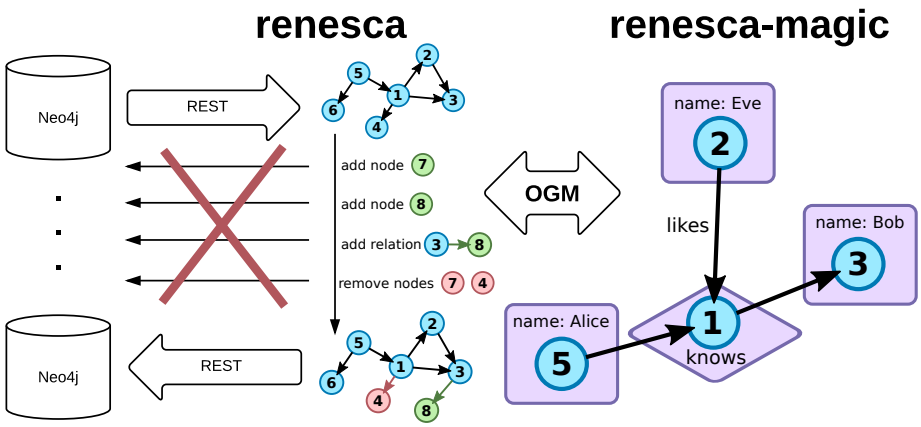

Fig. 1. The renesca-framework uses two libraries: renesca is an abstraction layer that allows to handle graphs over the REST-API of Neo4j. Changes can be done locally and persistence can be deferred as a Unit of Work. renesca-magic is a type-safe wrapper for the low-level property-graph model of Neo4j.

3 Accessing Neo4j Using renesca

Renesca provides the query interface to the Neo4j REST API. It manages submission of prepared statements and returns the results in appropriate data structures. The data is handled using the following concepts:

3.1 Treat Query Results as Graphs instead of Tables

Like the embedded version of Neo4j, with renesca it is possible to query a subgraph from the database and get the result as a graph *or* table data structure. The graph can be traversed like a Scala collection and properties are represented as hashmaps on nodes and relations. The property values can be casted to the expected type. The resulting graph consists of three classes: Node, Relation(startNode, endNode) and Graph(nodes, relations).

3.2 Track Changes and Persist Later as One Unit of Work

When modifying, creating and deleting nodes as well as connecting them with relationships, it is very expensive to submit a REST request for each change. In renesca we track changes and apply all of them at once when persisting the whole graph, with as few queries as possible. This takes fewer REST requests and leaves room for optimization. Changes to properties are also tracked and persisted. This approach allows to pass around the graph structure in the code and after all changes have been applied, persist once.

3.3 Example Code

Listing 1.1. Usage example of renesca

```
// establishing the database connection is left out in this
    example

db.query("CREATE (:ANIMAL {name:'snake'})-[:EATS]->(:ANIMAL {
    name:'dog'})")

val tx = db.newTransaction

// query a subgraph from the database
implicit val graph = tx.queryGraph("MATCH (n:ANIMAL)-[r]->()
    RETURN n,r")

// access the graph like scala collections
val snake = graph.nodes.find(_.properties("name").
asInstanceOf[StringPropertyValue] == "snake").get

// useful methods to access the graph (requires implicit val
    graph in scope)
```

```
// e.g.: neighbours, successors, predecessors, inDegree,
    outDegree, degree, ...
val name = snake.neighbours.head.properties("name").
asInstanceOf[StringPropertyValue].value
println("Name of snake neighbour: " + name) // prints "dog"

// changes to the graph are tracked
snake.labels += "REPTILE"
snake.properties("hungry") = true

// creating a local Node
// (a Node the database does not know about yet)
val hippo = Node.create

// changes to locally created Nodes are also tracked
hippo.labels += "ANIMAL"
hippo.properties("name") = "hippo"

// add the created node to the Node Set
graph.nodes += hippo

// create a new local relation
// from a locally created Node to an existing Node
graph.relations += Relation.create(snake, "EATS", hippo)

// persist all tracked changes to the database
// and commit the transaction
tx.commit.persistChanges(graph)
```

4 The Graph-Object Impedance Mismatch

The renesca framework provides graph query results which return object-graphs that can be worked with procedurally. Usually, entities in the business logic of an application directly correspond to records in the database – these are nodes in the context of graph databases. The abstraction layer renesca-magic generates boilerplate code for Node, Relation and Hyperrelation objects using a high level Scala DSL. This wraps the pure data-graph in an object-graph.

4.1 The renesca-magic Abstraction Layer

When only working with renesca, it is natural to write wrappers for specific nodes that correspond to objects, as these objects always have specific properties. So, instead of looking up in the property hashmap and casting every time, we write a wrapper class which implements typed getters and setters for the needed properties. Relations between objects can be implemented by wrapping the graph traversal with an appropriate getter. Therefore, we extended renesca with simple schema helpers to wrap nodes, relations and graphs.

For large models it can be very error prone to change the model and therefore refactor the boilerplate code. This issue leads to the idea of generating the

boilerplate based on a compactly described model. The code generator is implemented as a set of Scala macros which transform a class-based ER-model DSL to work with the graph database in a type safe way.

Scala macros are hygienic macros and therefore work on abstract syntax trees instead of strings of code. The trees which are read from the schema definition are transformed and directly compiled. The macros analyze the multiple inheritance hierarchies and relation graph of the ER-Model to decide which labels to set on nodes and which getters / setters to generate. This is the heart of renesca-magic. Explaining it here is beyond the scope of this article. The interested reader can examine the code online[9].

The following subsections explain how renesca-magic transforms the model definition in order to map it to the property-graph model used by the database.

Labels The names of the node and relation definitions are directly translated to labels and relation-types of the property graph model.

Properties Renesca-magic generates wrapper classes and factories for nodes and relations. Both can have properties which can be primitives or optional primitives. The properties can be immutable by writing them with a *val* and mutable by using a *var*. Default values can be specified with an assignment of an expression which is evaluated on creation. The classes are generated with getters and setters, taking mutability and optionality into account. The factories provide methods to wrap existing nodes or relations and methods to create new instances with the required properties and the optional ones as default parameters.

Graphs There is a wrapper for the whole graph, which provides access to the different types of nodes and relations contained in the graph. This graph can be persisted like the graph from renesca.

Relations and Neighbors The wrapper for relations takes two additional parameters. The start node and the end node of the directed relation. This triggers the generation of accessors in the start node and end node wrappers to access neighbors over this relation in both directions.

Multiple Inheritance When using the same property over and over again on different types of nodes it makes sense to define it only once in a trait and compose it into all the needed nodes by inheritance[10]. This helps to keep the schema definition DRY (Don't repeat yourself). The name of the trait is added to the list of labels. Like in OOP (Object oriented programming), all children of the trait can be handled as the same type. This allows to work with collections of nodes sharing the same properties. There are also traits for relations with the same functionality.

[9] `https://github.com/renesca/renesca-magic`

[10] Scala allows for multiple inheritance by using traits.

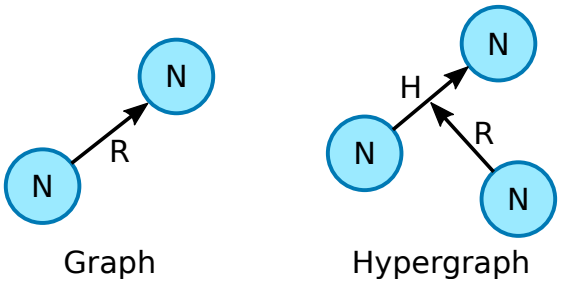

Fig. 2. *left:* A regular graph connects two nodes (N) using a relation (R). *right:* A hypergraph allows for hyperrelations (H) which can be connected to a Node (N) itself by a regular relation (R). (H) and (R) together form a hyperedge in mathematical terms.

Hyperrelations Hyperedges in mathematical terms are edges which connect an arbitrary set of vertices. In renesca-magic we define Hyperrelations as relations which get all characteristics of a node. They can be used as a drop-in replacement for nodes and relations. Hyperrelations can therefore connect two nodes and be connected with other nodes (see Figure 2). So this is a specialized form of the mathematical definition. Internally in the generated code they are represented by a node with an incoming and outgoing relation.

4.2 Example Boilerplate Code

Since these benefits are hard to imagine without examples we demonstrate the benefits of using renesca-magic in a short artificial example (see Listing 1.7 in the appendix). We want to have a simple two node-type graph with one relationship. A class *Animal* has a relationship *Eats* with a class *Food*. Each class has getter and setters to access its neighbors and its properties. We omitted any comments in the generated code to not blow up the code unintentionally, yet we get 50 lines of boilerplate.

The amount of boilerplate needed to represent and access such a simple relationship is far too large. Using a Scala-Macro we can reduce the 50 lines of code to a mere 13 lines of code — including comments (see Listing 1.2).

Listing 1.2. Macro code for our example `(Animal)-[eats]->(Food)`

```
import renesca.schema.macros

@macros.GraphSchema
object ExampleSchemaWrapping {
  // Nodes get their class name as uppercase label
  @Node class Animal { val name: String }
  @Node class Food {
    val name: String
    var amount: Long
```

```
  }
  // Relations get their class name as uppercase relationType
  @Relation class Eats(startNode: Animal, endNode: Food)
}
```

This short example also shows how to use properties that are immutable (using *val*) and mutable (using *var*). Accessing this graph is now equally simple (see Listing 1.3).

Listing 1.3. Creation of objects and changing properties

```
val snake = Animal.create("snake")
val cake = Food.create(name = "cake", amount = 1000)
val eats = Eats.create(snake, cake)

cake.amount -= 100
```

4.3 Additional Features Provided by renesca-magic

Wrapping induced subgraphs The framework renesca-magic supports wrapping of induced subgraphs from a schema. By using such a wrapper (i.e. by using graph) renesca-magic automatically generates accessors for each node and relation. This comes with the benefit of traversing an induced sub-graph with methods of the object itself. In our example (see Listing 1.4) we can see that the relations between specified nodes will be induced.

Listing 1.4. Wrapping an induced subgraph

```
import renesca.schema.macros

@macros.GraphSchema
object ExampleSchemaSubgraph {
  @Node class Animal { val name: String }
  @Node class Food {
    val name: String
    var amount: Long
  }
  @Relation class Eats(startNode: Animal, endNode: Food)

  // Subgraph induction
  @Graph trait Zoo { Nodes(Animal, Food) }
}

// Usage begins here
import ExampleSchemaSubgraph._

val zoo = Zoo(db.queryGraph("MATCH (a:ANIMAL)-[e:EATS]->(f:
    FOOD) RETURN a,e,f"))
val elefant = Animal.create("elefant")
```

```
val pizza = Food.create(name = "pizza", amount = 2)
zoo.add(Eats.create(elefant, pizza))
zoo.animals // provides Set(elefant)
zoo.relations // provides Set(elefant eats pizza)
db.persistChanges(zoo)
```

4.4 Traits and Relations to Traits

Since Scala allows for multiple inheritance using traits, we can map our relations to traits instead of classes. This allows for polymorphic relations between nodes that share common traits (see Listing 1.5, line 15).

Listing 1.5. Multiple inheritance using traits

```
@macros.GraphSchema
object ExampleSchemaTraits {
  // Inheriting Nodes receive their name as additional label
  @Node trait Animal { val name: String }

  // Node with labels FISH and ANIMAL
  @Node class Fish extends Animal
  @Node class Dog extends Animal

  @Relation trait Consumes {val funny:Boolean}

  // Relations can connect Node traits
  // instead of defining separate relations
  // for Fish and Dog explicitly
  @Relation class Eats(startNode: Animal, endNode: Animal)
      extends Consumes
  @Relation class Drinks(startNode: Animal, endNode: Animal)
      extends Consumes

  // Zoo contains all Animals (Animal expands to child nodes)
  @Graph trait Zoo { Nodes(Animal) }
}
```

4.5 Hyperrelations

We can use hypergraphs in renesca-magic by masquerading a node as a hyperrelation (see Fig. 2). In our example (see Listing 1.6, line 13) we model an online document system where articles are annotated with tags. The relation *tags* relates an article and a tag. The relation itself can now be in a relation *supports*, which can store who is supporting which tagging-action — thus a (tag, taggable)-tuple.

Listing 1.6. Multiple inheritance using traits

```
@macros.GraphSchema
object ExampleSchemaHyperRelations {
  @Node trait Uuid { val uuid: String = java.util.UUID.
      randomUUID.toString }
  @Node trait Taggable
  @Node class Tag extends Uuid { val name:String }
  @Node class User extends Uuid { val name:String }
  @Node class Article extends Uuid with Taggable { val
      content:String }

  // A HyperRelation is a node representing a relation:
  // (n)-[]->(hyperRelation)-[]->(m)
  // It behaves like node and relation at the same time
  // and therefore can extend node and relation traits
  @HyperRelation class TaggingAction(startNode: Tag, endNode:
       Taggable) extends Uuid
  // Because these are nodes, we can connect a HyperRelation
      with another Node
  @Relation class Supports(startNode: User, endNode:
      TaggingAction)
}
```

5 Performance Evaluation

In order to asses the impact of using an additional abstraction layer on top of Neo4j, we measure runtimes of the usage example from the renesca documentation.

In the benchmark example the database already contains two nodes connected by one edge. The example consists of two transactions. In the first transaction we query both nodes and set an additional label and property on one of them. Then we create another node and connect it to the previously modified node. In the second transaction we query one node and add a property. The first trial uses change tracking provided by the renesca library, while the native implementation uses explicit Cypher-Queries to do the same reads and modifications to the database[11].

5.1 Method

We run both implementations 2000 times on the same hardware with a local Neo4J 3.0.3 instance. We measure runtime and compare results using a Welch unpaired sample test. We report results using 95% confidence interval and test-result against a significance level of $\alpha = .05$. This means that we have a 95% chance of missing an existing difference in our sample. Since runtime data is

[11] The Benchmark is available at: `https://github.com/renesca/renesca-benchmark`

often not normally distributed, we also apply BCa bootstrapping to verify our test results.

5.2 Results

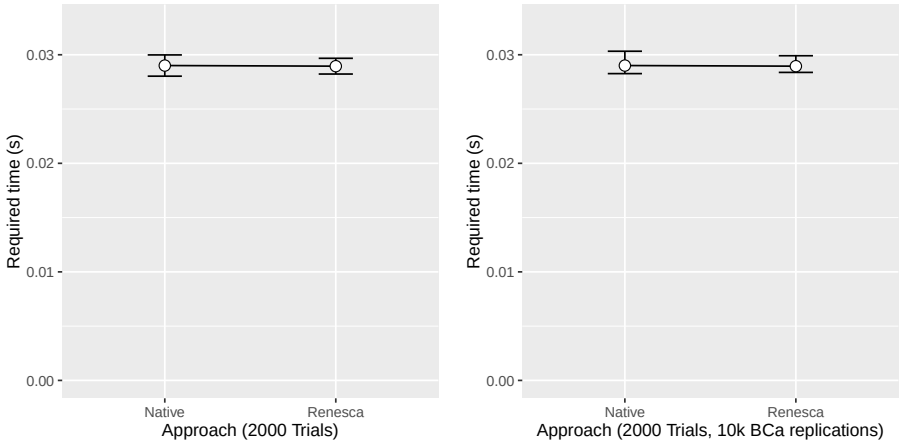

Fig. 3. Performance evaluation of renesca against a native implementation. Error bars denote 95% confidence intervals. The left image shows means and CI, the right image shows bootstrapped means and CIs with 10,000 BCa[12] replications.

After 2000 trials we could not detect a significant difference (see Fig. 3) in performance using a Welch two-sample test comparing means in required time ($t(3683.2) = -0.09142$, $p = .9272$, n.s.). Renesca has an average runtime of $M_R = 0.02894985$s ($SD_R = 0.0165216$S), while the native implementation has a mean runtime of $M_N = 0.02900663$s ($SD_N = 0.02232678$s).

The bootstrapped results confirm our results, even for our heavily skewed sample (i.e. long-tail distribution). Uncertainty only shifts to longer times.

5.3 Discussion

We evaluate our frameworks with very simplistic means, to ensure no dramatic overhead is generated from them. We can show that no meaningful overhead comes from using renesca in our examples. This benchmark is by far not extensive, yet only explorative in nature. But it is necessary to keep in mind that for any given abstraction layer a benchmark can be generated that brings the benefits of the layer to a halt. Direct API access is always faster than through any abstraction layer. The benefits are more sought in clever optimization (i.e. removing redundant operations on higher levels of abstraction), maintainability and increases in developing speed. The trade-off on different aspects in efficiency

(runtime vs. development) must be balanced to attain an effective abstraction layer.

6 Fields of Application

We see potential of using renesca in applications with complex data models which are not easily represented with ORMs. For example:

- cliques of Entities where each entity can be related to any other entity in the clique. This usually results in an quadratic amount of N:M relations in relational models.
- Relations to other Relations (Hyperrelations)

These examples describe advantages of graph databases in contrast to relational databases in general. They are not limited to renesca-magic, but the boilerplate generation allows to handle them with the same usability as ORMs for relational databases.

6.1 Argument Mapping Systems

Using all the features of renesca we implemented a real-life online discussion system that relies on a hypergraph-based data structure to organize its data. The system uses tags (similar as in Listing 6) to organize tagging and voting. The db-schema required to write 266 lines of code (LOC) including comments. Using renesca-magic it generated 2,739 LOC of boilerplate code without comments. This shows that we can reduce code size to a tenth using renesca-magic.

7 Conclusions and Future Work

In this paper we introduced two frameworks that improve the usability of the Neo4j database with Scala. The framework renesca implements access to the REST API of Neo4j and is the foundation of renesca-magic. The latter implements macros that allow object-graph-mapping (OGM). The amount of written code can be reduced by factor of 10. This improves both maintainability and extensibility. It also improves code-readability and facilitates Scala's power to use multiple inheritance.

Since cypher queries are not type safe, queries can create and retrieve data that does not match an existing object model. This is a source of common pitfalls for developers. Adding a query-parser that ensures type-safety in the query language at compile time could alleviate this burden off the user. At this time of writing Neo4j 3.0.0 is already released. It brings a high performance binary protocol called Bolt as an alternative to the REST API to access the database over a network. We plan to integrate this binary protocol into renesca to reduce the protocol overhead. We also plan to evaluate renesca using the methodology presented by Jouili and Vansteenberghe [10]. Naturally using an OGM will increase processing time, but determining under which circumstance this plays a role must be identified.

8 Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments on an earlier version of this manuscript. The authors thank the German Research Council DFG for the friendly support of the research in the excellence cluster "Integrative Production Technology in High Wage Countries".

References

1. Dev, H.: Privacy preserving social graphs for high precision community detection. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM (2014) 1615–1616
2. Holzinger, A., Ofner, B., Stocker, C., Calero Valdez, A., Schaar, A.K., Ziefle, M., Dehmer, M.: On graph entropy measures for knowledge discovery from publication network data. In: Availability, reliability, and security in information systems and HCI. Springer (2013) 354–362
3. Singh, M., Kaur, K.: Sql2neo: Moving health-care data from relational to graph databases. In: Advance Computing Conference (IACC), 2015 IEEE International, IEEE (2015) 721–725
4. Mpinda, S.A.T., Bungama, P.A., Maschietto, L.G.: Graph database application using neo4j (railroad planner simulation). In: International Journal of Engineering Research and Technology. Volume 4., ESRSA Publications (2015)
5. Lampoltshammer, T.J., Wiegand, S.: Improving the computational performance of ontology-based classification using graph databases. Remote Sensing **7**(7) (2015) 9473–9491
6. Urma, R.G., Mycroft, A.: Source-code queries with graph databases–with application to programming language usage and evolution. Science of Computer Programming **97** (2015) 127–134
7. Angles, R., Prat-Pérez, A., Dominguez-Sal, D., Larriba-Pey, J.L.: Benchmarking database systems for social network applications. In: First International Workshop on Graph Data Management Experiences and Systems, ACM (2013) 15
8. Holzschuher, F., Peinl, R.: Querying a graph database–language selection and performance considerations. Journal of Computer and System Sciences **82**(1) (2016) 45–68
9. Beis, S., Papadopoulos, S., Kompatsiaris, Y.: Benchmarking graph databases on the problem of community detection. In: New Trends in Database and Information Systems II. Springer (2015) 3–14
10. Jouili, S., Vansteenberghe, V.: An empirical comparison of graph databases. In: Social Computing (SocialCom), 2013 International Conference on, IEEE (2013) 708–715
11. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc. (2002)
12. Davison, A.C., Hinkley, D.V.: Bootstrap methods and their application. Volume 1. Cambridge university press (1997)

A Boilerplate code example

Listing 1.7. Full boilerplate code required to access a single relationship `(Animal)-[eats]->(Food)`

```
import renesca.graph._
import renesca.parameter._
import renesca.parameter.implicits._

case class Animal(node: Node) {
 val label = Label("ANIMAL")
 def eats: Set[Food] = node.outRelations.filter(_.
     relationType == Eats.relationType).map(_.endNode).filter
     (_.labels.contains(Food.label)).map(Food.wrap)
 def name: String = node.properties("name").asInstanceOf[
     StringPropertyValue]
}

object Animal {
 val label = Label("ANIMAL")
 def wrap(node: Node) = new Animal(node)
 def create(name: String): Animal = {
  val wrapped = wrap(Node.create(List(label)))
  wrapped.node.properties.update("name", name)
  wrapped
 }
}

case class Food(node: Node) {
 val label = Label("FOOD")
 def rev_eats(implicit graph: Graph): Set[Animal] = node.
     inRelations.filter(_.relationType == Eats.relationType).
     map(_.startNode).filter(_.labels.contains(Animal  .label
     )).map(Animal.wrap)
 def name: String = node.properties("name").asInstanceOf[
     StringPropertyValue]
 def amount: Long = node.properties("amount").asInstanceOf[
     LongPropertyValue]
 def `amount_=`(newValue: Long) { node.properties.update("
     amount", newValue) }
}

object Food {
 val label = Label("FOOD")
 def wrap(node: Node) = new Food(node)
 def create(amount: Long, name: String): Food = {
  val wrapped = wrap(Node.create(List(label)))
  wrapped.node.properties.update("amount", amount)
  wrapped.node.properties.update("name", name)
  wrapped
 }
```

```
}

case class Eats(startNode: Animal, relation: Relation,
    endNode: Food)

object Eats {
 val relationType = RelationType("EATS")
 def wrap(relation: Relation) = {
  Eats(Animal.wrap(relation.startNode), relation, Food.wrap(
      relation.endNode))
 }
 def create(startNode: Animal, endNode: Food): Eats = {
  wrap(Relation.create(startNode.node, relationType, endNode.
      node))
 }
}
```