UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Master of Science Thesis

Towards a Scalable Generation of Realistic Property Graphs with Arbitrary Schemas

AUTHOR: Xavier Fernández Salas
DIRECTOR: Dr. Josep Lluís Larriba-Pey
CO-DIRECTOR: Dr. Arnau Prat Pérez

Vancouver, October 15, 2018

Acknowledgments

I would like to say thank you to all the people who helped me archiving this goal, specially to my parents, for the enormous effort they made for giving me the opportunity to study and work in whatever I love.

I would also like to express my deepest gratitude to Larri and Arnau, for their expertise, understanding, patience and continuous encouragement. Without their supervision this project would have not been possible.

To my fiancée Laia, for all her support while I was working on this Master thesis, for being so patient with me and for following me to Canada.

I also place on record, my sense of gratitude to one and all, who directly or indirectly or indirectly, have helped me during this project.

For my grandfather Pepe and for my grandmother Fita.
I will never forget you both

Contents

List of Listings

List of Tables

List of Figures

Chapter 1

Introduction

Graphs are the backbone of many modern applications in the fields of social networks [1], fraud detection [2] recommendation engines [3], medicine [4], biology [5] or sociology [6], just to cite a few of them. Due the increasing demand for managing and analyzing graphs, many graph processing technologies have emerged, including graph databases such as *Neo4J* [7], *Amazon Neptune* [8], *Sparksee* [9] or *DGraph* [10], graph analytics frameworks such as *Pregel* [11], *iGraph* [12], or tools like *GraphX* [13], an API build on top of a general cluster computing framework like *Apache Spark* [14].

In order to compare systems and to assist users into choosing the platform that better suits their needs, several benchmarking efforts have been conducted on initiatives such as *LDBC Social Network Benchmark* [15], *Graphalytics* [16], *gMark* [17], *Graph 500* [18] or *LUBM* [19], among many others. Benchmarking graph processing systems is a complex task. Graphs from different domains have characteristics such as degree distribution, *node/edge* types and properties, or community structure, that affect the performance of the queries or of the analysis. This means that a query or algorithm that performs well in a graph might have performance issues on others, thus, benchmarks must put special care on selecting the datasets they will test the systems on.

However, obtaining real graphs with the desired characteristics is really difficult, either due to privacy concerns or because their holders are not willing to share them. Given this situation, benchmarks usually opt to generate graphs synthetically in a way that fits exactly the needs of the benchmark [11]. However, creating a synthetic graph generator to produce realistic graphs at scale is a very time-consuming and error-prone task, and no general frameworks exist for save developers from such burdens.

Additionally, different graph benchmarks accept graphs in different formats, making it compulsory to convert the generated synthetic graphs to those supported one, with all the inconvenience that this fact causes. Finally, we face the same problem if we want to use real-life graphs as models for the graph generation process.

The project related to this thesis, *DataSynth* [20], has the objective of creating a flexible and scalable graph data generation framework to let practitioners generate synthetic property graphs with the desired characteristics, from structural to dimensional. Additionally, it introduces an approach that has barely been considered, the modeling of correlations between properties and the graph structure, by means of a novel property-to-node matching algorithm that already yielded some promising results.

Unfortunately, at the moment *DataSynth* does not have a convenient to use API, where specifying a graph generation plan is currently really difficult. In addition, it only outputs the generated graphs in a single format, so they need to be manually translated if we want to use them as an input for different graph benchmarking tools.

The main contribution of this Masters thesis is providing *DataSynth* with a comprehensive domain specific language (*DSL*) to let users specify their data generation needs in a convenient way. This meta-language, which we called *Babel*, when executed is transformed into a lower level *Intermediate Language* (*IL*) that would be used by *DataSynth* to schedule the execution of the generation process on a state-of-the-art distributed map-reduce framework, such as *Hadoop* [21], *Apache Spark* [14] or *Apache Flink* [22], generating the graph.

On the other hand, in order to solve the graph format issue, an application/library named *Gnormalizer* was also created for extracting the structure of already existing graphs, and outputting the graph on a given format. This makes it possible to convert already existing graphs to a format that graph analytics tools can read, to obtain their parameters for their use on imitating the input graph with *DataSynth*. It allows for a simple integration with already existing graph benchmarking tools, converting the graphs to the required format just after the graph generation.

The rest of the document is structured as it follows:

- In Chapter 2, we provide an overview of the related project, *DataSynth* [20], and explain the way this Master thesis project integrates with it.

- In Chapter 3, we introduce the User API used for describing graph generation plans while providing the technical details of this metalanguage implementation.
- In Chapter 4, we talk about the Intermediate Language and explain the design decisions we made for it, detailing its syntax an capabilities.
- In Chapter 5, the complimentary library used for transforming graphs between formats and for extracting its structure is introduced.
- Finally, in Chapter 6 the conclusions are summarized and directions for future research areas are given.

Chapter 2

Background

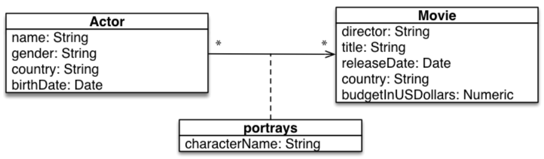
The project related to this thesis, *DataSynth* [20], is a framework for property graph generation with customizable schemas and characteristics. This framework has the main objective of assisting benchmark designers in generating property graphs efficiently at scale, saving them from implementing their own generators.

This chapter provides an overview of the framework architecture, its requirements and how this Master's thesis components will integrate with it. In Section 2.1 we review the different requirements a graph generator must fulfill, that is, what aspects of a graph it must be able to mimic in order to be practical for real benchmarks. Section 2.2 provides an overview on how the graph generation process works in *DataSynth* to meet such requirements. This has direct implications on how the *DSL* and *IL* proposed in this thesis are designed.

The thesis contributions to the framework, which are the *Domain Specific Language* (*DSL*), the *Intermediate Language* (*IL*) and the graph normalization library (*gnormalizer*) and how they integrate with the project are going to be explained in detail in their own chapters.

2.1 Graph generator requirements

In order to generate graphs for real benchmarks a graph generator needs to be capable of reproducing several aspects of a real graph, including its schema, scale, structure and properties' value distributions, among other graph characteristics inherent in a property graph generator. In this section we overview such requirements by means of the running example shown in Figure 2.1, which represents a movie acting network.



Property values distributions:
· **Actor**'s **country** follows a realistic $P_{country}$(X) distribution.
· **Actor**'s **name** is correlated with gender and country, following a feasible P_{name}(X|Y,X).
· **portrays' characterName** is correlated with the **Actor gender** and **country**, following a realistic $P_{characterName}$(X|Y,X)
· **Movie**'s **releaseDate** is greater than the connected actors **birthDate**.
· **Movie**'s **budgetInUSDollars** must be a positive number.
· (...)
Property/Structure correlations:
· **portrays'** degree distribution $D_{portrays}$ follows a power law.
· The **countri**es of the connected **Actor**s and **Movi**es follow $P'_{country}$(X, Y)
· (...)

Figure 2.1: Running example

2.1.1 Schema requirements

A generator should be flexible to adapt to a wide rage of domains, thus it should be able to generate graphs with diverse schemas consisting of multiple *node* and *edge* types and properties, including *edge* cardinalities. The most typical example is a Social Network Graph [1][3], but one might also be interested in generating graphs for other use cases, like a crypto-currency transactions graph [23] or bio-informatics [5].

In our running example, there are two *node* types, `Movie` and `Actor`, and one *edge* type, `portrayed`. `Actor` has the `String` properties `name`, `gender` and `country`, and a `Date` one named `birthDate`. `Movie` has five properties: `Director`, `title`, `releaseDate`, `country` and `budgetInUSDollars`. `director`, `title` and `country` are `String`s, `releaseDate` as a `Date` and `budgetInUSDollars` as a `Numeric`. The `portrayed` *edge* type, which has only the `String` property `characterName`, has a '*many to many*' cardinality between `Actor` and `Movie`.

2.1.2 Structure and Property/Structure correlations requirements

In order to generate graphs with realistic structural characteristics, the graph generator should be configurable with a set of graph structural properties, such as the graph *clustering coefficient*, *diameter*, *community distribution*, etc. This is relevant because graphs from different domains might have really diverse set of structural properties. In addition, the generators must also take into consideration the correlations between the properties and the graph structure.

For instance, in our running example there is a structural characteristic named $D_{portrays}$ that forces the `portrays` *edge* type to follow a power law. On the other hand, the connected `Movie` and `Actor country` should follow a joint probability distribution $P'_{country}(X,Y)$, because `Actor`s are more likely to film in their home country. The resulting graph should have a community structure with communities of `Actor`s from the same `country` sharing the screen in `movie`s from their home `country`.

2.1.3 Property Value Distribution requirements

The properties from the graph *nodes* and *edges* can follow a diverse set of probability distributions and correlations between them. In order to generate realistic graphs, a generator must be as flexible as possible with this fact, allowing the user to manually introduce their own property generators with their own set of distribution probabilities.

An example of this is shown in the running example displayed in Figure 2.1, where the `Actor`'s `name` has to be correlated with his/her `gender` and home `country`. On the other hand, the `portrayed characterName` will follow a $P_{characterName}(X|Y,X)$ distribution with a correlation with the connected `Actor`'s `gender` and `country`. In addition, the distribution requirements also include property constraints, like the `Movie budgetInUsDollars` being a positive number or the `Actor birthDate` being before the `releaseDate` property on the movie he/she participates on.

2.1.4 Scale requirements

Different property graph benchmarks use different notions of scale, like the number of *nodes*, of *edges*, a combination of both or even the graph size on disk. For instance, *Graph500* [18] is based on the number of *nodes*, *LUBM* [19] on the number of *edges*, *Graphalytics* [16] on a combination of *nodes* and *edges*, and the *LDBC Social Network Benchmark* [15] on the size of the generated graph. This means that a property graph generation should be capable of adapting to all these means of specifying the graph scale.

2.1.5 Other requirements

A generator should also support generating graphs at scale efficiently, so we can for instance generate large realistic property graphs with similar or even larger sizes than real life ones. The intention is also to support graph generation on distributed settings, because generating or storing large graphs efficiently might not be possible on single node machines. Thus, the generation process should be adaptable to distributed computing frameworks as *Apache Spark* [14] or *Hadoop* [21].

Finally, controlling a generator of this complexity while defining complex cross-domain schemas is not trivial, and if we are not defining a good interface the chances of this framework to be adopted will drop considerably. For solving this, in this thesis we are creating and implementing a Domain Specific Language (*DSL*) for defining the execution plans and the data schema of the graph to generate. This *DSL*, which we named *Babel*, when compiled, generates an *IL* with all the information a property graph generator needs to generate a realistic property graph at scale on existing computing frameworks.

2.2 Datasynths' Graph Generation Process

In this section we detail the *DataSynth* framework approach for generating property graphs fulfilling the requirements introduced in Section 2.1. For doing this, we use the Figure 2.2, which shows an overview of the *DataSynth* framework [20] architecture, and we detail what happens on each step shown there.

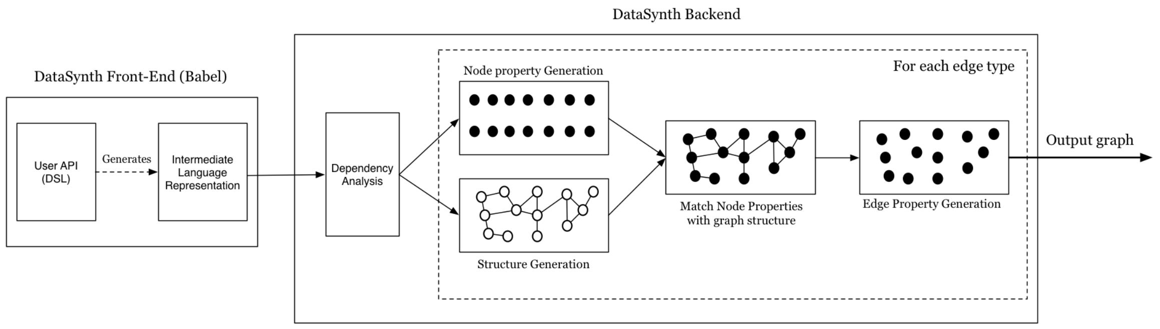


Figure 2.2: DataSynth general approach [20].

First, the user describes the graph requirements on the *DSL*, including all the graph structure, property, property/structure correlations, distributions and scale requirements. This *DSL* generates an *Intermediate Language* (*IL*) when compiled, that describes how the resulting graph should look like and the necessary information to generate it. This is the *DSL* and *IL* designed in this thesis.

The graph generation process starts by analyzing the schema described on the *IL*, and it outputs a dependency graph with the tasks the *generator* needs to execute in order to generate the graph. There are three types of tasks: *Generate property* (either *node* or *edge* property), *Generate structure* and *Graph matching*. These tasks are fully composable and their execution is completely independent if there is no direct dependency between them, so the back-end will be able to execute the independent ones in parallel. The *Generate property* task relays on a *Property Generator* (*PG*) for completing its mission, and the *Generate structure* does the same using a *Structure Generator* (*SG*).

Following the workflow described on the dependency graph, the generator first creates the instances of each *node* type independently with their own set of properties. Secondly the graph structure is generated following the degree distributions specified by the user in the *DSL* and present later in the *IL*. As a third step, the *edges* and the *nodes* are matched following the previously generated structure guided by the Property/Structure correlations defined by the user. Finally, the *edge* properties are generated considering the connected *nodes* properties and the user joint probability distributions.

In Section 2.2.1 we are detailing the storage structure supporting this graph generation process on a distributed setting, Section 2.2.2 overviews the generators used for generating the properties for all the graph *nodes* and *edges*, Section 2.2.3 provides detail on the graph structure generators, Section 2.2.4 talks about the property generation tasks, Section 2.2.5 describes to graph generation task and the Section 2.2.6 does the same for the graph matching ones.

2.2.1 Generator Data Model

One of the biggest challenges of *DataSynth* is making it as much scalable as possible, even relaying on multiple hosts for running the different generation tasks. For accomplishing this objective *DataSynth* uses a distributed data storage with two types of tables, *Property Tables* (*PT*s) and *Edge Tables* (*ET*s). *PT*s are used for storing the generated graph elements properties, and there is one per *node* and *edge* property, and it has just two columns, one for the *node* or *edge* identifier and another for the generated `value`. The *ET*s are being used for storing the graph *edges* without any of their properties, and there is a table of this kind per edge type, and the table will had three identifiers, the one from the *edge* and the one from the two connected *nodes*.

For instance, in our running example, a *PT* is created for the `Actor`'s `name`, `gender`, `country` and `birthDate`, for the `Movie`'s `director`, `title`, `releaseDate`, `country` and `budgetInUSDollars`, and for `Portrays`'s `characterName`. Additionally, we would have a single *ET* with the identifiers of each *edge* connected `Actor` and `Movie` and the identifier from the *edge* itself.

2.2.2 Property Generators (PGs)

The *Property Generators* (*PGs*) have the responsibility of generating a property for a *node* or for an *edge*. *PG*s are defined using the *DSL* and specify how properties are generated. The overall approach intends them to be pluggable and composable, and we also intend them to be usable in a distributed setting with reproducible results. In the *DSL* chapter Section 3.2.1 we show how *PG*s are defined in *Babel*.

In the way *DataSynth* is implemented a *PG* only needs to implement the method with signature "`def run(id: Long, r: (id) => Long, ...): T`" per each property. This method generates the property values for a given *node* or *edge* instance. The parameter "`id`" represents the instance of the *node* or the *edge* for which we are generating the property. The parameter "`r`" corresponds to a deterministic hashing function that generates a random number using the provided "`id`". Additionally, the *PGs* "`run`" method might also accept more parameter if we need to specify a correlation between property values.

Thanks to the fact that the function parameters are only the `id` and a deterministic function with the same `id` as input, we can recreate any property value in place just by knowing the `id`, the hashing function and its generator. Additionally, if the "`run`" method is implemented in a *deterministic* and *pure* way, we are able to completely parallelize the generation tasks, even if we are working in a distributed setting.

The "`run`" method is flexible enough to generate the properties for our running example. For instance, if we want to generate the `Actor`'s `gender` and `country` and for the `Movie`'s `director` we would use a generator with a "`run`" method with the following signature: `def run(id: Long, r: (id) => Long): String`. An example generator for a property correlated with another, like `Actor`'s `name`, which depends on his/her own `gender` and `country`, has the following "`run`" function signature: `def run(id: Long, r: (id) => String, gender: String, country: String): String`. In Section 3.2 we detail this type of function would be implemented if we use the *DSL*.

2.2.3 Structure Generators (SGs)

The *SG*s work in a really similar way as the *PG*s, but have the method with function signature "`def run(n: Long): ET`", which generates an *ET* with size `n` (number of *edges*). In addition, in case we want to generate the *ET* with a specific quantity of *edges*, the generator must implement the following helper function that returns the required input number of *nodes* for the "`run`" method: " `def getNumNodes(numEdges: Long): Long`"

In the API chapter Section 3.2.4 we are going to show how we define the *SGs* in the *DSL*.

2.2.4 Generate Property Tasks

The *Generate Property* task type has the objective of generating the properties for the graph *nodes* and *edges*, and that will be done calling an associated *PG*. Each call to the associated *PG*s "`run`" method will only generate a property, so we will need to call the *PG* as many times as the quantity of properties to generate with the corresponding parameters. As explained in Section 2.2.2 we can perform the calls in a parallel, even if the properties are correlated, thanks to the fact that we can regenerate the dependent properties in place using the corresponding generator.

2.2.5 Generate Structure Tasks

The *SG*s have the main responsibility of generating the graph structure considering the structure distributions specified by the user in the *DSL*. The *SG*s, when called, return an *ET* with all the *edge*s we require from an *edge* type. At this point thought, the *edges* properties will still not be considered, since they are generated after matching the *node* identifiers with the graph structure in the *Graph Matching* step.

2.2.6 Graph Matching Tasks

Once the graph structure and the *node* properties are generated, we need to match the generated *nodes* with the graph structure in such way that the joint property-structure probability distributions $P(X,Y)$ defined by the user are preserved. This distribution represents the probability of picking a random *edge* of the graph with values X and Y on its connected *node* pair. If there is no correlation at all between the *edge* type and a property, the matching is done randomly. For specific details on how this process works, please refer to [20].

Once the matching is done, the generator calls the *PG* task(s) to generate the missing *edge* properties, which can take into account the properties of the connected *nodes* by means of their identifiers and the corresponding *PG*s.

Chapter 3

Babel: A domain specific language for graph generation

Providing an user-friendly API is really important in order to simplify the usage of the framework, decreasing the development time, so the user is capable of generating complex graph structures easily.

The API is the bridge between the user and *DataSynth* and a lot of effort is put in its completeness. We can describe a rich set of graphs with it, implementing a diverse set of properties and distributions, being as much understandable, expressive and unambiguous as possible.

As a main requirement, the API needs to be expressive enough to be able to express all the graph generation requirements we previously described in Section 2.1. Additionally, as a non-functional requirement, we have considered that it would be really important for the framework adoption if the API is *Integrated Development Environment* (*IDE*) compatible, so developers are capable to use their own development tools, with all the amenities and help they provide, for defining the graph generation plans.

For implementing the API we have created a *Domain Specific Language* (*DSL*) for representing property graphs. This *DSL* is written on top of *Scala* [36] using *Macro Annotations* [37] for enriching the user source code at compile time. Using this technologies we reduce the amount of code required for defining a graph structure considerably, furthermore, these technologies are supported by the major *IDE* tools and editors, such as *Emacs* [38], *Visual Studio Code* [39], *Eclipse* [40] and *IntelliJ*

IDEA [41]. We choose *Scala* because it is a programming language specially intended for writing *DSL*s [42][43][44], and the *macro annotations* because they are a state-of-the-art source code enrichment technique in *Scala*. Additionally, the resulting *DSL* syntax fitted perfectly our expectations. On the other hand, the graph normalization library that will be presented in Chapter 5 is also using the same programming language, and it is the reference language in most distributed computing frameworks, such as the *Akka* toolkit [45] or *Apache Spark* [14].

This chapter has the objective of describing how the user can define all the process required for generating the *Intermediate Language* (*IL*) that is then used as input to *DataSynth*. In Section 3.1 we are describing in detail how the properties are defined for the graph *edges* and *nodes*, Section 3.2 shows how the generators are defined and Section 3.3 overviews how the user can describe the graph generation plans.

For this chapter we use the movie acting network running example shown in the Figure 2.1 from Chapter 2.

3.1 Graph Properties Definition

Each graph is different, and each one of them has different properties on the *nodes* and/or in the *edges*. In order to support graph generation of a diverse set of domains, the schema of *nodes* and *edges* are defined as classes, with each class with its own sets of properties. Every graph is formed by a set of *nodes* and *edges* with their own set of properties, including an identification one that is used for distinguishing them uniquely.

In this section we synthesize how to describe the *nodes* and *edges* properties in the *DSL*. In Section 3.1.1 we are detailing the programming language types that can be used in the *DSL* for defining the individual graph *node* or *edge* properties, Section 3.1.2 shows how we can define custom types in *Babel*, Section 3.1.3 details how to model a *node* type properties and Section 3.1.4 does the same for the *edge* ones.

3.1.1 Supported Primitive Property Types

The API supports any of the following native primitive types:

- Character types: `Char`
- String types: `String`
- Date types: `LocalDate`
- Timestamp types: `Instant`, `OffsetDateTime` and `ZonedDateTime`
- Duration types: `Duration`
- Interval types: `Interval`
- Integer Number types: `Byte`, `Short`, `Int`, `Long` and `BigInt`
- Decimal Number types: `Float`, `Double` and `BigDecimal`

All these types, once they are serialized to the IL, respect the same constraints as in the *Scala* programming language. This means that for instance a *Scala* `Int` is represented in the IL as a 32-bit signed *Integer* number. In Chapter 4 we detail in depth how these types are being serialized to the IL.

If we want to represent an optional type, we use the *Scala* `Option` class and wrap the primitive type inside it. For instance, if we want the `gender` property to be nullable in the running example `Actor`, we represent the type as: `Option[String]`. Furthermore, enumerations and other Scala `class` objects may also be nested, as long as all their parameters are from types supported by the framework.

3.1.2 Custom Primitive Types

There are several use cases where we may want to define a custom primitive type, for being more concise and/or for simplifying the generators. For instance, it does not make a lot of sense having empty '`String`s' on the generated *nodes* and *edges*, like on the running example `Movie`, where we definitively always want to have some characters on its `title`, `country` and `director`.

For accomplishing this requirement, we take advantage of the *Scala* '*literal types*' [46] and '*implicit conversions*' [47] features for defining them. For instance, if we want to define a NonEmptyString, we may implement in the *DSL* as in Source Code Snippet 3.1.

```scala
import babel.types._
import babel.types.primitives.constraints

val minLength: Int = 1
val minLenghtConstraint = constraints.Numeric.minLength(minLength)

type NonEmptyString = String.withMinLength(minLength)
```

Source Code Snippet 3.1: Example custom primitive type definition

It is worth of notice that implementing a custom type manually doesn't offer a lot of value right now in our current system, because we currently relay entirely on the generator for controlling the value creation with the desired constraints. But, if we invest time in the future for researching about automated generator creation, this *constraint* based custom types can potentially be part of the implementation foundations.

3.1.3 Node Property Model Definition

For defining a *node* type we only need to define a *Scala* `class` annotated with the macro annotation `@node` in front of it. With this annotation, the `class` attributes will become the properties of the *nodes* we want to describe. In our running example we would define the `Actor` and `Movie` *node* property models as in Source Code Snippet 3.8 and the Source Code Snippet 3.7.

```scala
@node class Actor(name: NonEmptyString,
                  gender: Option[NonEmptyString],
                  country: NonEmptyString,
                  birthDate: LocalDate)
```

Source Code Snippet 3.2: Example *node* property model definition for the running example `Actor` *node* type

```
@node class Movie(director: NonEmptyString,
                  title: NonEmptyString,
                  releaseDate: LocalDate,
                  country: NonEmptyString,
                  budgetInUSDollars: Option[Double])
```

Source Code Snippet 3.3: Example *node* property model definition for the running example `Movie` *node* type

3.1.4 Edge Property Model Definition

Defining an *edge* type is similar, but we use the annotation `@edge` instead, with the type of the source and target *nodes* as inputs for the annotation. In the same way as with the `@node` annotation, the attributes of the class represent the model of the *edges* we want to generate. The Source Code Snippet 3.4 shows an example implementation for the `portrayed` *edge* property model from our running example.

```
@edge(source = Actor, target = Movie, cardinality = ManyToMany)
class Portrayed(characterName: String)
```

Source Code Snippet 3.4: Example *edge* property model definition for the `portrayed` *edge* type from our running example

3.2 Generator Definition

In this section, we detail how we describe the generators in *Babel*, considering the generation requirements we described in Section 2.1.

In Section 3.2.1 we detail how we define a *PG* in *Babel*, Section 3.2.2 shows how we combine these *PG*s definition to form a *node*, Section 3.2.3 does the same for the *edges*, Section 3.2.4 overviews the *SG* definition in the *DSL* and in Section 3.2.5 we detail how we define a generator for the whole graph, joining the *PG*s and the *SG* for forming a graph.

3.2.1 Property Generators (PG)

For defining a property generator, we must accomplish all the requirements from Section 2.2.2 from Chapter 2. This means that, for describing the graph, we must define a generator for all the properties in every single *node* and *edge* model previously defined. Additionally, we must define a generator for joining the properties in each graph *node* and *edge*. Finally, the generator "`run`" method should accept the generated *node* or *edge* "`id`", the hashing function in the "`r`" parameter and a flexible number of dependent values in "`dependencies`". Considering all these requirements, we created the interface defined in Source Code Snippet 3.5 that all the property generators need to extend.

```
trait PropertyGenerator[T] {
  def run(id: Id, r: (ID) => T, dependencies: Any*): T
}
```

Source Code Snippet 3.5: Interface that needs to be extended by all the *PG*s.

For defining a property generator we only need to extend the `PropertyGenerator[T]` interface defined in Source Code Snippet 3.5, and provide the implementation we want including its probability distributions.

For instance, in our running example, the `Actor` has a `birthDate` property, and it is really important for generating movie acting communities, because actors are more likely to participate in movies with people from a similar age. An example property generator for this property is displayed in the Source Code Snippet 3.6. In this case, the uniform distribution is used for simplifying the example implementation.

```scala
object ActorBirthDateGenerator extends PropertyGenerator[LocalDate] {

  // This generator does not dependent on any other property,
  // so we are ignoring the 'dependencies' parameter.
  override def run(id: Id,
                   r: (Id) => Long,
                   dependencies: Any*): LocalDate =
    LocalDateGenerator.nextLocalDate(
      hash = r(id),
      min = NOW().minusYears(90),
      max = NOW(),
      distribution = Distribution.Uniform
    )
}
```

Source Code Snippet 3.6: Example definition of a graph *node* or *edge* property generator

3.2.2 Node Generators

For defining a *node* or an *edge* generator we need to implement a generator instance. For doing that, we need to add the `@generator` macro annotation, extending the previously generated model class. This annotation has the responsibility of checking that all *node* properties have an attached generator, yielding a compiler error if we miss any of them, and the *IDE* will show us which ones we still need to implement. On the other hand, the macro will also automatically implement the serialization method '`intermediateLanguage()`' for serializing the generator to the *IL*. In Section 4.5 we are detailing how the *DSL* source code expansion works, and how the generators will be reflected in the *IL*.

It is worth of notice that on *node* or *edge* generators we do not need to implement the "`run`" method, since *DataSynth* will automatically perform the required calls for all the generators with an associated *node* or *edge* property.

The Source Code Snippet 3.8 shows an example implementation of the running example `Actor` *node* generator, using the `ActorBirthDateGenerator` we defined at the Source Code Snippet 3.6. Additionally, the Source Code Snippet 3.7 shows an example `Movie` generator.

```scala
@generator
class MoviePropertiesGenerator extends Actor {

  override val directorNameGenerator: Generator[NonEmptyString] =
    new NameGenerator(
        dependsOn = Seq(countryGenerator)
    )

  override val countryGenerator: Generator[NonEmptyString] =
    new CountryGenerator()

  override val releaseDate: Generator[LocalDate] =
    new MovieReleaseDateGenerator()

  override val titleGenerator: Generator[Int] =
      new MovieTitleGenerator(
          dependsOn = Seq(directorNameGenerator, countryGenerator)
      )

  override val budgetInUSDollarsGenerator: Generator[Double] =
    new BudgetInUSDollarsGenerator(
        dependsOn = Seq(releaseDateGenerator, countryGenerator)
    )
}
```

Source Code Snippet 3.7: Example *node* generator for the running example `Movie` *nodes* properties

```
@generator
class ActorPropertiesGenerator extends Actor {

  override val nameGenerator: Generator[NonEmptyString] =
    new NameGenerator(
      dependsOn = Seq(genderGenerator, countryGenerator)
    )

  override val genderGenerator: Generator[Option[NonEmptyString]] =
    new GenderGenerator()

  override val countryGenerator: Generator[NonEmptyString] =
    new CountryGenerator()

  override val actorBirthDateGenerator: Generator[LocalDate] =
    new ActorBirthDateGenerator()
}
```

Source Code Snippet 3.8: Example *node* generator for the running example `Actor` *nodes* properties

3.2.3 Edge generators

Generating the *edges* properties is done in the same way as with the *node* generators from Section 3.2.2, but extending the *edge* model class instead. The Source Code Snippet 3.9 shows an example *edge* generator for the running example `portrayed` *edge* model. The `@generator` macro will automatically enrich the class for forcing us to implement all the generators for each of the *edge* properties, and it will also automatically implement the method for serializing the *edge* generator to the *IL*.

```
@generator
class PortrayedPropertiesGenerator extends Portrayed {

  override val characterNameGenerator: Generator[NonEmptyString] =
    new characterNameGenerator()
}
```

Source Code Snippet 3.9: Example *edge* generator for the running example `portrayed` properties

3.2.4 Structure Generators (SGs)

Once we have implemented the *node* generators, we can define the graph structure generator extending the `StructureGenerator[T]` interface shown in the Source Code Snippet 3.10, that is based on the requirements from Section 2.2.3. In order to make a simpler implementation, we can out-source the structure creation to an existing graph generator, such as RMAT [48], saving us from the burdens of implementing the graph structure generator manually.

```
trait GraphStructureGenerator[T] {
  val run(n: Long): T
  val getNumNodes(numEdges: Long): Long
}
```

Source Code Snippet 3.10: Interface that needs to be extended by all the *SG*s

3.2.5 Graph Generator

Once we have defined all our *PG*s and *SG*s we can generate the graph generator itself that will bind all of them together. This generator will be used for pointing to *DataSynth* the target elements of the different graph generation phases that we detailed in the Section 2.2 from Chapter 2. The Source Code Snippet 3.12 shows the interface that we need to implement for accomplishing this.

```
trait GraphGenerator {
  override val nodePropertyGenerators: PropertyGenerator[_]
  override val edgePropertyGenerators: PropertyGenerator[_]
  override val graphStructureGenerator: GraphStructureGenerator[_]
}
```

Source Code Snippet 3.11: Example graph generator

An example implementation is shown in Source Code Snippet 3.12 were we define our running example graph generator. In this example we are referencing the *PG*s defined in the Source Code Snippet 3.7, the Source Code Snippet 3.8 and the Source Code Snippet 3.9, and a *SG* implementing the interface from the Source Code Snippet 3.10.

```
@generator
class MovieActingNetworkGenerator extends GraphGenerator {

  override val nodePropertyGenerators =
    Seq(
      ActorPropertiesGenerator(),
      MoviePropertiesGenerator()
    )

  override val edgePropertyGenerators =
    Seq(PortrayedPropertiesGenerator())

  override val graphStructureGenerator =
    MovieActingNetworkStructureGenerator()
}
```

Source Code Snippet 3.12: Example graph generator

3.3 Intermediate Language Generation

Once we have implemented the graph generators, we are able to define the execution plan, outputting the *Intermediate Language* we need to execute in any of the framework back-ends.

For defining the plan execution, we only need to a use the *Builder* class `Babel`, then we may use its API for defining how we want the graph to be generated and its quantities.

The Source Code Snippet 3.13 shows a simple graph execution plan definition for our running example. It uses the `MovieActingNetworkGenerator` defined at the Source Code Snippet 3.4, outputting the *Intermediate Language* that will be used for generating the graphs. In Chapter 4 we are going to provide detail about the IL, and show example outputs the API is going to generate.

```
Babel
  .builder()
  .generate(MovieActingNetworkGenerator)
  .build()
  .intermediateLanguage()
```

Source Code Snippet 3.13: Example execution plan generator

Chapter 4

Intermediate Language

One of the goals of *DataSynth* is to be decoupled from the actual technology executing the data generation process, either if it is implemented on distributed computing frameworks such as *Apache Spark* [14], *Hadoop* [21] or *Apache Flink* [22], or in single node implementations. Unfortunately, implementation approaches may differ completely, using, for instance, different programming languages or even different programming paradigms. A system implementing all the stack at once is really complex, since it becomes a code monolith really difficult to program and maintain.

Creating an intermediate language representation enables us to decouple entirely the *DSL* from the execution implementation, making the code more modular and easier to maintain. Additionally, by doing this separation, anyone is capable of creating their own front-end or back-end implementation without having to code everything, only needing to stick to framework contracts. This strategy is broadly used on the vast majority of the new state-of-the-art compilers, having the clearest examples on the *Bytecode* in the *JDK* [24], or the *IR* in the *LLVM* compiler [25].

This chapter explains in detail the design of the framework Intermediate Language (*IL*), including the technological decisions. Section 4.1 introduces the base of the language syntax and how the language itself is structured, Section 4.2 explains the type of fields we can use on the generated graph *nodes* or *edges*, Section 4.3 describes how the graph structure models are defined, Section 4.4 describes the way of defining a graph generator and overviews how the IL is generated from the *DataSynth DSL* we described in Chapter 3.

4.1 Language Syntax and Structure

In order to accelerate the back-end development, an already existing data communication language like JSON [26] is being used. In this way, development time is minimized, given that almost every programming language has mature libraries for handling this type of object notation [26]. On the other hand, most developers are familiar with JSON, so they would be capable of understanding the base syntax easily, and they would have the possibility even to code the intermediate code directly, at the cost of losing the type safety and the IDE auto-completion provided by the *DSL*.

The intermediate language output is divided into modules, each one of them containing all the required information from a category that the back-end implementations need to process.

The language contains the following modules, which are processed in order:

- `GraphNodeAndEdgePropertyModels`: Contains the structure definition of the graph *nodes* and *edges*. For instance, all the `Actor`, `Movie` and `portrayed` from the running example introduced in Figure 2.1 from Chapter 2 have its own associated model. Each model will contain a list of the properties associated to that *node* or *edge*, like the running example `Actor`, that has the `name`, `gender`, `country` and `birthDate` properties.

- `Generators`: Contains the definition of a set of generators that, when executed, will generate the graph *node* and *edges* properties following the models described in the ”Property Models” section. As described in Chapter 2, every single graph property will have its own generator, like the running example `Movie`, that has a generator for each of its `director`, `title`, `releaseDate`, `country` and `budgetInUSDollars`.

The Source Code Snippet 4.1 displays the base Intermediate Language structure. The models defined in Section 4.3 belongs to the `"graphNodeAndEdgePropertyModels"` field and the generators defined on Section 4.4 are placed inside the `"generators"`.

```
{
  "graphNodeAndEdgePropertyModels": { /*.. Model definitions . */},
  "generators": { /*... Generator Definitions ..*/ }
}
```

Source Code Snippet 4.1: Structure of all the *IL* generation plan definitions

4.2 Type system

Data generation needs to adapt to the model independently from the implementation platform, so we need to define the models in the intermediate language describing their types and constraints. Any back-end implementation implementing this language should adapt all the primitive types we are defining in this section to its own type and library ecosystem, but, at the same time, forcing the constraints defined in the Intermediate Language.

4.2.1 Primitive Types

The simplest atom of a programming language type system are the primitive types, which represent a value. The combination of these values types can be used for forming graph *nodes* or *edges*, which may represent any entity or relationship we may think of. In the intermediate language we do not only consider the conventional types, but also a set of nice-to-have types as part of the core type system, forcing this metalanguage implementations to be more flexible and convenient to use. In summary, the primitive types we have added are: Numeric Types, Identifier Types, Time Types, Character String Types and the Enumeration types.

In Section 4.2.1.1 we are going to detail the Numeric types, Section 4.2.1.2 overviews the identifier types, Section 4.2.1.3 the time types, Section 4.2.1.4 the Character String types and Section 4.2.1.5 details the enumeration types.

4.2.1.1 Numeric types

Most traditional programming languages include a set of integer types which normally are `Byte` (1 bytes), `Short` (2 bytes), `Int` (4 bytes), `Long` (8 bytes) and decimal number ones like `Float` (4 bytes) and `Double` (8 bytes) [27][28][29]. In all these cases the numbers can be negative, but in some languages they also have their non-negative counterparts, which are usually referred as 'unsigned number types' [27][29]. There are programming languages that also define number types with arbitrary precisions, so developers can specify their required thresholds.

For these types we decided to follow an approach similar to the one used on the *Ruby* [30] programming language, that abstrasts the numeric types to the user, so the user only works directly with a single type, so she/he only needs to define the thresholds and let the compiler decide the internal number representation [31][32]. In our case, all numbers are represented with the `Number` type and the thresholds are defined using constraints.

For the first version of the Intermediate Language we have implemented the following numeric constraints:

- `MinValue`: Represents the minimum accepted value for the property.
- `MaxValue`: Represents the maximum accepted value for the property.
- `NumberDecimals`: If we want the numeric property to have decimals, we need to set this value to set the precision we need.

For instance, a '*16-Byte unsigned integer*', which would be represented on $C++$ as an '`unsigned short`' [27], would be represented on the IL as on Source Code Snippet 4.2.

```
{
  "exampleNumberField": {
    "Type": "Number",
    "Constraints": {
      "MinValue": 0,
      "MaxValue": 65535,
      "MaxNumberDecimals": 0
    }
  }
}
```

Source Code Snippet 4.2: Example graph *node* or *edge* model number field definition

4.2.1.2 Identifier Types

If we want to identify a generated graph *node* or *edge* uniquely from the rest, we probably want an identifier specially designed for distributed environments settings as the Universally Unique Identifier (UUID) [33].

This type of implementation cannot be simpler to define on the IL due to the fact that it does not have any associated constraint because most of the times these types are auto-generated, being the definition as simple as the one shown on the Source Code Snippet 4.3.

```
{
  "exampleIdentifier": {
    "Type": "UUID"
  }
}
```

Source Code Snippet 4.3: Example graph *node* or *edge* identifier property definition

4.2.1.3 Time types

Time types are not usually considered a primitive type in almost any language [27][28][29][35], but we considered them to be primitive ones because we expect them to be present on a lot of different use cases, like for instance, in our running example of Figure 2.1, the `Actor`'s `birthDate` and the `releaseDate` from the `Movies`.

In order to avoid having problems with the time/date formats, the intermediate language sticks to the international standard for time/dates ISO-8601 [34]. We also included the value `"now()"` to represent the current time, so the generators won't be allowed to create elements in the future.

For the first version of the *IL* we included the following two types:

- `Date`: For representing the date, it will be represented as a date in ISO-8601, for instance: `"1905-01-01"`
- `Timestamp`: For including the time of the day into the day. The format will be like a timestamp in ISO-8601, such as: `"2017-03-17T14:02:25.629"`.

Both the `Date` and the `Timestamp` types share the following constraints:

- `minTime`: Minimum time in the ISO-8601 format for the property. For instance, we may have either a `"1905-01-01"`, `"2017-03-17T14:02:25.629"` or `"now()"` for representing the current time.
- `maxTime`: Maximum time for the property in the ISO-8601 format. The same formats as with the `minTime` constraint apply.

The Source Code Snippet 4.4 represents a `Date` model in the *IL* and the Source Code Snippet 4.5 does the same for the `Timestamp` ones.

```json
{
  "exampleDateValue": {
    "Type": "Date",
    "Constraints": {
      "minTime": "1905-01-01",
      "maxTime": "now()",
    }
  }
}
```

Source Code Snippet 4.4: Example graph *node* or *edge* model `Date` property

```json
{
  "exampleTimestamp": {
    "Type": "Timestamp",
    "Constraints": {
      "minTime": "2017-03-17T14:02:25.629",
      "maxTime": "now()",
    }
  }
}
```

Source Code Snippet 4.5: Example graph *node* or *edge* `Timestamp` property

4.2.1.4 Character String Types

Text data generation is really common and, with high probability, is going to be required in most data generation tasks, so it is really important that the implemented system supports this kind of data type. For defining all kinds of text the type `String` is used.

In the first version of the *IL* we accept the following constraints for this type:

- `minLength`: Non negative Integer value for representing the mininum length of the `String` property. For instance, if the value is 2 (and the `maxLength` is arbitrary big) we would accept `"abc"` and `"ab"`, but we will refuse `"a"` and `""`.
- `maxLength`: Non negative Integer value that works in the same way as `minLength` but specifying the maximum length of the `String` property. For instance, if we set it to 3 (and the `minLength` is 0), we would accept `""`, `"a"`, `"ab"`, `"abc"`, but we will refuse `"abcd"` property values.
- `encoding`: Represents the encoding of the `String` to generate, just in case we need to generate special characters not available in the standard, at the moment we only accept the values `"UTF-8"` and `"UTF-16"`.

An example *text* property is displayed in Source Code Snippet 4.6, where we would show the `name` of our running example `Actor`.

```json
{
  "name": {
    "Type": "String",
    "Constraints": {
      "minLength": 1,
      "maxLength": 80,
      "encoding":"UTF-8"
    }
  }
}
```

Source Code Snippet 4.6: Example graph *node* or *edge* model Character String property definition

4.2.1.5 Enumeration Types

In case we want a parameter to be one of the given list of values, we may choose an `Enumeration` type. An example implementation can be found at the Source Code Snippet 4.7. This type will only have the `acceptedValues` constraint, that accepts a list of the accepted values formatted as `String`s.

```
{
  "movieRating": {
    "Type": "Enumeration",
    "Constraints": {
      "acceptedValues": [
        "Outstanding",
        "Good",
        "Mixed",
        "Negative",
        "Horrible"
      ]
    }
  }
}
```

Source Code Snippet 4.7: Example graph *node* or *edge* enumeration property definition

4.3 Model Definition

Once the type system is defined, we can start composing several of these fields for describing the graph *node* or *edge* structure as required. For instance, we can create a structure by nesting its parameters into a model object, indicating if a parameter is required or unique, as it is shown on the Source Code Snippet 4.8. By default, parameters can have duplicates and are required. Finally, the Intermediate language also accepts working with collections of values. For doing that, we need to create the '`Collection`' type, and nest the required model inside it, as shown in Source Code Snippet 4.9.

```
{
  "ExampleModelWithTwoProperties": {
    "firstProperty": {
      "Type": "UUID",
      "isUnique": true
    },
    "secondProperty": {
      "Type": "Number",
      "Required": false,
      "Constraints": {
        "MinValue": 0,
        "MaxValue": 104,
        "MaxNumberDecimals": 0
      }
    }
  }
}
```

Source Code Snippet 4.8: Example graph *node* or *edge* custom model overriding the default uniqueness and optionality settings

```
{
  "ExampleModelWithACollectionProperty": {
    "collectionProperty": {
      "Type": "Collection",
      "Model": "Person",
      "MaxNumberInstances": 14,
      "AllowDuplicates": true
    }
  }
}
```

Source Code Snippet 4.9: Example graph *node* or *edge* model introducing the concept of collections

4.4 Graph Generators

Once the structure of the graph *nodes* and *edges* are present we can proceed defining the generators for them. These generators, when executed, will create an instance of the defined *node* or *edge*. Section 4.4.1 explains how the graph *nodes* and *edges* model definitions are generated and the Section 4.4.2 shows how the graph structure generation is defined.

4.4.1 Graph Property Generator

Defining a *node* or an *edge* generator in the Intermediate Language is a complex task in some cases. So we opened the door for serializing a class generator directly into the AST, the only requirement is that the generator class implements the contract '`PropertyGenerator[T]`' shown in Chapter 3 at the Source Code Snippet 3.5. The Intermediate Language, once the compiled source code is pushed to a public access web repository and the URL to access it will be referenced in the *IL* in the same way as in the example Source Code Snippet 4.10.

```
{
  "CodeBasedGeneratorExample": {
    "type": "property-generator",
    "generatorFor": "ActorGenerator",
    "runtime": "JVM-8",
    "language": "Scala-12",
    "s3-bucket": "https://babel.s3-eu-west-1.amazonaws.com/",
    "s3-file": "actor-generator.class"
}
```

Source Code Snippet 4.10: Example code-based generator definition on the Intermediate Language

4.4.2 Graph Structure Generator

The first option is to provide the target parameters we want to obtain in the final graph and let the back-end implementation do the rest. In this case, the Intermediate language representation would be formatted as the one on the Source Code Snippet 4.11.

```
{
  "GraphStructureGenerator": {
    "type": "structure-generator",
    "runtime": "JVM-8",
    "language": "Scala-12",
    "s3-bucket": "https://babel.s3-eu-west-1.amazonaws.com/",
    "s3-file": "BTERGraphMovieActingNetworkGraph-generator.class"
}
```

Source Code Snippet 4.11: Example property graph based generator

Another option for generating a graph is using a sample graph, that, in the same way as with the code based generators we described on the Section 4.4.1, needs to be accessible to the public, so we need to ensure that none of the published sample graph *nodes* or *edges* are confidential. The library described on the Chapter 5 would be used for confidentializing it.

The Source Code Snippet 4.12 shows how sample graph based generators are defined on the Intermediate Language.

```
{
  "SampleGraphBasedGenerator": {
    "type": "graph",
    "format": "edge-list",
    "s3-bucket": "https://babel.s3-eu-west-1.amazonaws.com/",
    "s3-file": "example-graph.txt"
  }
}
```

Source Code Snippet 4.12: Example Sample graph based generator

4.5 Intermediate Language Generation using the DSL

As it is mentioned in Chapter 3, the *DSL* is based on the *Scala* programming language compile-time macro annotations. In this section we are providing detail on how the *DSL* generates the *IL*. These macros enrich the source code written by the user as a compilation phase, as shown in Figure 4.1, where the source code is expanded to something that will be used for generating the *IL*.

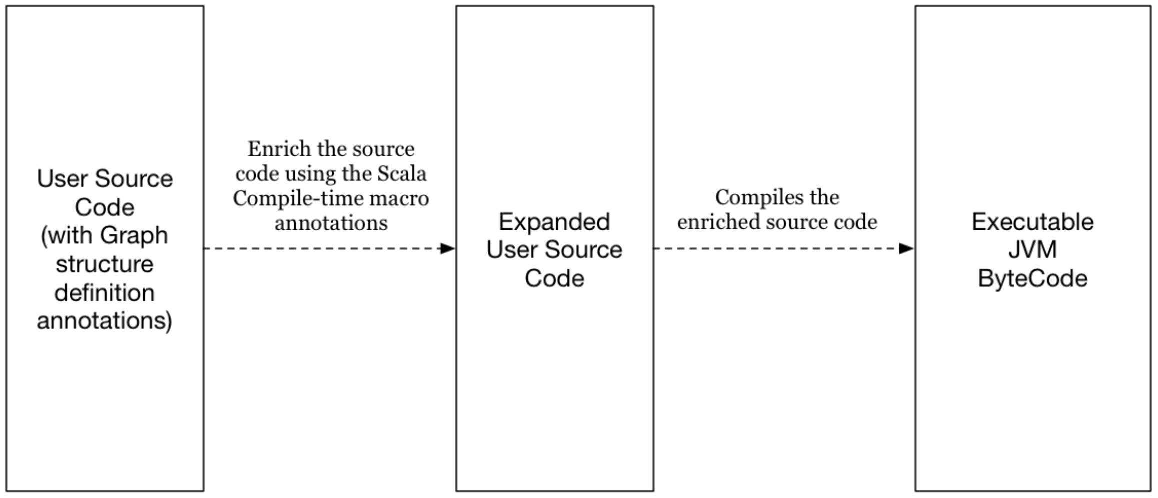

User Source Code (with Graph structure definition annotations)
Enrich the source code using the Scala Compile-time macro annotations
Expanded User Source Code
Compiles the enriched source code
Executable JVM ByteCode

Figure 4.1: Overall overview of compilation time phases when using the *DSL* compile time macro annotations, outputting an executable JVM ByteCode

4.5.1 Generating the IL from the DSL

As described in Chapter 3, for a *node* and/or an *edge* property model we only need to add `@node` in front of a *Scala* class if we want to define a *node* model, or a `@edge` if we want to define an *edge* one. For instance, if an user wants to define in the *IL* our running example `Actor` model displayed in the Source Code Snippet 3.8, it has to call the `intermediateLanguage()` method generated by the macro annotation `@node`. For doing the generation, given the initial `Actor` class definition, the macro annotation expands the code to the source code shown on the Source code Snippet 4.13. When the user calls that `intermediateLanguage()` method, the *IL* shown in the Source Code Snippet 4.14 will be generated. The output will be placed inside the parent *IL* structure defined in Section 4.1, concretely as an element of the `"graphNodeAndEdgePropertyModels" Array`. The generators, such as the one shown in the Source Code Snippet 4.10, are included as an element from the "generators" section.

```scala
case class Actor(name: String,
                 gender: Option[String],
                 country: String,
                 birthDate: java.time.LocalDate)

object Actor extends ActorCompanion(isRequired = true)
object OptionalActor extends ActorCompanion(isRequired = false)

private sealed abstract class ActorCompanion
  private (override val isRequired: Boolean)
  // If it is an Edge, extends `EdgeDefinition` instead
  extends NodeDefinition {

  override val typeName: String = "Actor"
  override val isRequired: Boolean = true

  override def intermediateLanguage(): Json = {
    import babel.types._
    import io.circe._
    import io.circe.syntax._

    val objectTypesJson =
        JsonObject.fromMap(typeMap.mapValues(_.asJson)).asJson

    val objectPropertiesJson =
        JsonObject.fromMap(
          ListMap(
             "type" -> "object".asJson,
             "properties" -> objectTypesJson
          )
        ).asJson

    JsonObject.fromMap(Map("Actor" -> objectPropertiesJson)).asJson
  }

  override def toString: String = intermediateLanguage.spaces2

  def typeMap: SortedMap[String, types.Type] = {
     import babel.types._
     ListMap[String, types.Type](
          ("name", classOf[String]),
          ("gender", Option(classOf[String])),
          ("country", String),
          ("birthDate", classOf[java.time.LocalDate])
     )
  }
}
```

Source Code Snippet 4.13: Output source code from the Source Code Snippet 3.8 after the macro annotation @node expansion compilation phase

```
{
  "Actor" : {
    "type" : "object",
    "properties" : {
      "name" : {
        "typeName" : "Text",
        "isRequired" : true,
        "constraints" : [
          {
            "name" : "MinLength",
            "value" : "0"
          },
          {
            "name" : "Encoding",
            "value" : "UTF-8"
          }
        ]
      },
      "gender" : {
        "typeName" : "Text",
        "isRequired" : false,
        "constraints" : [
          {
            "name" : "MinLength",
            "value" : "0"
          },
          {
            "name" : "MaxLength",
            "value" : "60"
          },
          {
            "name" : "Encoding",
            "value" : "UTF-8"
          }
        ]
      },
      "country" : {
        "typeName" : "Text",
        "isRequired" : true,
        "constraints" : [
          {
            "name" : "MinLength",
            "value" : "0"
          },
          {
            "name" : "Encoding",
            "value" : "UTF-8"
          }
        ]
      },
      "birthDate" : {
        "typeName" : "Date",
        "isRequired" : true,
        "constraints" : [
            {
              "name" : "MinDate",
              "value" : "1918-01-01"
            },
            {
              "name" : "MaxDate",
              "value" : "Now"
            }
          ]
        }
      }
    }
}
```

Source Code Snippet 4.14: Output Intermediate Language from the Source Code Snippet 4.13 model class serialization, containing all its types and constraints

Chapter 5

Gnormalizer: Graph normalization framework

As part of this Master thesis, an open-source library was created for reading real world graphs stored in different formats and to convert them into formats readable by a rich set of graph benchmark frameworks, such as *iGraph* [12], *GraphX* [13] or the one related to this project, *DataSynth* [20].

Pre-processing graphs in some of the most popular graph formats, such as the "*Edge List*", yield an extra complexity. Some of the benchmarks require the *edges* to be sorted by its source *node* first, and then by its target one. All of this is not as trivial as it seams, because a lot of graphs do not fit into memory, and the results are required to be sorted.

In order to optimize the performance, depending on the size of the input graph, the library chooses automatically between two different graph loading techniques, an *in-memory* and an *out-of-core* one. In addition, the framework is designed to take profit of all the computer cores, being fully parallel.

This chapter describes how this library is designed and implemented. Section 5.1 shows an overview of how the library is implemented. Section 5.3 shows the *out-of-core* algorithm, used for changing the format of arbitrary big graphs, that do not fully fit in memory. If the graphs fit in memory, the *in-memory* approach shown in Section 5.2 is used instead. Section 5.4 shows the API of the library, either if its being used as a standalone tool or as a *Babel* module. In Section 5.5 we evaluate the performance of the library using a set of benchmarks for both the *in-memory* and the *out-of-core* loading techniques.

5.1 Framework Overview

This framework, that can be used either as a library or as an standalone application, as we detail in Section 5.4, has a straightforward workflow. First, it receives an input graph, in any of the supported formats, and after its execution is completed it outputs a sorted graph in the desired format, as it is shown on the Figure 5.1.

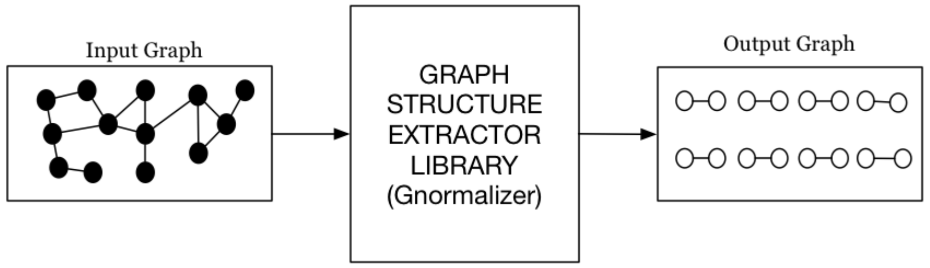


Figure 5.1: Overview of the graph structure extractor framework (Gnormalizer)

As explained in Chapter 3, *Babel* includes this tool as a core part of the API's functionality, being the main usage workflow exemplified on the Figure 5.2. On this case, *Gnormalizer* is used for converting an input graph into a format that a graph processing library, such as *iGraph* [12], can understand, so it can obtain all the graph statistics, adding them to the IL for its usage during the graph generation process. On the other hand, we can also add the converted graph as part of the IL, so the back-end may output a graph with its real structure but with synthetic *node* and *edge* parameters.

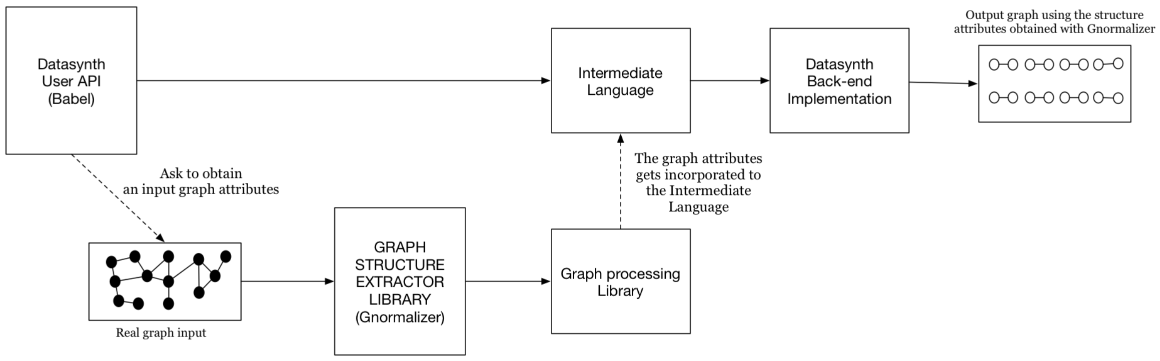


Figure 5.2: Workflow using *Gnormalizer* as a middle-ware between *Babel* and any graph processing framework for obtaining its attributes, in order to incorporate them to the Intermediate Language representation.

In the same way as *Babel*, the *Scala* [36] programming language was chosen as the implementation language. The most important reasons are its native and simple integration with a rich set of *JVM* based distributed computing frameworks, including *Spark* [14], *Hadoop* [21], and *Flink* [22]. In addition, *Gnormalizer* can be easily reused from these frameworks for reading the synthetic graphs generated using *DataSynth*. Finally, *Scala* is also faster than *Java* in the scenarios were an impure functional programming approach is used [49][50]. This fact enables creating workflows without an excessive effort similar to the one shown on the Figure 5.3, were the generated graphs are automatically evaluated using multiple benchmarking frameworks, each one of them having its own input format.

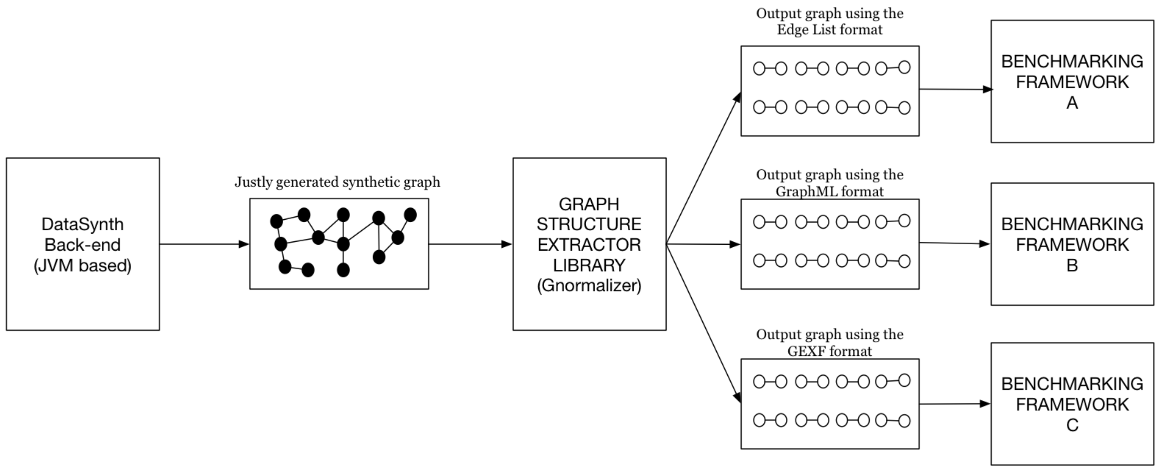


Figure 5.3: Workflow using *Gnormalizer* for translating automatically a justly generated graph into the format required for its benchmarking on multiple frameworks, taking profit on its compatibility with JVM based technologies, such as Spark [14], Flink [22] or Hadoop [21].

5.2 Extracting graphs fitting in memory

The best case scenario is that where the input graph fits entirely in memory, so we do not really need to use the hard drive for performing the graph processing, improving the algorithm performance considerably, mainly because the access to memory is orders of magnitude faster than the access to disk [52].

When the library reads a file, and, while it is processing each *node* from the input graph, it automatically asks to a mapper, which we called `NodeIndexMapper`, for the index assigned to the justly read *node*, generating a new index if it was not previously indexed. This index, that will be an integer value between `0` to `N-1`, where `N` is the number of *nodes* in the graph, corresponds to the output identifier for each graph *node*. Doing this mapping will help with the sorting afterwards, because sorting using integers is faster than with `String`s, and if we want to output in a common format, such as "*Edge List*", we don't need the *node* properties anyway.

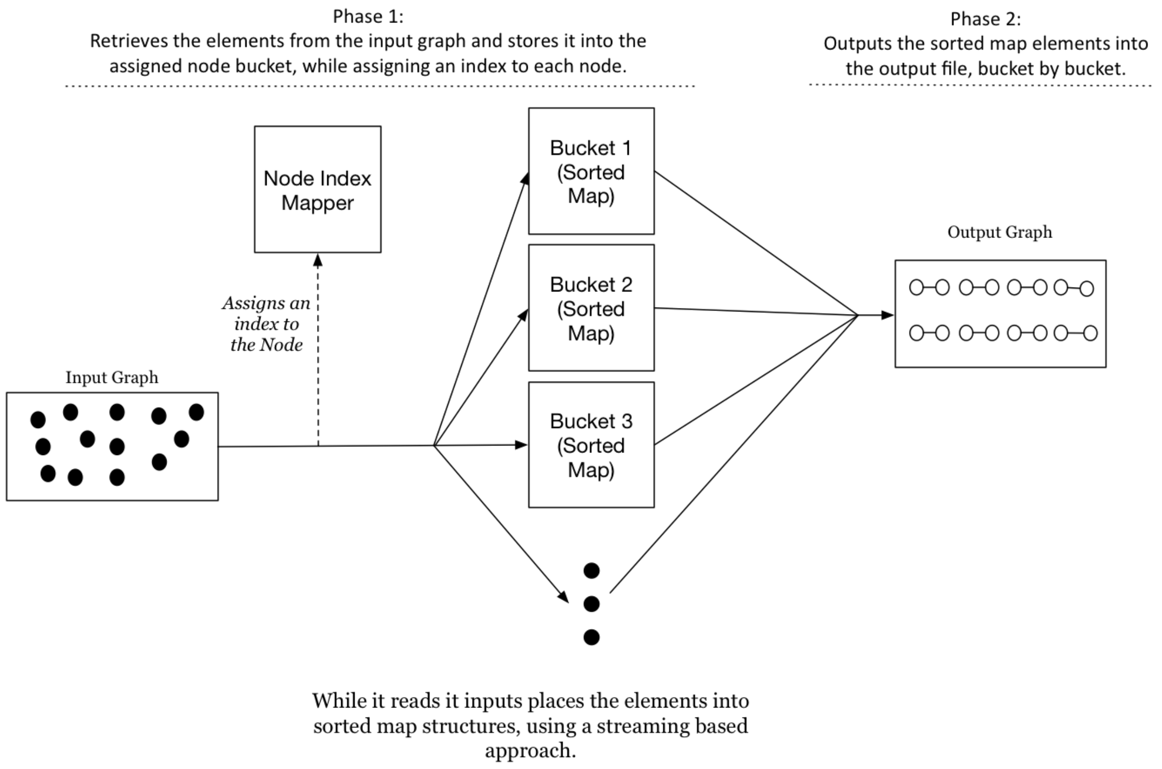

Phase 1:
Retrieves the elements from the input graph and stores it into the assigned node bucket, while assigning an index to each node.
Phase 2:
Outputs the sorted map elements into the output file, bucket by bucket.
Node Index Mapper
Assigns an index to the Node
Input Graph
Bucket 1 (Sorted Map)
Bucket 2 (Sorted Map)
Bucket 3 (Sorted Map)
Output Graph
While it reads it inputs places the elements into sorted map structures, using a streaming based approach.

Figure 5.4: In-memory graph structure extraction approach overview

Every time an *edge* is read from the input graph and the index of its endpoints are assigned using the *Node Index Mapper*, it is inserted in a `TreeMap` structure, using the *edge* source as the key, and a `TreeList` for all the target *nodes* associated with the source. Doing this, the parsed *edges* are already stored in memory sorted, having in both cases a computational complexity of *log(n)* [51].

Unfortunately, using this approach alone would not be scalable because the insertion complexity is $log(n)$, and if we store too many *edges* in the structure we are going to start to be penalized by the increasing logarithmic complexities.

For solving this issue, the library uses *edge buckets*, choosing between with a helper `HashMap`, reducing the tree searches to the minimum when running the library algorithm. Each of the *buckets* is responsible of handling all the *edges* whose source belongs to one of its *nodes*, being it possible to configure the number of owned *nodes*. If the graph is not directed, we are currently treating the first endpoint we see for doing the *bucket* assignation, that usually is done in order by the index.

As it is explained on Section 5.5.2, the default number of *nodes* per *bucket* is semi-arbitrary set to 4,500 *nodes*. Changing this number may improve the performance for some graphs and decrease it in others, as it will be shown on Section 5.5.

Once all the graph *edges* are placed into this graph structures, all of their *edges* are serialized to an output file, in the provided graph format.

The whole workflow is summarized in Figure 5.4, showing an overview on how the *in-memory* sorting algorithm works. If the graph is extremely small, (5Mb at most) a simple Quick-sort algorithm is used, because on the Section 5.2 benchmarks we saw a performance increase of about 15% for those scenarios.

5.3 Extracting graphs not fitting in memory

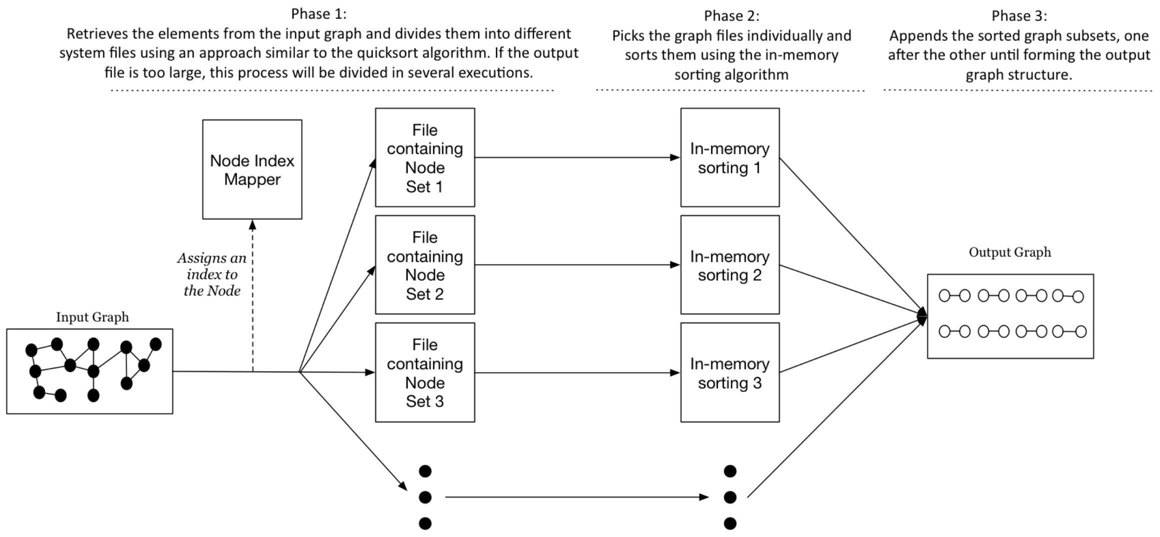

Phase 1:
Retrieves the elements from the input graph and divides them into different system files using an approach similar to the quicksort algorithm. If the output file is too large, this process will be divided in several executions.
Phase 2:
Picks the graph files individually and sorts them using the in-memory sorting algorithm
Phase 3:
Appends the sorted graph subsets, one after the other until forming the output graph structure.
Node Index Mapper
Assigns an index to the Node
Input Graph
File containing Node Set 1
File containing Node Set 2
File containing Node Set 3
In-memory sorting 1
In-memory sorting 2
In-memory sorting 3
Output Graph

Figure 5.5: Out-of-core graph structure extraction approach

When the graph does not fit entirely in memory, we need to use an alternative approach to the one defined on Section 5.2. Compared to the *in-memory* version, it splits the graph file contents into different separate files stored in the Hard Disk Drive, so the memory does not need to store simultaneously all the file content. This implementation is based on the merge-sort algorithm [53].

This approach works in a similar way to the *in-memory* one, but it temporally places, if required, the graph in separate files. When all the graph is extracted from the input file and split between files, each of the generated files run individually the *in-memory* algorithm in order, appending the output graph to the same file.

During all this process, the *Node Index Mapper* is kept entirely in memory, but only stores the graph *node* unique identifiers and its assigned indexes, so the amount of memory used is a small fraction in comparison to the total graph size. This is possible because typically in real graphs the number of *edges* is much larger than the number of *nodes*.

An overview on how this algorithm works, supposing that the optimizer chooses not to bypass any intermediate storing to disk, is displayed on the Figure 5.5.

5.4 Framework API

For using *Gnormalizer* we can either use it as a library or as an application. If we use it as a library we only need to add it as a dependency, and then use its API directly. This API is really simple, we only require to specify the input and output file locations and call the `execute()` command. In addition, if we want to override the *bucket* size, because as it is shown in Section 5.5 the bucket size affects performance, and it may affect the performance differently depending of the type of the graph. A full usage example of the API is shown in the Source Code Snippet 5.1, including an overload of the used *bucket size* used by the inner algorithms to 3000. In the Source Code Snippet 5.2 we show the second option, that is calling the *Gnormalizer* application directly from the *shell* with the parameters `"--input"` (file), `"--if"` (input format), `"--output"` (file) and `"--of"` (output format) specified. Optionally, we can override the *bucket* size with the parameter `"--bucketSize"`.

```
Gnormalizer
  .inputFile("./input/connections.graph", GraphFormat.EdgeList)
  .outputFile("./output/connections.graph", GraphFormat.EdgeList)
  .execute(bucketSize = 3000)
```

Source Code Snippet 5.1: Example usage of the *Gnormalizer* as a library

```
$ gnormalizer \
     --input ./input/connections.graph -if EdgeList \
     --output ./output/connections.graph -of EdgeList \
     --bucketSize 3000
```

Source Code Snippet 5.2: Example execution of Gnormalizer from the *Shell*

5.5 Experiments

A set of already existing graphs found on the Standford Graph Database (SNAP) [54] were chosen as the project framework training graphs, each of them with different sizes and structures.

For doing the analysis we are executing a test suite against multiple different *bucket* sizes, until the system crashes because an `Exception` is thrown, an execution takes 10 times more than the previous, or we reach the 100.000 *bucket* size. We execute the test 4 times, the first one for warming up the *JVM* and the other three for checking the performance. The results displayed across this section represents the average of the 3 later executions.

With these benchmarks we evaluate the algorithm performance against multiple *bucket* sizes, and determine the default *bucket* sizes values with them.

Table 5.1 shows he characteristics of the machine we used to run the experiments, and in Table 5.2 we detail the software versions used for running the benchmarks.

Processor	Intel I7-4960HQ - 4 cores @ 2,6 GHz
L1 cache	256 KB
L2 cache	1 MB
L3 cache	6 MB
Memory	8Gb 1600 MHz DDR3
Hard Disk	Crucial MX300 525GB 3D NAND SATA M.2 (2280) Internal SSD - CT525MX300SSD4

Table 5.1: Technical characteristics of the computer used for running the benchmarks

Mac OS X	Sierra - 10.12.6
Java Virtual Machine	JDK 8 - 1.8.0_181
Scala	2.12.7
Gnormalizer library (Benchmarked library)	0.4
FS2 - Functional Streams 2 library	1.0.0
Cats-Effect (IO monad library)	1.0.0
Better Files (File handling library)	3.6.0

Table 5.2: Software and library versions used on the benchmark execution

5.5.1 In-memory algorithm Benchmark results

In this section, we are benchmarking the *in-memory* algorithm using the real-life graphs shown in Table 5.3. In Section 5.5.1.1 we are running the *in-memory* algorithm on the 'EU email communication network' graph, in Section 5.5.1.2 we are doing the same for the 'Social Circles: Twitter' graph and the 'Patent Citation Network' is been benchmarked in Section 5.5.1.3. Finally, in Section 5.5.1.4 we compare the results of these benchmarks with another ones using the Quick-sort [61] algorithm instead.

	EU Email Communication Network [55]	Social Circles: Twitter [56]	Patent Citation Network [57]
Size	1.5Mb	44.6Mb	280.5Mb
Number *nodes*	265,214	81,306	3,774,768
Number *edges*	420,045	1,768,149	16,518,948

Table 5.3: Small graphs used for benchmarking the in-memory algorithm

5.5.1.1 Benchmark results for the 'EU email communication network' graph

For the 'EU email communication network' graph the best processing time was when each bucket owned 83,750 *nodes*, doing the whole processing on just 1049.21 milliseconds, processing 400,344.06 *nodes* per second in average. The Figure 5.6 shows the results of the benchmarks when running the *in-memory* algorithm on the "EU email communication" network graph depending on the bucket size.

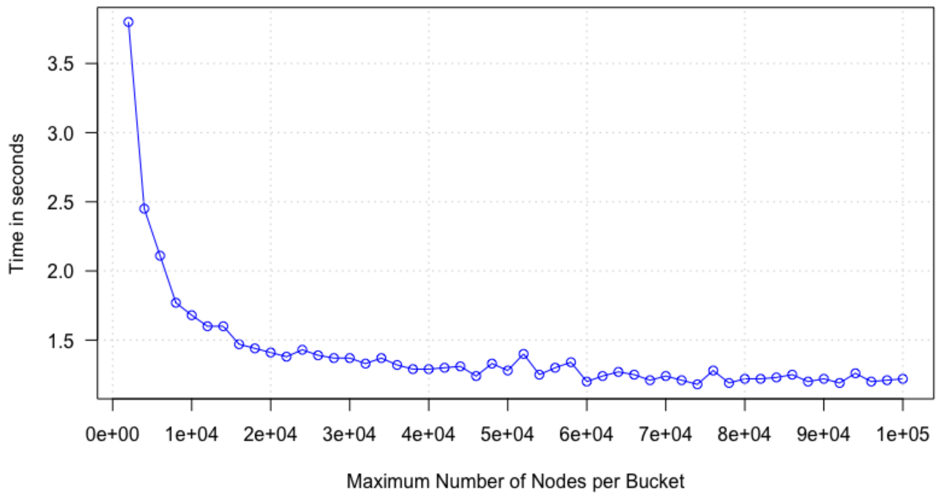


Figure 5.6: Average execution time to read and sort the 1.5Mb 'EU email communication' graph (265,214 *nodes* and 420,045 *edges*) depending on the *bucket* size.

5.5.1.2 Benchmark results for the 'Social Circles: Twitter' graph

The library spent a minimum of 5.62 seconds for processing the 'Social Circles: Twitter' graph when the maximum bucket size was of 29,500 *nodes*, handling an average of 314,617.26 *edges* per second. The Figure 5.7 shows the results of the benchmarks when running the *in-memory* algorithm on the "Social Circles: Twitter" network graph depending on the bucket size. It is worth of notice that when the *bucket* size is greater than 40,000 *nodes*, the processing time gets worse significantly fast, and in Section 5.5.1.5 we are going to enumerate our theories about why this phenomenon happens.

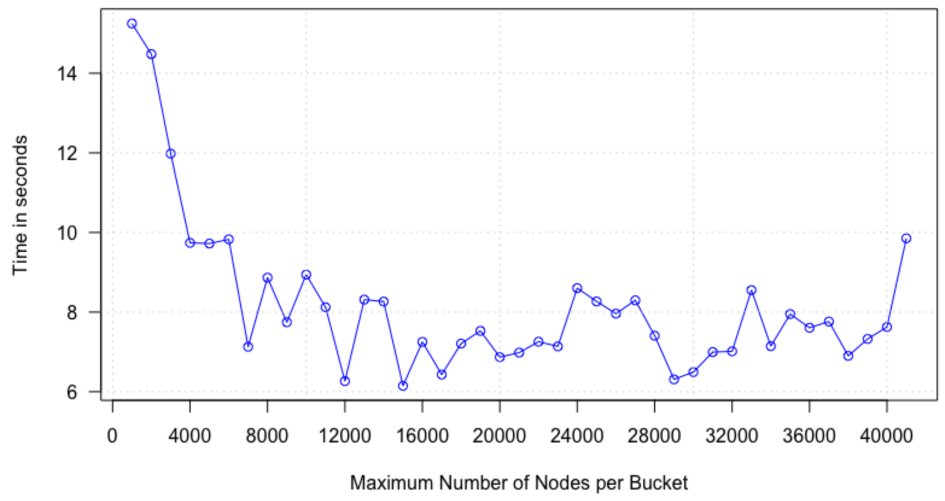


Figure 5.7: Average execution time to read and sort the 44.6Mb 'Social Circles: Twitter' graph (81,306 *nodes* and 1,768,149 *edges*) depending on the *bucket* size.

5.5.1.3 Benchmark results for the 'Patent citation' graph

The minimum processing time for the patent citation graph processing was obtained when the bucket size consisted of a maximum of 21,000 *nodes*, doing the whole execution on 24.33 seconds, handling 678,860.9 *edges* per second. The performance also started to suffer an important degradation in a similar point point as with the 'Social Circles: Twitter' graph, but the degradation started when the maximum *bucket* size was of 42,000 *nodes*. Figure 5.8 shows the results of the benchmarks when running the *in-memory* algorithm on the 'Patent citation' graph depending on the bucket size.

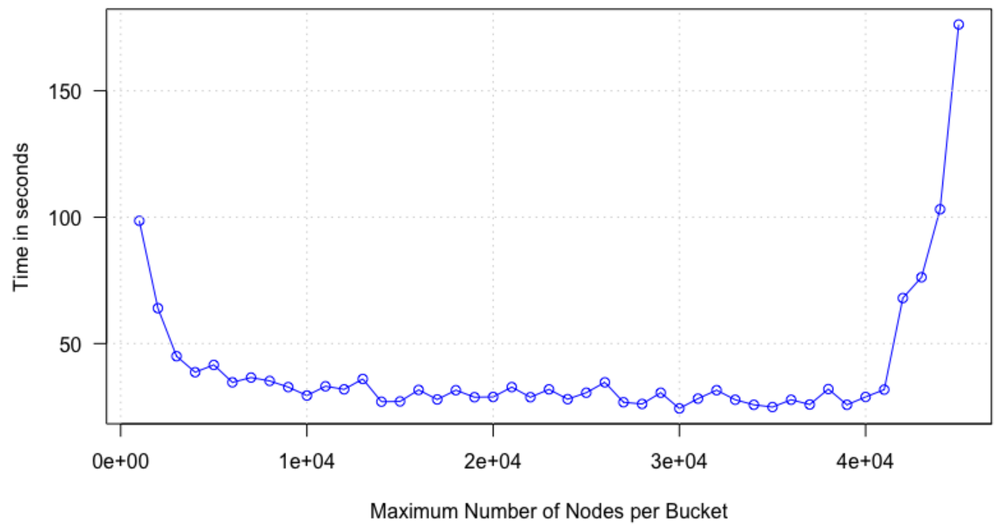


Figure 5.8: Average execution time to read and sort the 280.5Mb 'Patent citation network graph' graph (3,774,768 *nodes* and 16,518,948 *edges*) depending on the *bucket* size.

5.5.1.4 Comparing the results with the ones obtained when using the Quick-Sort algorithm directly on the graphs

Since the graphs used in this section are relatively small, we tried running the Quick-Sort algorithm directly on them. For the 'EU email communication network' we were capable of doing the processing on just 759.75 milliseconds, about a 27.6% faster. On the other hand, we got a `StackOverflowException` when trying to sort the other graphs using this algorithm, so this algorithm has a blocking limitation for graphs exceeding a few *MegaBytes*, because it heavily relays on the *Java Stack*. If we increase the *Java Stack* size overriding the parameters in the *JVM*, we start having consistently much worse results once the size of the graph exceeded the size of the computer *cache*.

5.5.1.5 Performance degradation when the bucket size is too big

Both the *in-memory* and the *out-of-core* algorithms experienced in our benchmarks a significant performance degradation once the *bucket* size reaches a certain point, that varies depending on the graph, but it appears to be stabilized at around 8,000 *nodes* in big graphs. This case can be observed in our test graph benchmarks from Figure 5.7, Figure 5.8, Figure 5.9, Figure 5.10 and Figure 5.11.

We have several theories about the root cause, but we think it is highly probable that this is caused because of the *cache* size. If the *bucket* size is too big we may arrive to a point were the buckets do not fit into it, so the algorithm will run in a smaller performance *cache* level, or even in the *RAM* memory, which is significantly slower. This assumptions gains weight because *Gnormalizer* is highly parallel and it divides the whole graph between all the available threads, and, since the real-world graph are usually sorted, every thread will have a subsection of a 'sorted' graph that will use continuously the same *buckets* until the source *node* changes. Since the *bucket* size is too big, not all of the threads *buckets* fit into the *cache*, making the graph normalization significantly slower.

5.5.2 Out-of-core algorithm Benchmark results

In this section we are detailing the results of the benchmarks done using the *out-of-core* algorithm. For doing this, we used the graphs shown in Table 5.4, and we executed a suite of test with different *bucket* sizes, with 250 *node* increments, until a `JavaHeapException` stopped the benchmark execution. Section 5.5.2.1 shows the results of the *out-of-core* algorithm when using the 'Google+' graph, the Section 5.5.2.2 does the same for the 'Amazon reviews' one and the results from the 'Friendster social network and ground-truth communities' graph are shown in Section 5.5.2.3. Finally, the Section 5.5.2.4 explains how we selected the default *bucket size* parameters for the library using the results from these benchmarks.

	Google+ [58]	Amazon Reviews [59]	Friendster social network and ground-truth communities [60]
Size	1.34Gb	11.02Gb	32.36Gb
Number *nodes*	107,614	9,084,722	65,608,366
Number *edges*	13,673,453	34,686,770	1,806,067,135

Table 5.4: Big graphs used for benchmarking the out-of-core algorithm

5.5.2.1 Benchmark results for the 'Google+' graph

On the the 'Google+' benchmark, the best result was obtained when the *bucket* size consisted of a maximum of 4,250 *nodes*, in this case the total processing time was of 121.244 seconds, processing in average 112,776.33 *edges* per second. The Figure 5.9 shows the performance distribution depending on the selected *bucket size*.

In spite this graph fits entirely into memory, we benchmarked it using the *out-of-core* algorithm for checking if graph size affected the performance significantly, due to the shared *Node Index Mapper*. The experiments shown that doing the conversion of the graph 'in-memory' was only a 26.69% faster, if we consider the number of processed *edges* per second.

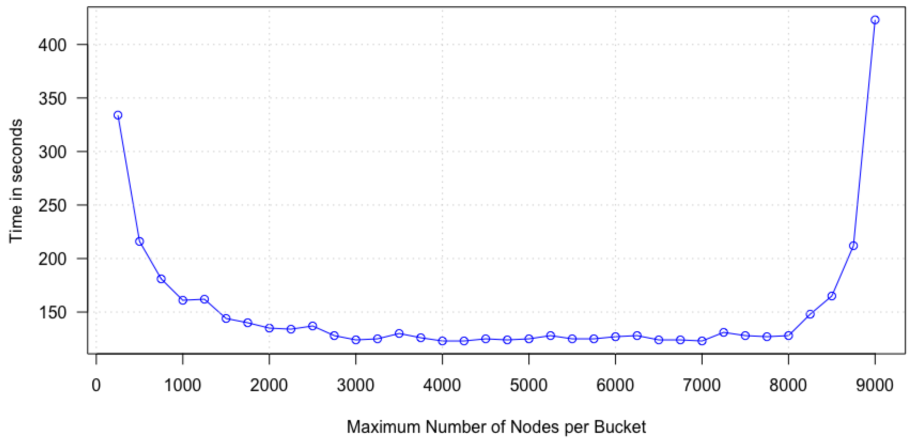


Figure 5.9: Average execution time to read and sort the 1.34Gb Google+ graph (107,614 *nodes* and 13,673,453 *edges*) depending on the *bucket* size.

5.5.2.2 Benchmark results for the 'Amazon reviews' graph

On the the 'Amazon reviews' benchmark, the best result was obtained when the *bucket* size consisted of a maximum of 3,000 Nodes, in this case the total processing time was of 1,164.94 seconds, processing in average 29,775.58 *edges* per second.

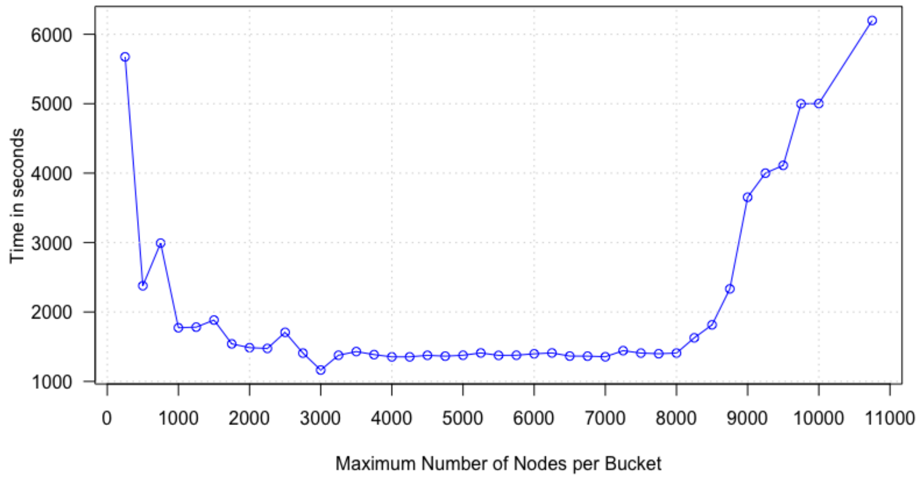


Figure 5.10: Average execution time to read and sort the 11.02Gb Amazon Reviews graph (9,084,722 *nodes* and 34,686,770 *edges*) depending on the *bucket* size.

5.5.2.3 Benchmark results for the 'Friendster social network and ground-truth communities' graph

For the 'Friendster social network and ground-truth communities' graph, the best results were obtained when having a *bucket* size of 4,750 *nodes*, spending 20,236.8 seconds for the whole conversion, processing an average of 89,246.68 *edges* per second.

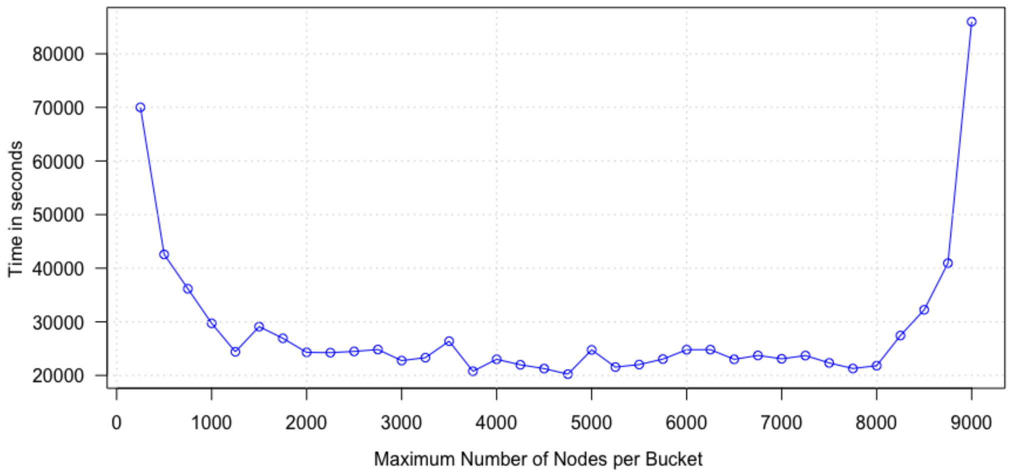


Figure 5.11: Average execution time to read and sort the 32.36Gb Friendster graph (65,608,366 *nodes* and 1,806,067,135 *edges*) depending on the *bucket* size.

5.5.2.4 Picking a default bucket size using the results

In all the *out-of-core* benchmarks the performance curve were really similar when running the benchmarks across different *bucket* sizes, showing in all the cases that an optimal performance was obtained when choosing a *bucket* size between 3,000 and 8,000 nodes. In Section 5.5.1.5 we detail our theories about the root cause of the performance degradation happening once the *bucket* size exceeds the 8,000 *nodes*.

On the other hand, If we consider the average number of processed *edges* per second as the benchmark target, we can see that the performance difference between the graphs is evident. We have a hint that this is caused by the graph structure itself and not because of its size, because the processing performance of the 32.36Gb graph was comparatively three times faster than the 11.02Gb one.

Considering that the performance distribution between *bucket* sizes was considerably consistent in all the tested cases, and the problem of selecting a wrongly sized *bucket size*, we defaulted the *bucket* size for *Gnormalizer* semi-arbitrarily to a maximum of 4.500 *nodes*, but we let, as shown in Section 5.4, overload the bucket size depending on the type of graph we have.

Chapter 6

Conclusions

The work in this master thesis has the main objective to design and implement a comprehensive and flexible system for generating synthetic graphs. This was accomplished by providing an implementation independent Domain Specific Language (*DSL*). This metalanguage, when compiled, generates an Intermediate Language (*IL*) representation that can be executed by any supported generator, in our case *DataSynth*, outputting a synthetic graph with the desired *nodes* and *edges* properties and cardinalities.

In order to favor the platform adoption and to enable implementations without using the *DSL*, the *IL* is design to be as human readable as possible. This metalanguage implements, as part of its core functionality, the main character and numeric primitive types defined in most modern programming languages, adding to them more complex types like date, time, duration and scheduling ones, just to cite a few of them.

Both the *DSL* and the *IL* are flexible enough to represent many kinds of graphs generation plans we may think on, including, if it is required, custom code in our desired programming language. This code would be serialized and included as part of the *IL*, so it can be used during the graph generation.

As a secondary thesis output, a library named *Gnormalizer* was designed and implemented for parsing and converting real and/or synthetic graphs to different graphs formats. This tool allows converting real or synthetic graphs to the formats of third-party graph processing frameworks and benchmarks accept.

Some benchmarking frameworks require graph formats where the input graph is sorted. Additionally, real world or synthetic graphs may be of arbitrary big sizes, so the library is capable of handling graphs that do not fit in memory. This added the requirement to the library of being capable of sorting in an out-of-core scenario.

Using the library as a research sandbox, a parallel streaming out-of-core algorithm was developed for performing the graph conversions, yielding promising results such as been able to perform the format conversion, including the sorting of a 1.34Gb graph in 122.84 seconds, a 11.02Gb graph in 1,164.94 seconds and a 32.36Gb one in 20,236.80 seconds.

6.1 Future work

During this thesis the foundations of a generic graph generation system have been set, but it still needs to be integrated with *DataSynth* and potentially with another third-party graph processing frameworks. Once this integration is done, we would automatically evaluate the graphs as soon as they are generated, using any third-party graph processing framework or benchmark.

Another research line is related with *Gnormalizer* and its out-of-core streaming fully parallel approach when sorting the input graphs. The investigation done in this thesis about this topic was really superficial, due to time and scope constraints, but its results were promising enough for consider doing more research on this topic. Further investigation may provide a better approach for sorting files that do not fit entirely into the memory, when traditional sorting algorithm do not take profit of all the resources provided by the computer.

Also in the *DSL*, we may start a research line on automating the creation of *node* or *edge* property generators. These generators, would be automatically generated from the type *constraints* specified by means of the *DSL*.

In order to increase the utility of the *DSL*, it would also be really interesting to convert it to a synthetic graph generator orchestrator, being capable of handling generation plans on distributing computing environments, using some SAAS on-line platform, such as *Azure* [62] or *AWS* [63].

Finally, on the longer run, we would be able to improve significantly the feasibility of synthetic graphs, thanks to the usage of recursive graph generation pipelines for training the generation algorithms using Artificial Intelligence [64][65] and Deep Neural Network [66] approaches.

References

[1] OTTE, Evelien; ROUSSEAU, Ronald. Social network analysis: a powerful strategy, also for the information sciences. Journal of information Science, 2002, vol. 28, no 6, p. 441-453.

[2] NOBLE, Caleb C.; COOK, Diane J. Graph-based anomaly detection. En Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003. p. 631-636.

[3] DEBNATH, Souvik; GANGULY, Niloy; MITRA, Pabitra. Feature weighting in content based recommendation system using social network analysis. En Proceedings of the 17th international conference on World Wide Web. ACM, 2008. p. 1041-1042.

[4] BARABÁSI, Albert-László; GULBAHCE, Natali; LOSCALZO, Joseph. Network medicine: a network-based approach to human disease. Nature reviews genetics, 2011, vol. 12, no 1, p. 56.

[5] BARABASI, Albert-Laszlo; OLTVAI, Zoltan N. Network biology: understanding the cell's functional organization. Nature reviews genetics, 2004, vol. 5, no 2, p. 101.

[6] OTTE, Evelien; ROUSSEAU, Ronald. Social network analysis: a powerful strategy, also for the information sciences. Journal of information Science, 2002, vol. 28, no 6, p. 441-453.

[7] Neo4J graph database. (n,d.) Retrieved June 6, 2018, from: `https://neo4j.com/`

[8] AWS Neptune graph database. (n,d.) Retrieved June 6, 2018, from: `https://aws.amazon.com/neptune/`

[9] Sparksee graph database. (n,d.) Retrieved June 6, 2018, from: http://www.sparsity-technologies.com/

[10] Dgraph graph database. (n,d.) Retrieved June 6, 2018, from: https://dgraph.io/

[11] MALEWICZ, Grzegorz, et al. Pregel: a system for large-scale graph processing. En Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010. p. 135-146.

[12] iGraph - Network Analysis Software. (n, d.) Retrieved June 6, 2018, from: http://igraph.org/

[13] GraphX, Apache Spark API for graphs and graph-parallel computation. (n, d.) Retrieved June 6, 2018, from: https://spark.apache.org/graphx/

[14] Apache Spark, Unified Analytics Engine for Big Data. (n, d.) Retrieved June 6, 2018, from: https://spark.apache.org/

[15] LDBC Council: Social Network Benchmark. (n,d.) Retrieved June 6, 2018, from: http://ldbcouncil.org/developer/snb

[16] LDBC Council: Graphalytics Open-source Graph Processing Benchmark Suite. (n,d.) Retrieved June 6, 2018, from: https://graphalytics.org/

[17] G. Bagan, A. Bonifati, R. Ciucanu, G. H. . Fletcher, A. Lemay and N. Advokaat, "gMark: Schema-Driven Generation of Graphs and Queries," in IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 4, pp. 856-869, April 1 2017.

[18] Graph 500, Large-scale graph benchmarks. (n,d.) Retrieved June 6, 2018, from: http://graph500.org/

[19] GUO, Yuanbo; PAN, Zhengxiang; HEFLIN, Jeff. LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web, 2005, vol. 3, no 2-3, p. 158-182.

[20] Prat-Pérez, Arnau & Guisado-Gámez, Joan & Fernández Salas, Xavier & Koupý, Petr & Depner, Siegfried & Basilio Bartolini, Davide. (2017). Towards a property graph generator for benchmarking. 1-6. 10.1145/3078447.3078453.

[21] Apache Hadoop distributed computing framework. (n,d.) Retrieved June 6, 2018, from: http://hadoop.apache.org/

[22] Apache Flink: Scalable Streaming Framework. (n,d.) Retrieved June 6, 2018, from: https://flink.apache.org/

[23] RON, Dorit; SHAMIR, Adi. Quantitative analysis of the full bitcoin transaction graph. En International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2013. p. 6-24.

[24] Java JDK specification. (n, d.) Retrieved August 25, 2017, from: https://docs.oracle.com/javase/specs/jvms/se8/html/

[25] LATTNER, Chris; ADVE, Vikram. LLVM: A compilation framework for lifelong program analysis & transformation. En Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004. p. 75.

[26] ECMA-404 The JSON Data Interchange Standard. (n,d.) Retrieved May 27, 2018, from: https://www.json.org/

[27] Variables and types on C++. (n,d.) Retrieved June 17, 2018, from: http://www.cplusplus.com/doc/tutorial/variables/

[28] Primitive Data Types on Java. (n,d.) Retrieved June 17, 2018, from: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

[29] Go programming language data types. (n,d.) Retrieved June 17, 2018, from: https://www.tutorialspoint.com/go/go_data_types.htm

[30] Ruby programming language. (n,d.) Retrieved June 17, 2018, from: https://www.ruby-lang.org/es/

[31] Numeric type on Ruby. (n,d.) Retrieved June 17, 2018, from: https://ruby-doc.org/core-2.2.0/Numeric.html

[32] LISKOV, Barbara; ZILLES, Stephen. Programming with abstract data types. En ACM Sigplan Notices. ACM, 1974. p. 50-59.

[33] Universally Unique Identifiers (UUIDs). (n,d.) Retrieved September 2, 2017, from: https://www.itu.int/en/ITU-T/asn1/Pages/UUID/uuids.aspx

[34] Data elements and interchange formats – Information interchange – Representation of dates and times (n,d.) Retrieved Sep from: https://www.iso.org/standard/40874.html

[35] PIERCE, Benjamin C.; BENJAMIN, C. Types and programming languages. MIT press, 2002.

[36] The Scala Programming Language. (n,d.) Retrieved May 14, 2017, from: https://www.scala-lang.org/

[37] Scala Macro Annotations. (n,d.) Retrieved January 24, 2018, from: https://docs.scala-lang.org/overviews/macros/annotations.html

[38] GNU Emacs, An extensible, customizable, free/libre text editor — and more. (n,d.) Retrieved 24 August, 2018, from: https://www.gnu.org/software/emacs/

[39] Visual Studio Code code edition tool. (n,d.) Retrieved 24 August, 2018, from: https://code.visualstudio.com/

[40] Eclipse - Integrated Development Environment. (n,d.) Retrieved 24 August, 2018, from: https://www.eclipse.org/ide/

[41] IntelliJ IDEA - Integrated Development Environment. (n,d.) Retrived 24 August, 2018, from: https://www.jetbrains.com/idea/

[42] SLOANE, Tony. Experiences with domain-specific language embedding in Scala. En Domain-Specific Program Development. 2008. p. 7.

[43] DEVITO, Zachary, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. En Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011. p. 9.

[44] HOFER, Christian; OSTERMANN, Klaus. Modular domain-specific language components in scala. En ACM SIGPLAN Notices. ACM, 2010. p. 83-92.

[45] Akka, free and open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. (n,d.) Retrieved August 24, 2018, from: https://akka.io/

[46] Literal-based singleton types in *Scala*. (George Leontiev, Eugene Burmako, Jason Zaugg, Adriaan Moors, Paul Phillips, Oron Port, Miles Sabin and Adriaan Moors) Retrieved October 8, 2018, from: `https://docs.scala-lang.org/sips/42.type.html`

[47] Implicit conversion in *Scala*. (Heather Milles, Martijn Hoekstra) Retrieved October 10, 2018, from: `https://docs.scala-lang.org/tour/implicit-conversions.html`

[48] CHAKRABARTI, Deepayan; ZHAN, Yiping; FALOUTSOS, Christos. R-MAT: A recursive model for graph mining. En Proceedings of the 2004 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2004. p. 442-446.

[49] PANKRATIUS, Victor; SCHMIDT, Felix; GARRETÓN, Gilda. Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. En Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 2012. p. 123-133.

[50] PANKRATIUS, Victor; SCHMIDT, Felix; GARRETÓN, Gilda. Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. En Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 2012. p. 123-133.

[51] MUNSON, John C.; KOHSHGOFTAAR, Taghi M. Measurement of data structure complexity. Journal of Systems and Software, 1993, vol. 20, no 3, p. 217-225.

[52] JACOBS, Adam. The pathologies of big data. Communications of the ACM, 2009, vol. 52, no 8, p. 36-44.

[53] COLE, Richard. Parallel merge sort. SIAM Journal on Computing, 1988, vol. 17, no 4, p. 770-785.

[54] SNAP, Stanford Network Analysis Project (n,d.) Retrieved December 4, 2016, from: `https://snap.stanford.edu/`

[55] SNAP, Stanford Network Analysis Project: Graph Database repository: EU email communication network. (n,d.) Retrieved December 4, 2016, from: `https://snap.stanford.edu/data/email-EuAll.html`

[56] SNAP, Stanford Network Analysis Project: Graph Database repository: Social circles: Twitter. (n,d.) Retrieved October 6, 2018, from: https://snap.stanford.edu/data/ego-Twitter.html

[57] SNAP, Stanford Network Analysis Project: Graph Database repository: Patent citation network. (n,d.) Retrieved December 4, 2016, from: https://snap.stanford.edu/data/cit-Patents.html

[58] SNAP, Stanford Network Analysis Project: Graph Database repository: Egonets Google+. (n,d.) Retrieved December 4, 2016, from: https://snap.stanford.edu/data/egonets-Gplus.html

[59] SNAP, Stanford Network Analysis Project: Graph Database repository: Amazon Reviews. (n,d.) Retrieved July 16, 2018, from: https://snap.stanford.edu/data/web-Amazon.html

[60] SNAP, Stanford Network Analysis Project: Graph Database repository: Friendster social network and ground-truth communities. (n,d.) Retrieved December 4, 2016, from: https://snap.stanford.edu/data/com-Friendster.html

[61] HOARE, Charles AR. Quicksort. The Computer Journal, 1962, vol. 5, no 1, p. 10-16.

[62] Microsoft Azure Cloud Computing Platform. (n,d.) Retrieved October 10, 2018, from: https://azure.microsoft.com/

[63] Amazon Web Services (AWS). (n,d.) Retrieved October 10, 2018, from: https://aws.amazon.com/

[64] NILSSON, Nils J. Principles of artificial intelligence. Morgan Kaufmann, 2014.

[65] RUSSELL, Stuart J.; NORVIG, Peter. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.

[66] NIELSEN, Michael A. Neural networks and deep learning. USA: Determination press, 2015.