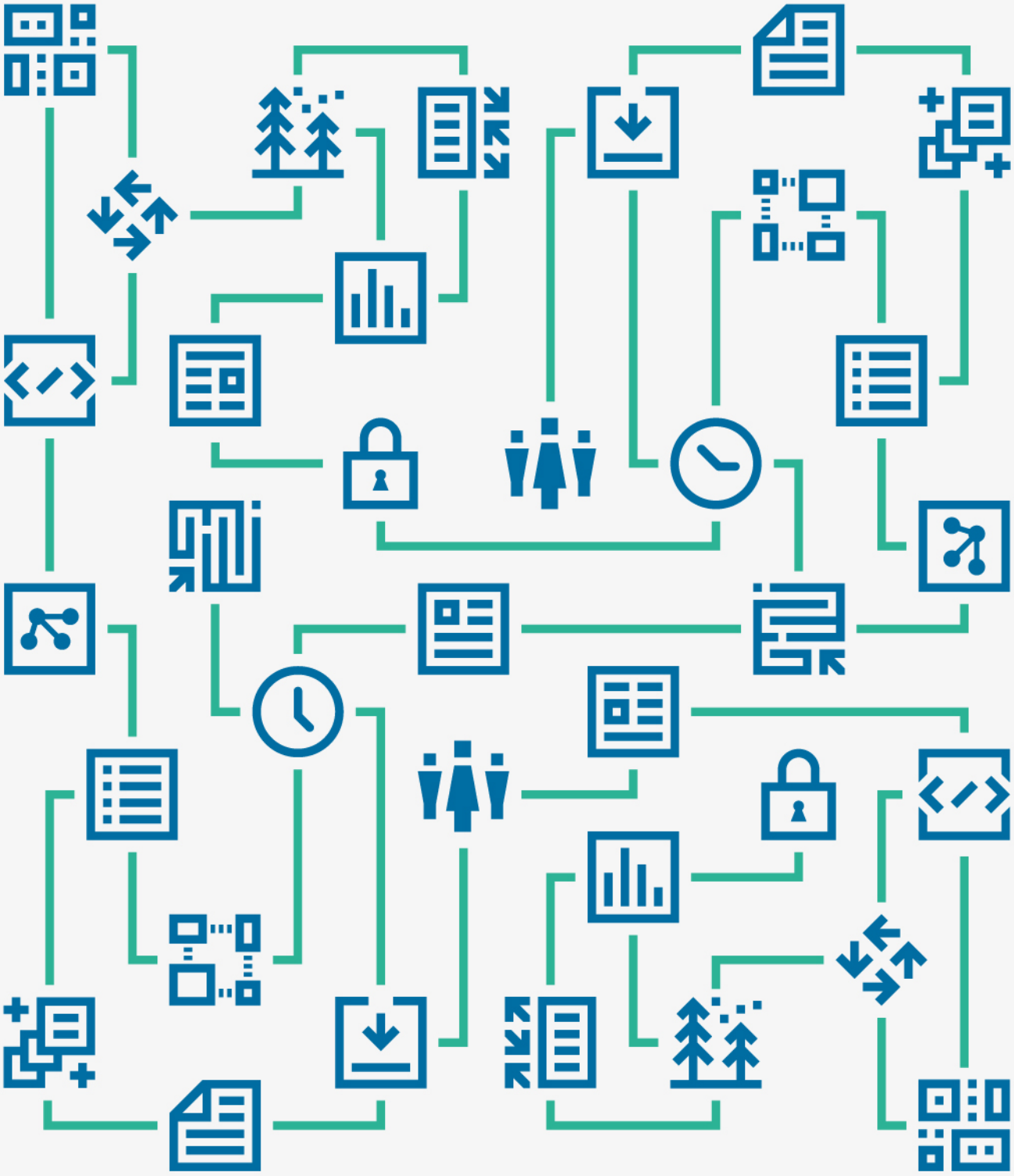
User Manual

Sparksee by Sparsity Technologies



Contents

Introduction **7**

About this document . . . 7

Notation . . . 8

License . . . 9

Support . . . 10

Documentation section . . . 10

Support section . . . 10

Graph database **11**

General concepts . . . 11

Sparksee graph database . . . 11

Graph data model . . . 11

Types . . . 12

Node and edges . . . 13

Attributes . . . 13

Indexing . . . 14

Processing . . . 15

Installation **19**

System requirements . . . 19

Download . . . 19

Java . . . 20

With Maven . . . 20

With Android . . . 21

.NET . . . 21

MS Visual Studio users . . . 22

Command-line users . . . 24

C++ . . . 24

Windows users . . . 25

Linux/MacOS users . . . 27

Android . . . 28

iOS . . . 28

Python . . . 29

Objective-C . . . 30
MacOS . . . 30
iOS . . . 31

API **33**

Database construction . . . 33
Cache configuration . . . 36
Nodes and edges . . . 37
Attributes and values . . . 43
Definition . . . 43
Use . . . 45
Objects . . . 51
Objects Iterator . . . 52
Combination . . . 54
Query operations . . . 56
Regular expressions . . . 59
Navigation operations . . . 61
Neighbor index . . . 65
Transactions . . . 65
Import/export data . . . 69
Visual export . . . 69
Data import . . . 79
Data export . . . 84
Scripting . . . 90
ScriptParser class . . . 90
Interactive execution . . . 90
Algorithms . . . 91
Traversal . . . 92
Context . . . 94
Shortest path . . . 95
Connectivity . . . 98
Community detection . . . 100

Configuration **105**
Set-up . . . 105
SparkseeConfig class . . . 105
Properties file . . . 106
Variables . . . 107
License . . . 108
Log . . . 108
Rollback . . . 108
Recovery . . . 108
Storage . . . 109
Cache . . . 110
SparkseeHA . . . 111
Basic configuration . . . 112

Scripting **113**
Schema definition . . . 113
Create and open . . . 113
Types and attributes . . . 114
Load . . . 116
Load nodes . . . 117
Load edges . . . 119
Other . . . 120

High availability **123**
Architecture . . . 123
Design . . . 123
How it works . . . 125
Future work . . . 126
Configuration . . . 127
Installation . . . 127
ZooKeeper . . . 127
SparkseeHA . . . 128
Example . . . 129
HAProxy . . . 129
ZooKeeper . . . 130
SparkseeHA . . . 131

Maintenance and monitoring **133**

Backup . . . 133

Recovery . . . 135

Runtime information . . . 136

Logging . . . 136

Dumps . . . 136

Statistics . . . 137

Third party tools **141**

TinkerPop . . . 141

Blueprints . . . 141

Gremlin . . . 145

Rexster . . . 146

Pacer . . . 146

Introduction

About this document

This is the complete User Manual for Sparksee graph database which covers all the technical details relating to Sparksee, so it should be the go-to document for any doubts regarding this graph database. It has been divided into the following chapters & sections:

- Introduction

  This chapter includes notation convention, license details and support information.

- Graph Database

  This is an introductory chapter providing general graph database concepts and specific Sparksee graph database definitions.

- Installation

  This chapter explains how to install Sparksee in order to use is it as an embedded database in applications.

- API

  In this chapter developers learn how to use Sparksee APIs, with explanations and examples of all the functionalities.

- Configuration

  This chapter explains how to configure Sparksee, including how to monitorize the system, help with the deployment of applications and memory tuning.

- Scripting

  Information on the use of scripts to create a graph schema and loading data into the graph is available in this chapter.

- High availability

  This chapter explains every detail of this functionality that allows using Sparksee replicated.

- Maintenance and monitoring

  This chapter covers specific information about how to use Sparksee tools to monitorize and maintain the DB.

- Third-party tools

  This chapter reveals some tricks for using the Blueprints interface, and links to other third-party APIs.

This document is addressed to architects and developers.

The Sparksee manual is not intended to be followed as a tutorial document: examples are solely created to help understand the concepts explained. Every chapter may be consulted independently of the others, so the reader may hop directly to the chapter and section of interest.

Check other support documents for different approaches to learning how to use Sparksee graph database.

Notation

Please consider the following notation convention assumed for this document:

- This is a Java code-block example:

  Java

```
// Java code-block
```

- This is a C# code-block example:

  [C#]

```
// C# code-block
```

- This is a C++ code-block example:

  C++

```
// C++ code-block
```

- This is a Python code-block example:

  Python

```
# Python code-block
```

- This is an Objective-C code-block example:

  Objective-C

```
// Objective-C code-block
```

- This is the notation to refer a method from a class:

  ret class#method(type1 arg1, ...)

  Where:

- ret is the return type of the method
- class is the name of the class
- method is the name of the method itself
- type1 and arg1 are the type and name of the first argument respectively. When there is more than one argument, a comma-separated list of pair type-argument is given.

The notation may be simplified by leaving out some parts of the full signature.

License

Sparksee is distributed under a proprietary license considering the following variables:

- **Size of the graph**

  The license recognizes XSMALL (up to 1M objects in a single graph), SMALL (up to 10M objects), MEDIUM (up to 100M objects), LARGE (up to 1B objects) or VERY LARGE (more than 1B objects) graphs.

- **Number of concurrent sessions**

  1 session for the free trial or UNLIMITED for the rest of licenses.

- **HA functionality**

  This can be enabled or disabled.

Sparsity Technologies offers the following license agreements for Sparksee:

- **Free license**

  Sparksee is free to use, in its XSMALL size, for non-commercial purposes. Download Sparksee from the download section of our website.

- **Free commercial license**

  Sparksee is free to use, in XSMALL size, for commercial purposes, but recognition is requested. Please contact us.

- **Development license**

  Sparsity Technologies may offer development licenses for any size free of charge to qualified companies. Please check if you are qualified by requesting your development license.

- **Research license**

  Sparsity encourages research by offering free licenses to PhD students, academics and other university staff, for non-commercial purposes. Please check if you are qualified by requesting your research license.

- **Commercial license**

  Sparksee Commercial license allows the software, implemented with Sparksee, to be used commercially with options for SME or Corporate. Check Sparksee price list for commercial purposes.

- **Startup license**

  This is a special license for those companies inside Sparksee's partnership program. It includes a free development license (see conditions above) and a special commercial license. Please read the conditions and benefits of the Sparksee Partnership program.

Support

Sparksee technical support is offered online through the website in the documentation and support sections.

Documentation section

All the written information about Sparksee is available in this section. A developer will find the following relevant documents:

- **Starting Guide**

  Quick guide to building graphs with Sparksee. Includes sources of the examples ready to compile & run.

- **User Manual**

  What you are reading now is the complete manual for Sparksee. It should be the go-to document for everything regarding Sparksee.

- **HA manual**

  Separate manual for quick access to SparkseeHA's information.

- **Reference guides**

  Reference programming guides for each Sparksee's supported languages. They can be consulted online or offline (inside the Sparksee download).

In addition Sparksee documentation includes a brochure, presentations, tutorials and webinars, which are addressed to other stakeholders, such as CTOs and decision makers.

Support section

Sparksee has its own Google group for technical support. Questions and answers are shared between the members of the Sparksee community and the members of the Sparksee technical team.

Visit Sparksee discussion group.

Graph database

General concepts

In mathematics, a graph is an abstract representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by mathematical abstractions called vertices, and the links that connect some pairs of vertices are called edges. Typically, a graph is represented in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. The figure below is an example of this concept.

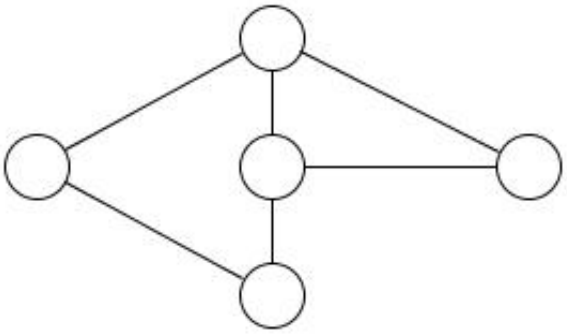


Figure 2.0: Graph

Vertices are also referred to as nodes and their properties are often called attributes. For the remainder of the document, graphs will be composed of **nodes**, **edges** and **attributes**.

Sparksee graph database

Sparksee is an embedded graph database management system tightly integrated with the application at code level. As a graph database, stored data is modeled as a graph.

Unlike relational databases where the data model is standard, graph database vendors propose different versions of the graph data model according to the description of a graph as explained in the previous section.

Graph data model

The Sparksee graph model is based on a generalization of the graph concept which can be defined as a **labeled attributed multigraph**. In Sparksee we refer to the label as the type.

These are its main features:

- All **node** and **edge** objects belong to a **type**.

- **Edges have a direction**. The source node is called the tail node and the target or destination node is called the head node.

  Edges can also be undirected. Sparksee undirected edges do not have a restricted direction; in fact they can be interpreted as bidirectional as both nodes play the head and tail roles at the same time.

  Considering this particularity, Sparksee graph model could also be called a mixed multigraph.

- Node and edge objects can have one or more **attributes**.

- As a multigraph, there are **no restrictions on the number of edges between two nodes**, even if those edges belong to the same type. In addition, loops are allowed.

This data model is more suitable for modeling complex scenarios such as the one in the Figure 2.1 which could be hardly represented using the simplest graph model. In Figure 2.1, there are two types of nodes (PEOPLE represented by a star icon and MOVIE shown as a clapperboard icon) both of which have an attribute (called respectively Name and Title) as well as a value. For instance the *Scarlett Johansson* (Name) node belongs to the PEOPLE type (star icon). Also there are two types of edges (DIRECTS shown in blue and CAST shown in orange). CAST (between PEOPLE and MOVIE) has an attribute called Character. Moreover, whereas DIRECTS is a directed edge, as it has an arrow pointing to its head node, CAST is an undirected edge type. More attributes could be added to both node and edge objects. Displaying the multigraph property, the *Woody Allen* node and *Manhattan* node are linked by two different edges.

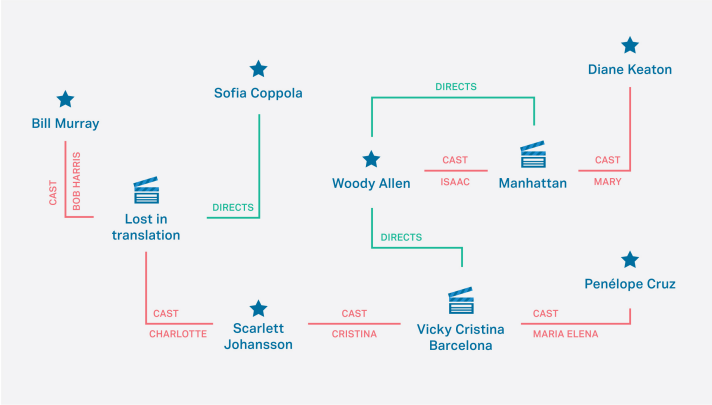


Figure 2.1: Sparksee multigraph

Types

Nodes and edges in Sparksee must be of a certain type.

All Sparksee types are identified by a public user-provided unique name, the *type name*, and an immutable Sparksee-generated unique identifier, the *type identifier*. The type identifier is used to refer the type when using Sparksee APIs as is explained in the 'Nodes and edges section of the 'API' chapter.

In Figure 2.1 the types created are PEOPLE and MOVIES (node types) and CAST and DIRECTS (edge types). Note that we refer to the types with their type name.

Node and edges

Sparksee objects are node or edge instances of a certain type. When they are created they are given an immutable Sparksee-generated unique identifier, the *object identifier* (OID). The OID is used to refer the object when using Sparksee APIs as is explained in the 'Nodes and edges' section from the 'API' chapter.

Nodes and edges must belong to a certain type and may have attributes.

In Figure 2.1, 18 objects (9 nodes and 9 edges) are displayed.

Attributes

Sparksee attributes are identified by a unique public user-provided name, the *attribute name*, and an immutable Sparksee-generated unique identifier, the *attribute identifier*. As in the case of type identifiers, an attribute identifier is used to refer the attribute when using Sparksee APIs as is explained in the 'Attributes and values' section of the 'API' chapter.

Sparksee considers the following kind of attributes:

- **Attributes defined within the scope of a type**, the most general type. For this type of attributes the name must be unique amongst all the other attributes defined for that type. Objects (node or edge objects) belonging to that type are the only ones allowed to set and get values for that attribute. For example, we could define the attribute ID for the PEOPLE and MOVIES node type, resulting in two different attributes. Thus, only PEOPLE objects would be able to use the first attribute while MOVIES objects would be able to use the second.
- **Node attributes**. They are not restricted to only one node type. For example, we could define the node attribute NAME for all the node objects of the graph, no matter which specific node type they belong to.
- **Edge attributes**. They are not restricted to only one edge type. For example, we could define the edge attribute WEIGHT for all the edge objects of the graph, no matter which specific edge type they belong to.
- **Global attributes**. They are not restricted to a node or edge type. For example, we could define the global attribute ID for all the objects of the graph, no matter which type they belong to. Take into account that the attribute name must be unique among all other global attributes.

Sparksee attributes are defined for a domain or data type; all values of an attribute belong to a specified data type with the exception of the null value, which does not belong to any data type. Valid Sparksee data types are:

- **Boolean** TRUE or FALSE values.
- **Integer** 32-bit signed integer values.
- **Long** 64-bit signed integer values.
- **Double** 64-bit signed double values.
- **Timestamp** Distance from Epoch (UTC) with millisecond precision. Valid timestamps must be within the range ['1970-01-01 00:00:01' UTC, '2038-01-19 03:14:07' UTC].
- **String** Unicode string values. Maximum length is restricted to 2048 characters.
- **Text** Large unicode character object. See 'Text attributes' in the 'Attributes and values' section of the 'API' chapter.
- **OID** Sparksee object identifier values.

Moreover, Sparksee attributes are univalued, which means that an object (node or edge) can only have one value for an attribute. Note that null may also be that value.

Figure 2.2 shows the attributes extracted from Figure 2.1. PEOPLE nodes have an *Id* and *Name*, MOVIES an *Id* and a *Title* and edges of type CAST have an attribute *Character* showing the name of the character of that actor in the movie. Note that we refer to the attributes by the attribute name.

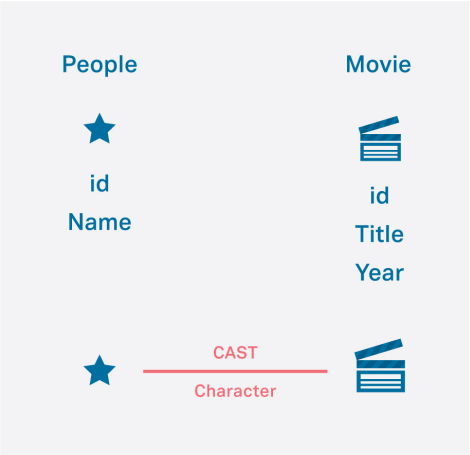


Figure 2.2: Sparksee attributes

Indexing

Attributes Different index capabilities can be set for each Sparksee attribute. Depending on these capabilities there are three types of attributes:

- **Basic attributes**: there is no index associated to the attribute.
- **Indexed attributes**: there is an index automatically maintained by the system associated to the attribute.
- **Unique attributes**: the same as for indexed attributes but with an added integrity restriction: two different objects cannot have the same value, with the exception of the null value.

Sparksee operations accessing the graph through an attribute will automatically use the defined index, significantly improving the performance of the operation. Note that only a single index can be associated to an attribute.

Edges A specific index can also be defined to improve certain navigational operations. Thus, the *neighbor* index can be set for an specific edge type to be used automatically by the neighbor API (see the 'Navigation operations' section of the 'API' chapter) significantly improving the performance of this operation.

Processing

A Sparksee-based application is able to manage more than one *database*, each of them working independently. It is important to keep in mind that a single database can be accessed by a single application or process at a time. Also the connection (open) to the database can only be made once.

Access to the database must be enclosed within a *session*, and multiple sessions can concurrently access the same database.

Sessions A *session* is a stateful period of a user's activity with a database; it can also be described as an instance of database usage.

Whereas a database can be shared among multiple threads, a session cannot because it is not thread-safe. Also **all manipulation of a database must be enclosed into a session**. A graph can only be operated inside a session.

Session responsibilities include management of transactions and temporary data.

Figure 2.3 shows a representation of a basic Sparksee-based application architecture where the application can manage multiple databases, each of them accessed by multiple threads and each handling a session.

Transactions A Sparksee transaction encloses a set of operations and defines the granularity level for the concurrent execution of sessions.

There are two types of transactions: Read or Shared, and Write or Exclusive. **Sparksee's concurrency model is based on the N-readers 1-writer model**, meaning that multiple read transactions can be executed concurrently whereas write transactions are executed exclusively.

When a transaction starts with the 'begin' instruction becomes self-defined for the operations it contains. Initially, a transaction starts as a read transaction and if a method updates the persistent graph database then it automatically

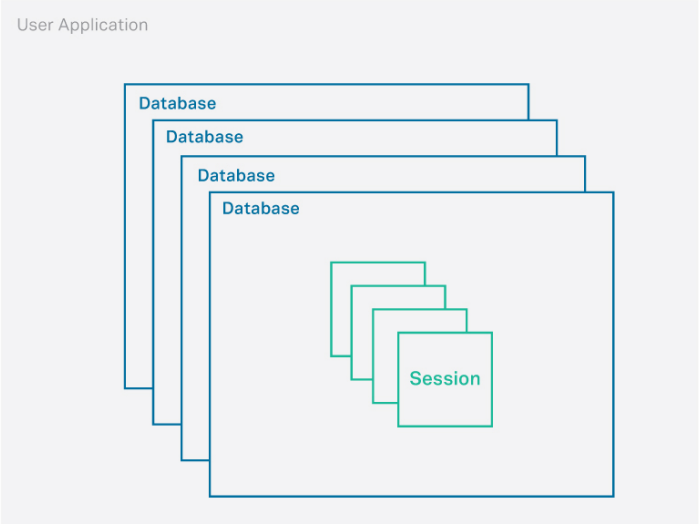


Figure 2.3: Sparksee application architecture

becomes a write transaction. To become a write transaction all other read transactions must have finished first. You can also directly start a write transaction by using the 'beginUpdate' instruction instead. That will avoid any possible lost update problem; but keep in mind Sparksee's concurrecny model when creating this type of transactions.

Users can manage transactions in two different ways:

- **Explicit use of transactions**: The user explicitly calls the beginning and end (commit or rollback) of a transaction (see 'Transactions' section of the 'API' chapter). All operations between the beginning and the end of the transaction are grouped within a single transaction.
- **Autocommited mode**: A transaction is automatically started before each operation and automatically closed when the operation finishes. This results in transactions of size 1 in number of operations.

Explicit use of transactions may improve the performance of concurrently executed sessions, so it is highly recommended.

Temporary data Some operations may require the use of temporary data. This temporary data is automatically managed by the session removing it when the session is closed. For this reason, temporary data may also be referred as Session data.

Large collections of object identifiers, its iterators and session attributes are examples of temporary data.

Session attributes are a further example of temporary data. Whereas attributes are persistent in the graph database, session attributes are temporary and exclusive for a session:

- As they are automatically removed when the session is closed, these attributes do not require a user-provided name identifier.
- Session attributes can be global too.
- Session attributes are only manipulated within their session, so the other sessions cannot see them.

Installation

System requirements

Sparksee is available for several platforms:

- **Windows**, both 32 and 64 bit.
- **Linux**, both 32 and 64 bit.
- **Mac OS X**.
- Mobile: **Android, iOS, QNX**

Furthermore, Sparksee is available for several programming languages:

- **Java**, requiring a JVM 5.0 or newer.
- **.NET**, requiring Microsoft .NET 2.0 or newer.
- **C++**, requiring libc6 (Linux) or MS Visual Studio 2005 (Windows), or newer.
- **Python**, requiring Python v2.7.
- **Objective-C**

Sparksee has a small footprint and it does not have any specific requirements for the memory size or disk capacity of the platform. Thus, these aspects will solely depend on the user's data size and application.

Download

Sparksee is available for several programming languages, so the first thing to do is download the right package for a development platform and language. All the packages can be downloaded in the download section of the Sparsity technologies website.

Java developers can also get Sparksee through Apache Maven instead of manually downloading the packages. More information is available in Sparksee maven project.

Available packages:

- **Sparkseejava:** this is the package for all new Sparksee Java users.
- **Sparkseenet:** Sparksee is natively available for .NET developers.
- **Sparkseecpp:** a C++ interface is also available. Sparksee core is C++, so we punch it another level to the API.
- **Sparkseepython:** Sparksee is available for Python 2.7 developers.
- **Sparkseeobjc:** A native Objective-C api is provided for iOS and MacOS developers. The C++ interface can still be used in Objective C projects.

Once Sparksee is downloaded and unpacked, the content of most packages should look like this:

- doc: this is the directory that contains Sparksee API **reference** documentation in html or the specific format for the different platforms.
- lib: a directory with Sparksee libraries. This may have subdirectories when different files are required regarding each operating system.
- ReleaseNotes.txt: a text file with basic information on the latest release changes. release changes.
- LICENSE.txt: contains Sparksee licensing information.

According to the programming language particularities, some packages may include additional content.

Sparksee package is now ready to be used. No further installation is required, with the exception of Objective-C on MacOS, where an installer is provided to easily copy the framework to the right standard directory.

However, some additional steps can be taken to make Sparksee usage easier for every language.

We will use the HelloSparksee application for the examples which is available for download here.

Java

All the libraries required to develop a java application with Sparksee are contained in a jar file located at the lib directory (sparkseejava.jar).

In Windows, the core of Sparksee are native C++ libraries that require the appropiate Microsoft Visual C Runtime installed. The current java version requires the MSVC 2012 runtime.

If the user is not using Maven, a path to this file should be added into the CLASSPATH environment variable. However, to avoid any misunderstandings, in this document we will explicitly set the *classpath* for each of the commands. The HelloSparksee application can be compiled and run as follows:

```
$ javac -cp sparkseejava.jar HelloSparksee.java
$ java -cp sparkseejava.jar;. HelloSparksee
```

With Maven

With Apache Maven it is even easier. Since Sparksee is in the maven central repository, adding the dependency to the correct Sparksee version into a pom.xml file should be enough:

```
<dependency>
  <groupId>com.sparsity</groupId>
  <artifactId>sparkseejava</artifactId>
  <version>5.0.0</version>
</dependency>
```

With Android

The procedure to use Sparkseejava in Android is the same for Eclipse and Android Studio, but we have separated some steps to better explain the procedure in each environment.

- Copy the sparkseejava.jar file from the **lib/** directory to the **libs/** directory of your android project.
- If you want different ".apk" files for the different target architectures instead of a single application file that supports all platforms or you only want to support certain architectures, you just need to remove from inside the sparkseejava.jar the subdirectories of the platforms that you don't want.
- Refresh the project explorer if you can't see the copied files.

Android Studio

- Right click on the **libs/sparkseejava.jar** file and select Add as ...library.
- Verify that in your **build.gradle** you have a command to compile the jar file. Probably there will be the command compile fileTree(dir: 'libs', include: ['*.jar']) If it's not present you could add this command compile files('libs/sparkseejava.'jar) to compile at least the **sparkseejava.jar** file.
- Set a minimum sdk version >= 9 in the **build.gradle** file.

Using Eclipse

- Right click on the **libs/sparkseejava.jar** file and select Build Path > Add to Build Path.
- Set in your **AndroidManifest.xml** a minimum sdk version greater or equal to 9.

.NET

The .NET package contains subdirectories in the lib directory for each platform and compiler (windows32_VS2012, windows64_VS2012, ...). You should chose the right one for 32 or 64 bit systems and your complier. For each system, the main library included is sparkseenet.dll which should be included in the .NET projects.

All the other available libraries in the package are native dlls that sparkseenet .dll must be able to find at run time. Although it is not a requirement, these libraries may be copied to the system folder.

The core of Sparksee are native C++ libraries that require the appropiate Microsoft Visual C Runtime installed on all the computers running your application.

MS Visual Studio users

If you are a MS Visual Studio IDE developer, you need to add the reference to the Sparksee .NET library (**sparkseenet.dll**) in your project and set the build platform to the appropiate specific platfrom (x64 or x86). Please, check that in the "Configuration Manager" from the "BUILD" menu of Visual Studio, the "Platform" selected for the build is NOT "Any CPU". If it's "Any CPU", you must select "New" to set a "x86" or "x64" build target.

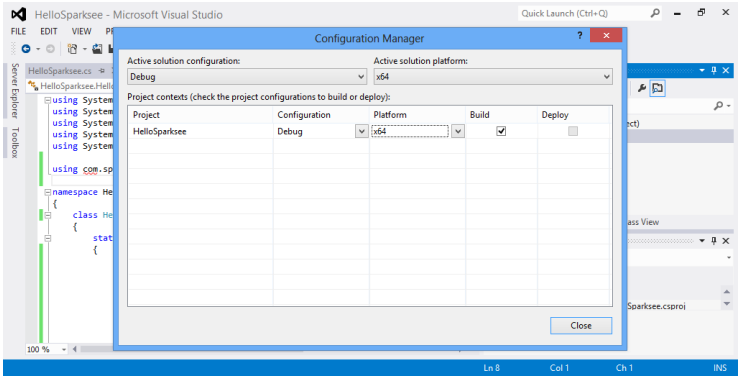


Figure 3.1: .Net compilation - setting the platform

Since all the other libraries included in the package are native libraries that will be loaded at runtime, they must be available too.

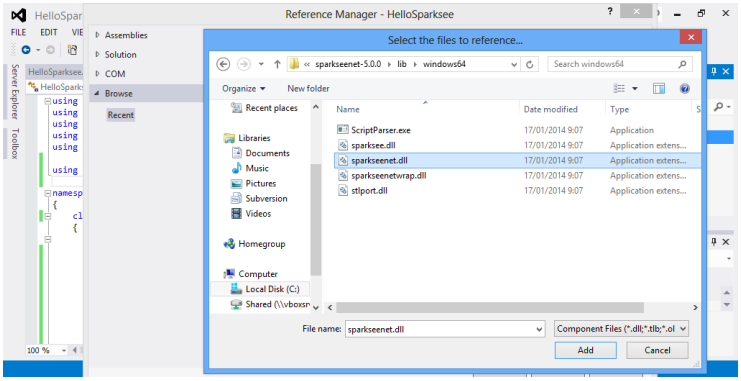


Figure 3.2: .Net compilation - adding the reference

The best option is to copy all the other ".dll" files into the same directory where your application executable file will be.

Using the development environment, this can be done using the option Add existing Item, choosing to see Executable files and selecting the other three native libraries (sparksee.dll, sparkseenetwarp.dll and stlport.dll).

Next you must select the three libraries in the Solution Explorer window and set the property Copy to Output as Copy Always for all three native libraries.

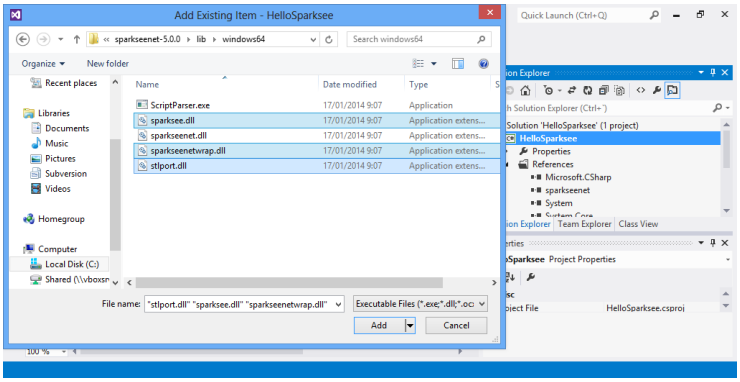


Figure 3.3: .Net compilation - adding existing item

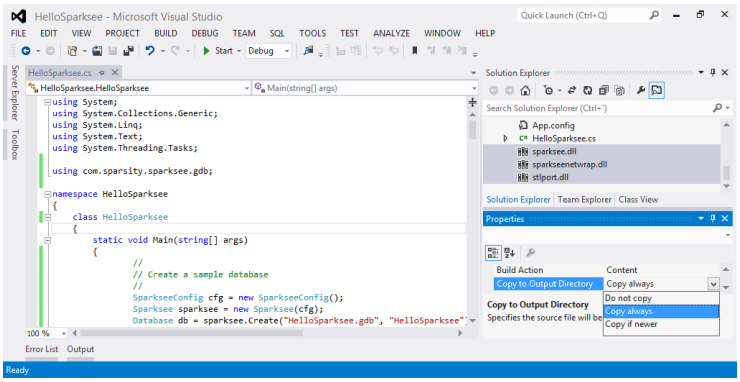


Figure 3.4: .Net compilation - copy to output

Instead of copying the native libraries to the application target directory, an alternative would be to put all the native ".dll" files into your Windows system folder (System32 or SysWOW64 depending on your Windows version).

Now the application is ready to be built and run like any Visual Studio project.

Command-line users

Alternatively the command-line interface might be used as well to test the sample application.

First setup a compiler environment with the vsvars32.bat (or vcvarsall.bat) for 32 bit MS Visual Studio.

```
> call "C:\Program Files\Microsoft Visual Studio 11.0\Common7\Tools\vsvars32.bat"
```

or with the vcvarsall.bat file with the appropiate argument for 64 bit MS Visual Studio.

```
> call "C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat"
    amd64
```

Then the application can be compiled and run (assuming all the libraries have already been copied to the same directory) as follows:

```
> csc /out:HelloSparksee.exe /r:sparkseenet.dll HelloSparksee.cs
> HelloSparksee.exe
```

C++

The C++ lib folder contains native libraries for each available platform (Windows 32/64 bit, Linux 32/64 bit and MacOS 64 bit, ...).

In Linux and MacOS, you may want to add the path to the correct subdirectory to your LD_LIBRARY_PATH (linux) or DYLD_LIBRARY_PATH(MacOS) environment variables.

In Windows the libraries can either be copied to the system folders or the user must take care to always include them in the projects.

The rest of the files included in this package, such as the ones in the includes directory, will be needed at compilation time.

For iOS you have to uncompress the ".dmg" file to get the **Sparksee.framework** directory. This directory contains the include files, the static library and the documentation. But you may prefer to use the Objective-C version of Sparksee.

The Sparksee C++ interface contains include files and dynamic libraries in order to compile and run an application using Sparksee. The general procedure is to first add the include directories to your project, then link with the supplied libraries corresponding to your operating system and finally copy them to any place where they can be loaded at runtime (common places are the same folder as the target executable file or your system libraries folder).

Let's have a look at a more detailed description of this procedure in the most common environments.

Windows users

When using the development environment Microsoft Visual Studio, the first step should be to add to the Additional include directories, C++ general property of the project, the sparksee subdirectory of the includes folder from the Sparksee package.

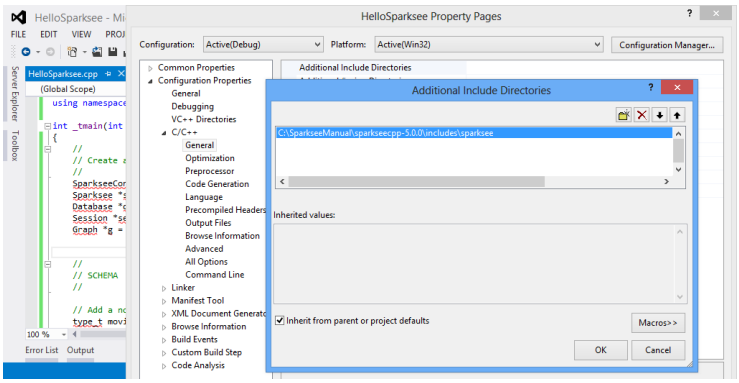


Figure 3.5: C++ compilation - include directories

This must also be done with the library directory, so the Additional library directories linker general property must be edited to add the correct subdirectory of the Sparksee lib folder for the operating system and compiler used.

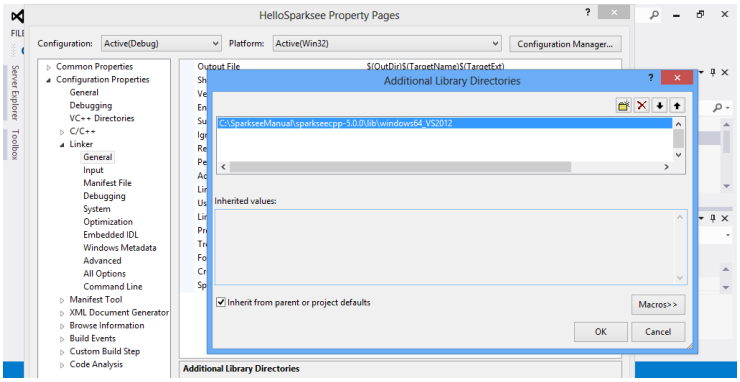


Figure 3.6: C++ compilation - add library directories

After this, the *sparksee* library should be added to the Additional Dependencies linker input property.

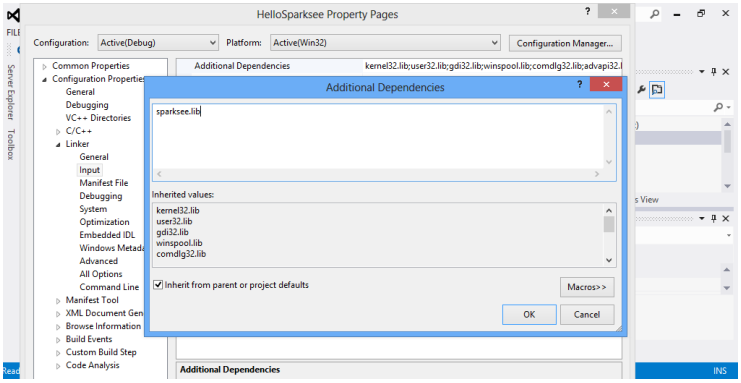


Figure 3.7: C++ compilation - add dependencies

Finally ensure that the dll files can be found at run time. An easy way to do this is to add a post-process in the project to copy the dll files to the output folder where the application executable will be built.

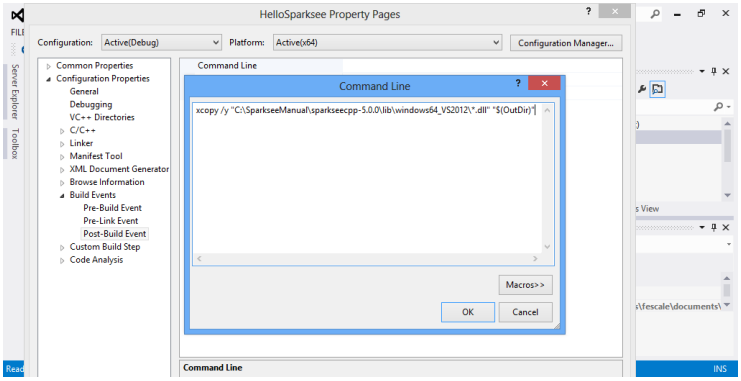


Figure 3.8: C++ compilation - post build event

An alternative would simply be to put all the native *.dll files into the Windows system folder (System32 or SysWOW64 depending on the Windows version).

It's important to check that the build platform selected matches the libraries that you are using.

Now it is ready to build and run the application like any MS Visual Studio project.

Optionally, to quickly test the HelloSparksee sample application, the command line can be used instead of the MS Visual Studio approach. First, set up the compiler environment with the vsvars32.bat (or vcvarsall.bat) file with the 32-bit MS Visual Studio.

```
> call "C:\Program Files\Microsoft Visual Studio 11.0\Common7\Tools\vsvars32.bat"
```

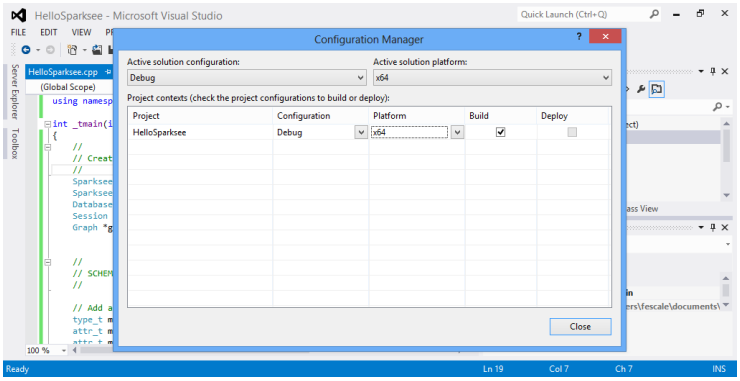


Figure 3.9: C++ compilation - platform

or with the vcvarsall.bat file with the appropiate argument for the 64-bit MS Visual Studio.

```
> call "C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat"
    amd64
```

Then to compile and run the application (example on a 32-bit Windows):

```
> cl /I"path_to_the_unpacked_sparksee\includes\sparksee" /D "WIN32" /D "_UNICODE"
    /D "UNICODE" /EHsc /MD /c HelloSparksee.cpp

> link /OUT:"HelloSparksee.exe" HelloSparksee.obj /LIBPATH:"
    path_to_the_unpacked_sparksee\lib\windows32" "sparksee.lib"

> xcopy /y "path_to_the_unpacked_sparksee\lib\windows32\*.dll" .

> HelloSparksee.exe
```

Linux/MacOS users

We are not going to focus on any specific integrated development environment for Linux or Mac OS as that is beyond the scope of this manual. Instead we will give an explanation of the procedure which can be adapted to the specifics of any development environment.

- In the includes directory, there is the subdirectory sparksee that must be added as included search directories in the project.
- The lib directory contains a subdirectory for each available operating system. The correct directory for the computer should be added as a link search directory in the project.
- To link the application, the **sparksee**, and the **pthread** libraries must be used in this order.

- Finally the directory where the libraries can be found may be added to the LD_LIBRARY_PATH, or DYLD_LIBRARY_PATH in MacOS, environment variable to ensure that they will be found at runtime.

Optionally, to quickly test the HelloSparksee sample application, the command line can be used as well:

```
$ g++ -I/path_to_the_unpacked_sparksee/includes/sparksee -o HelloSparksee.o -c
    HelloSparksee.cpp

$ g++ HelloSparksee.o -o HelloSparksee -L/path_to_the_unpacked_sparksee/lib/
    linux64 -lsparksee -lpthread

$ export LD_LIBRARY_PATH=/path_to_the_unpacked_sparksee/lib/linux64/

$ ./HelloSparksee
```

Android

The Android is not very different from the Linux usage.

- You also must add the **includes/sparksee** directory to the includes search path.
- The sparksee dynamic library and the provided stlport_shared library must be included in the applicacion. There are 4 versions of each library in subdirectories from the **lib/** directory. You must use the appropiate libraries for the processor target of your project.
- The **stlport_shared** is the NDK library must be included in the application instead of just being linked.
- The **z** and **dl** libraries from the Android NDK must be linked too.
- Setting an explicit memory limit to the Sparksee cache (using the SparkseeConfig class) is highly recommended. For more information about SPARKSEE cache and the SPARKSEEConfig class check the Configuration chapter and the reference guides.

iOS

Once you have extracted the **Sparksee.framework** directory from the distribution ".dmg" file, the basic steps to use Sparksee in your Xcode application project are the following:

- Add the Sparksee include files to the search path in your application project: The path to Sparksee.framework/Resources/sparksee/ must be added as non-recursive to the User Header Search Paths option on the build settings of your xcode application project. This is required because Sparksee include files use a hierarchy of directories which is not usual in an xcode framework. Therefore, they can't be in the regular headers' directory of the framework.

- Add the **Sparksee.framework** to the Link Binary With Libraries build phase of your application project. You can just drag it there.

- Choose the appropiate library: libstdc++ (GNU C++ standard library) or libc++ (LLVM C++ standard library with C++11 support) in the C++ Standard Library option in the build settings of the compiler. The option choosen must match the downloaded version of the Sparkseecpp for iOS.

- Remember that all the source files using C++ should have the extension ".mm" instead of ".m".

- Take into account that, after all these changes, a Clean of your Project may be needed.

- Setting an explicit memory limit to the Sparksee cache (using the SparkseeConfig class) is highly recommended. For more information about SPARKSEE cache and the SPARKSEEConfig class check the Configuration chapter and the reference guides.

Python

Python lib contains the native libraries for each available platform (Windows 32/64 bits, Linux 32/64 bits and MacOS 64 bits) in addition to the Python module.

Make sure to include both the Python module (called sparksee.py) and the wrapper library named "_sparksee" in an accessible folder for Python, please refer to Python module search path for more information about the search path.

The other available library included in Sparksee's Python distribution is the dynamic native library. This must be located in a specific system directory defined by the following:

- DYLD_LIBRARY_PATH for MacoOSX systems.

- LD_LIBRARY_PATH for Linux systems.

- PATH environment variable for Windows systems.

Once the installation is completed you can run the script normally. The following example assumes that the python2.7 executable is available in your path, otherwise you should write the full path to your Python executable.

```
$ python ./HelloSparksee.py
```

Objective-C

For the Objective-C Mac OS and iOS versions, the content of the dmg file is slightly different because, you will either find the **Sparksee.framework** directory or its installer. The documentation is in the framwork subdirectory Resources/Documentation.

- **MacOS**

  The dmg file for Mac OS contains an installer (SparkseeInstaller.pkg) to run. This will extract the framework directory to the right location (/Library/Frameworks/Sparksee.framework).

  If you are asked for a target disk, you must choose the main system disk. Your application will depend on the Sparksee dynamic library, so it will need to find it in the right location at runtime.

- **iOS**

  The dmg file for iOS contains the **Sparksee.framework** directory without an installer because you can copy it to any place.

  When building your application the Sparksee library will be embedded. Sice it's not a dynamic library framework, it doesn't need to be found at any specific location at runtime.

The MacOS and iOS versions can be included in you application project in the same way and could be used equally from the source code. But the installation of the framework and deployment of your application is slightly different.

MacOS

We have seen how to download, unpack the dmg file and install the Sparksee package to get the **Sparksee.framework** installed in the /Library/Frameworks / directory. The Mac OS version of the framework is a standard framework containing a dynamic library, so it's installed in a fixed standard location.

To use the Sparksee Objective-C framework in your application, you have to add the **Sparksee.framework** from /Library/Frameworks/Sparksee.framework to the Link Binary With Libraries build phase of your application project.

Then you can import the header in your source code.

```
#import <Sparksee/Sparksee.h>
```

Take into account that your application will depend on the Sparksee dynamic library, therefore the Sparksee framework must be installed on the target computers either manually or redistributed with your own installer.

Alternatively, you could include the framework as a private framework inside your application, but then a modification of the framework library location on the @executable_path instead of the standard /Library/Frameworks/ would be required.

iOS

We have already seen how to download and unpack the dmg file. For iOS, you don't need to install the framework because it would be already there. You can copy it directly to wherever you want because there is not any established standard dynamic library framework.

To use the Sparksee Objective-C framework in your application, you have to add the **Sparksee.framework** to the Link Binary With Libraries build phase of your application project. You can just drag it there.

If you don't use C++ code in your project, you have to explicitly add the right C++ standard library because the Sparksee library core depends on it. Click on the "+" sign of the same "Link Binary With Libraries" build phase of your application project, then select the appropiate C++ library ("libc++.dylib" for LLVM C++11 version or "libstdc++.6.dylib" for the GNU C++ version) and finally click the "Add" button.

If you use C++ in your code, choose the right library ("libstdc++ (GNU C++ standard library)" or "libc++ (LLVM C++ standard library with C++11 support)") in the "C++ Standard Library" option in the build settings of the compiler depending on the Sparksee version downloaded.

Then you can import the header in your source code.

```
#import <Sparksee/Sparksee.h>
```

Your iOS application will contain the Sparksee library embedded (it's a static library), so your application deployment should be exactly the same as any other iOS application.

Setting an explicit memory limit to the Sparksee cache (using the SparkseeConfig class) is highly recommended. For more information about SPARKSEE cache and the SPARKSEEConfig class check the Configuration chapter and the reference guides.

API

This chapter explains how to use Sparksee API to manage a Sparksee graph database and perform basic operations. Later sections include a detailed explanation of the following topics:

- Construction of a database
- Definition of node and edge types
- Creation of node and edge objects
- Definition and use of attributes
- Query nodes and edges as well as attributes and values
- Management of object collections

Moreover, higher functionality also available in the API is explained as well, including:

- Import/export data
- Scripting
- Graph algorithms

Most of the functionality provided by the Sparksee API is included in the com.sparsity.sparksee.gdb package or namespace in Sparkseejava and Sparkseenet respectively, and in the sparksee::gdb namespace in Sparkseecpp. If not, the specific package or namespace is indicated.

Database construction

The construction of a Sparksee database includes the following steps:

1. **Initialize the configuration**. The basic default configuration only requires the instantiation of the SparkseeConfig class. See the 'Configuration' chapter for a more detailed explanation of this instantiation & also check the advanced configuration options.

2. Second, a **new instance of Sparksee** must be created. The configuration previously created is set-up during this instantiation. Sparksee is a Database factory which allows a Sparksee graph database to be opened or created.

3. Now it is time to **create the database** itself. A single Sparksee instance can open or create N different databases, but a database can only be concurrently accessed once - take a look at the 'Processing' section of the 'Graph database' chapter for more information. Each database created must have a unique path. Database is considered the Session factory.

4. A Session is a stateful period of activity of a user with a database. Therefore, the manipulation of a Database must be enclosed into a **Session instance**. A Database can only be accessed once, and multiple Session instances can be obtained from a Database instance. However, as sessions are not thread-safe, each thread must manage its own session. This is further explained in the 'Processing' section of the 'Graph Database' chapter

5. A Graph instance must be obtained from the Session, to perform graph-based operations on the graph database.

6. Sparksee, Database and Session must be closed (deleted in the case of C++) when they are no longer in use, to ensure all data is successfully flushed and persistently stored. Moreover, it is important that they are closed in the correct order, which is the reverse of their creation.

This example shows the construction of a new empty graph database. To open an already-existing Sparksee graph database the method Sparksee#open should be used instead of Sparksee#create. Also, it is possible to open the database in read-only mode by using the parameter read from the Sparksee#open method:

Java

```
import com.sparsity.sparksee.gdb.*;

public class SparkseejavaTest
{
    public static void main(String argv[])
    throws java.io.IOException, java.lang.Exception
    {
        SparkseeConfig cfg = new SparkseeConfig();
        Sparksee sparksee = new Sparksee(cfg);
        Database db = sparksee.create("HelloSparksee.gdb", "HelloSparksee");
        Session sess = db.newSession();
        Graph graph = sess.getGraph();
        // Use 'graph' to perform operations on the graph database
        sess.close();
        db.close();
        sparksee.close();
    }
}
```

[C#]

```
using com.sparsity.sparksee.gdb;

public class SparkseenetTest
{
    public static void Main()
    {
        SparkseeConfig cfg = new SparkseeConfig();
        Sparksee sparksee = new Sparksee(cfg);
        Database db = sparksee.Create("HelloSparksee.gdb", "HelloSparksee");
        Session sess = db.NewSession();
        Graph graph = sess.GetGraph();
        // Use 'graph' to perform operations on the graph database
        sess.Close();
        db.Close();
        sparksee.Close();
    }
}
```

C++

```
#include "gdb/Sparksee.h"
#include "gdb/Database.h"
#include "gdb/Session.h"
#include "gdb/Graph.h"
#include "gdb/Objects.h"
#include "gdb/ObjectsIterator.h"

using namespace sparksee::gdb;

int main(int argc, char *argv[])
{
    SparkseeConfig cfg;
    Sparksee *sparksee = new Sparksee(cfg);
    Database * db = sparksee->Create(L"HelloSparksee.gdb", L"HelloSparksee");
    Session * sess = db->NewSession();
    Graph * graph = sess->GetGraph();
    // Use 'graph' to perform operations on the graph database
    delete sess;
    delete db;
    delete sparksee;
    return EXIT_SUCCESS;
}
```

Python

```
# -*- coding: utf-8 -*-
import sparksee

def main():
    cfg = sparksee.SparkseeConfig()
    sparks = sparksee.Sparksee(cfg)
    db = sparks.create(u"Hellosparks.gdb", u"HelloSparksee")
    sess = db.new_session()
    graph = sess.get_graph()
    # Use 'graph' to perform operations on the graph database
    sess.close()
    db.close()
    sparks.close()

if __name__ == '__main__':
  main()
```

Objective-C

```
#import <Foundation/Foundation.h>
#import <Sparksee/Sparksee.h>

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        STSSparkseeConfig *cfg = [[STSSparkseeConfig alloc] init];
        // The license key is required for mobile versions.
        //[cfg setLicense: @"THE_LICENSE_KEY"];
        STSSparksee *sparksee = [[STSSparksee alloc] initWithConfig: cfg];
        // If you are not using Objective-C Automatic Reference Counting , you
        // may want to release the cfg here, when it's no longer needed.
        //[cfg release];
        STSDatabase *db = [sparksee create: @"HelloSparksee.gdb" alias: @"
            HelloSparksee"];
        STSSession *sess = [db createSession];
        STSGraph *graph = [sess getGraph];
        // Use 'graph' to perform operations on the graph database
        [sess close];
        [db close];
        [sparksee close];
```

```
        // If you are not using Objective-C Automatic Reference Counting , you
        // may want to release the sparksee here , when it's closed.
        //[sparksee release];
    }
    return 0;
}
```

Cache configuration

Once an Sparksee instance is created using a SparkseeConfig (see the 'Configuration' chapter), most configuration settings can not be modified. The Cache maximum size is an exception. In a server dedicated to running the Sparksee graph database, once the Sparksee instance is created, you may never need to modify your initial settings. But in a device were Sparksee is not it's main process, a dynamic modification of the maximum cache in use may be required.

For example, in a mobile platform the OS may require an application to release memory in order to give this scarce resource to another application. If you fail to release the memory, the process can be stopped. To handle this dynamic change there are a few new Database methods that can be helpful.

- GetCacheMaxSize Returns the current cache maximum setting in megabytes.
- SetCacheMaxSize Sets a new value for the cache maximum size (in megabytes). It may fail if the new value is too big or too small.
- FixCurrentCacheMaxSize The current cache in use may have not reached the maximum setting yet. If it's less than the maximum and more than the minimum required for the current pools, the current cache usage will be set as the maximum, so it will not grow anymore.

Java

```
SparkseeConfig cfg = new SparkseeConfig();
cfg.setCacheMaxSize(1024); // 1GB Cache
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.create("HelloSparksee.gdb", "HelloSparksee");
Session sess = db.newSession();
Graph graph = sess.getGraph();
...
db.setCacheMaxSize(db.getCacheMaxSize()/2); // Try to reduce the Cache maximum in
     half
...
```

[C#]

```
SparkseeConfig cfg = new SparkseeConfig();
cfg.SetCacheMaxSize(1024); // 1GB Cache
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.Create("HelloSparksee.gdb", "HelloSparksee");
Session sess = db.NewSession();
Graph graph = sess.GetGraph();
...
```

```
db.SetCacheMaxSize(db.GetCacheMaxSize()/2); // Try to reduce the Cache maximum in
    half
...
```

C++

```
SparkseeConfig cfg;
cfg.SetCacheMaxSize(1024); // 1GB Cache
Sparksee *sparksee = new Sparksee(cfg);
Database * db = sparksee->Create(L"HelloSparksee.gdb", L"HelloSparksee");
Session * sess = db->NewSession();
Graph * graph = sess->GetGraph();
...
db->SetCacheMaxSize(db->GetCacheMaxSize()/2); // Try to reduce the Cache maximum
    in half
...
```

Python

```
cfg = sparksee.SparkseeConfig()
cfg.set_cache_max_size(1024) // 1GB
sparks = sparksee.Sparksee(cfg)
db = sparks.create(u"Hellosparks.gdb", u"HelloSparksee")
sess = db.new_session()
graph = sess.get_graph()
...
db.set_cache_max_size(db.get_cache_max_size()/2) // Try to reduce the Cache
    maximum in half
...
```

Objective-C

```
STSSparkseeConfig *cfg = [[STSSparkseeConfig alloc] init];
[cfg setCacheMaxSize: 1024]; // 1GB
STSSparksee *sparksee = [[STSSparksee alloc] initWithConfig: cfg];
STSDatabase *db = [sparksee create: @"HelloSparksee.gdb" alias: @"HelloSparksee"];
STSSession *sess = [db createSession];
STSGraph *graph = [sess getGraph];
...
[db setCacheMaxSize: [db getCacheMaxSize]/2]; // Try to reduce the Cache maximum
    in half
...
```

Nodes and edges

A graph database is a set of objects (nodes and edges) where each object belongs to a type. Node and edge types define the schema of the graph database and they are required to create new objects.

All types have a unique user-provided string identifier, the *type name* as well as a unique and immutable Sparksee-generated numeric identifier, the *type identifier*. The method Graph#newNodeType creates a new type of node.

The type identifier will be used to refer that type in all of the APIs requiring that information. For example: the Graph#newNode method, that creates a

new node object, needs a single argument which is the node type identifier to establish that the new object will belong to that type.

Whereas node types only require a name for their construction, edge types have other options. Edge types can be directed or undirected and restricted or non-restricted. These topics are explained in the 'Types' section of the 'Graph database' chapter. Check out the parameters of Graph#newEdgeType and Graph #newRestrictedEdgeType methods in the reference guides to see how to set the different options. In addition, a specific index to improve some operations (such as neighbor retrieval) can be defined for edge types. To fully understand the benefits of this index, see the 'Indexing' section of the 'Graph database' chapter.

When a node or edge object is created (with Graph#newNode or Graph#newEdge respectively), a unique and immutable Sparksee-generated numeric identifier is returned. This identifier is known as the *object identifier* or OID. Thus, all operations on a node or edge object will require this OID as a parameter.

Note that on certain languages the api methods and classes may have slightly different names in order to conform to the language conventions. For instance, all the classes in Objective-C have a STS prefix (from Sparsity Technologies Sparksee), so the Graph#newNode and Graph#newEdge methods become STSGrah #createNode and STSGrah#createEdge. In the document explanations we will usually reference the classes and methods using the most common names (without prefixes), but the exact names can be seen on the code samples and can easily be found on the specific api language reference documentation.

The following examples include the creation of types and objects. We are creating 2 "PEOPLE" which have a relationship between them, "FRIEND".

Java

```
Graph graph = sess.getGraph();
...

int peopleTypeId = graph.newNodeType("PEOPLE");
int friendTypeId = graph.newEdgeType("FRIEND", true, true);

long people1 = graph.newNode(peopleTypeId);
long people2 = graph.newNode(peopleTypeId);
long friend1 = graph.newEdge(friendTypeId, people1, people2);
```

[C#]

```
Graph graph = sess.GetGraph();
...

int peopleTypeId = graph.NewNodeType("PEOPLE");
int friendTypeId = graph.NewEdgeType("FRIEND", true, true);

long people1 = graph.NewNode(peopleTypeId);
long people2 = graph.NewNode(peopleTypeId);
long friend1 = graph.NewEdge(friendTypeId, people1, people2);
```

C++

```
Graph * graph = sess->GetGraph();
...
```

```
type_t peopleTypeId = graph->NewNodeType(L"PEOPLE");
type_t friendTypeId = graph->NewEdgeType(L"FRIEND", true, true);

oid_t people1 = graph->NewNode(peopleTypeId);
oid_t people2 = graph->NewNode(peopleTypeId);
oid_t friend1 = graph->NewEdge(friendTypeId, people1, people2);
```

Python

```
graph = sess.get_graph()
...

people_type_id = graph.new_node_type(u"PEOPLE")
friend_type_id = graph.new_edge_type(u"FRIEND", True, True)

people1 = graph.new_node(people_type_id)
people2 = graph.new_node(people_type_id)
friend1 = graph.new_edge(friend_type_id, people1, people2)
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...

int peopleTypeId = [graph createNodeType: @"PEOPLE"];
int friendTypeId = [graph createEdgeType: @"FRIEND" directed: TRUE neighbors: TRUE
    ];

long long people1 = [graph createNode: peopleTypeId];
long long people2 = [graph createNode: peopleTypeId];
long long friend1 = [graph createEdge: friendTypeId tail: people1 head: people2];
```

Complementary to adding new node or edge types or adding new node or edge objects, Sparksee allows the removal of node or edges types and objects. Specifically, the method Graph#removeType removes a type and the method Graph# drop removes an object.

Other methods to interact with the schema are Graph#findType to discover if a type already exists, TypeList Graph#findTypes to retrieve all existing types and Type Graph#getType to get a specific type to be used in other operations.

The following examples create new node types called “PEOPLE” and edge types called “FRIEND”, if they do not previously exist. Then all existing types are traversed and removed. Note that the following examples contain methods regarding attributes that are explained in the next section, and are solely here to illustrate what kind of methods will be using the results from the types’ methods:

There are several restrictions to be able to remove a type:

- It can not have any attribute. If it has attribute, you have to remove it’s attributes first.
- If it’s a node type, It can not exist any restricted edge type using using the node type as head or tail. If that’s the case, you have to remove the restricted edge type first.

- If it's a node type, It can not exist any edge from an unrestricted edge type using any node of this type as a head or tail. If that's the case, you have to drop the edge before trying to remove the node type.

In the following example, we could use FindTypes and try to remove all the types in one loop. But if we try to remove a node type with any of the previous restrictions before removing the conflicting edge types (or edges), the operation would fail.

So we will first get only the edge types (with FindEdgeTypes) instead of getting all the types (with FindTypes) to remove only the edge types. The procedure could then be safely repeated for the node types (with FindNodeTypes or with FindTypes because only node types remain) without the risk of any operation failing for the previous restricions because all the edge types would have been removed before. But we don't show it in the sample code because it would be exactly the same changing only the method to get the types (FindNodeTypes).

Java

```
Graph graph = sess.getGraph();
...
int peopleTypeId = graph.findType("people");
if (Type.InvalidType == peopleTypeId)
{
    peopleTypeId = graph.newNodeType("people");
}
int friendTypeId = graph.findType("friend");
if (Type.InvalidType == friendTypeId)
{
    friendTypeId = graph.newEdgeType("friend", true, true);
}
...
TypeList tlist = graph.findEdgeTypes();
TypeListIterator tlistIt = tlist.iterator();
while (tlistIt.hasNext())
{
    int type = tlistIt.next();
    Type tdata = graph.getType(type);
    System.out.println("Type " + tdata.getName() + " with " + tdata.getNumObjects
        () + " objects");

    AttributeList alist = graph.findAttributes(type);
    AttributeListIterator alistIt = alist.iterator();
    while (alistIt.hasNext())
    {
        int attr = alistIt.next();
        Attribute adata = graph.getAttribute(attr);
        System.out.println(" - Attribute " + adata.getName());

        graph.removeAttribute(attr);
    }

    graph.removeType(type);
}
```

[C#]

```
Graph graph = sess.GetGraph();
...
int peopleTypeId = graph.FindType("people");
if (Type.InvalidType == peopleTypeId)
{
    peopleTypeId = graph.NewNodeType("people");
```

```
}
int friendTypeId = graph.FindType("friend");
if (Type.InvalidType == friendTypeId)
{
    friendTypeId = graph.NewEdgeType("friend", true, true);
}
...
TypeList tlist = graph.FindEdgeTypes();
TypeListIterator tlistIt = tlist.Iterator();
while (tlistIt.HasNext())
{
    int type = tlistIt.Next();
    Type tdata = graph.GetType(type);
    System.Console.WriteLine("Type " + tdata.GetName() + " with " + tdata.
        GetNumObjects() + " objects");

    AttributeList alist = graph.FindAttributes(type);
    AttributeListIterator alistIt = alist.Iterator();
    while (alistIt.HasNext())
    {
        int attr = alistIt.Next();
        Attribute adata = graph.GetAttribute(attr);
        System.Console.WriteLine(" - Attribute " + adata.GetName());

        graph.RemoveAttribute(attr);
    }

    graph.RemoveType(type);
}
```

C++

```
Graph * graph = sess->GetGraph();
...
type_t peopleTypeId = graph->FindType(L"people");
if (InvalidType == peopleTypeId)
{
    peopleTypeId = graph->NewNodeType(L"people");
}
type_t friendTypeId = graph->FindType(L"friend");
if (InvalidType == friendTypeId)
{
    friendTypeId = graph->NewEdgeType(L"friend", true, true);
}
...
TypeList * tlist = graph->FindEdgeTypes();
TypeListIterator * tlistIt = tlist->Iterator();
while (tlistIt->HasNext())
{
    type_t type = tlistIt->Next();
    Type * tdata = graph->GetType(type);
    std::wcout << L"Type " << tdata->GetName() << L" with " << tdata->
        GetNumObjects() << L" objects";

    AttributeList * alist = graph->FindAttributes(type);
    AttributeListIterator * alistIt = alist->Iterator();
    while (alistIt->HasNext())
    {
        attr_t attr = alistIt->Next();
        Attribute * adata = graph->GetAttribute(attr);
        std::wcout << L" - Attribute " << adata->GetName();

        delete adata;
        graph->RemoveAttribute(attr);
    }
    delete alist;
    delete alistIt;

    delete tdata;
    graph->RemoveType(type);
}
```

```
delete tlist;
delete tlistIt;
```

Python

```
graph = sess.get_graph()
...
people_type_id = graph.find_type(u"PEOPLE")
if people_type_id == sparksee.Type.INVALID_TYPE:
    people_type_id = graph.new_node_type(u"PEOPLE")

friend_type_id = graph.find_type(u"FRIEND")
if friend_type_id == sparksee.Type.INVALID_TYPE:
    friend_type_id = graph.new_edge_type(u"FRIEND", True, True)
...
type_list = graph.find_edge_types()
for my_type in type_list:
    type_data = graph.get_type(my_type)
    print "Type", type_data.get_name(), " with ", type_data.get_num_objects(), "
        objects"

    attribute_list = graph.find_attributes(my_type)
    for attribute in attribute_list:
        attribute_data = graph.get_attribute(attribute)
        print " - Attribute ", attribute_data.get_name()
        graph.remove_attribute(attribute)

    graph.remove_type(my_type)
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
int peopleTypeId = [graph findType: @"people"];
if ([STSType getInvalidType] == peopleTypeId)
{
    peopleTypeId = [graph createNodeType: @"people"];
}

int friendTypeId = [graph findType: @"friend"];
if ([STSType getInvalidType] == friendTypeId)
{
    friendTypeId = [graph createEdgeType: @"friend" directed: TRUE neighbors: TRUE
        ];
}
...
STSTypeList * tlist = [graph findEdgeTypes];
STSTypeListIterator * tlistIt = [tlist iterator];
while ([tlistIt hasNext])
{
    int type = [tlistIt next];
    STSType * tdata = [graph getType: type];
    NSLog(@"Type %@ with %lld\n", [tdata getName], [tdata getNumObjects]);

    STSAttributeList * alist = [graph findAttributes: type];
    STSAttributeListIterator * alistIt = [alist iterator];
    while ([alistIt hasNext])
    {
        int attr = [alistIt next];
        STSAttribute * adata = [graph getAttribute: attr];
        NSLog(@" - Attribute %@\n", [adata getName]);
        [graph removeAttribute: attr];
    }

    [graph removeType: type];
}
```

Attributes and values

All node and edge types can have a set of attributes which are part of the schema of the graph database. An attribute should have a single value, and once created it is possible to set and get that value multiple times for each object.

Definition

In general, attributes are defined within the scope of a node or edge type and identified by a unique string identifier provided by the user. As is explained in the 'Attributes' section of the 'Graph database' chapter, it is possible to have two attributes with the same name but defined for two different types. For example, we could define the attribute "Name" for two different node types "PEOPLE" and "MOVIE", resulting in two different attributes. Since attributes are defined within the scope of a type, only objects belonging to that type will be able to set and get values for that attribute.

Figure 4.0: Attributes

The method to define a new attribute is Graph#newAttribute. As well as the parent type and the name, the definition of an attribute includes the datatype and the index-capabilities.

The datatype restricts the domain of the values for that attribute, thus all the objects having a non-null value for the attribute will have a value belonging to that domain. For example, in the previous example, all the objects having a value for the "Name" attribute will belong to the String datatype domain. Existing datatypes are defined in the Datatype enum class which includes the following:

- **Boolean** TRUE or FALSE values.
- **Integer** 32-bit signed integer values.
- **Long** 64-bit signed integer values.
- **Double** 64-bit signed double values.

- **Timestamp** Distance from Epoch (UTC) with millisecond precision. Valid timestamps must be in the range ['1970-01-01 00:00:01' UTC, '2038-01-19 03:14:07' UTC].
- **String** Unicode string values. Maximum length is restricted to 2048 characters.
- **Text** Large unicode character object. See 'Text attributes' section in this chapter.
- **OID** Sparksee object identifier values.

The indexing-capabilities of an attribute determine the performance of the query operations on that attribute as is explained in the 'Indexing' section of the 'Graph Database' chapter. Different index options are defined in the AttributeKind enum class which includes basic, indexed and unique attributes.

The indexing-capabilities of an attribute can be updated later using the Graph #indexAttribute method.

When an object is created it will have the default value for all the attributes defined for the type of the new object (by default the attribute value is null). This default value can also be defined afterwards by using the method Graph# setAttributeDefaultValue.

Analogously to node and edge types, when an attribute is defined a unique and immutable Sparksee-generated numeric identifier is returned. This identifier is the *attribute identifier* and will be used for all those APIs requiring an attribute as a parameter, like the Graph#setAttribute method which sets a value for a given attribute and object (explained in next section).

Node or Edge specific, Global and Session attributes Although the most common type of attributes are those whose scope is defined as **type specific** - Objects (node or edge objects) belonging to that type are the only ones allowed to set and get values for that attribute - we can also create attributes with a wider scope.

It is also possible to define the scope of an attribute as **node specific** using Type#NodesType as the parent type identifier argument when calling the Graph# newAttribute method. Whereas regular attributes are restricted to those objects belonging to the specified parent type, node attributes may be used for any node type in the graph. Therefore, any node of the graph, no matter which node type it belongs to, can set and get the values of a node attribute.

Analogous to the former kind is the user may define the scope of an attribute as **edge specific** using Type#EdgesType as the parent type identifier argument when calling the Graph#newAttribute method. Whereas regular attributes are restricted to those objects belonging to the specified parent type, node attributes may be used for any edge type in the graph. Therefore, any edge of the graph, no matter which edge type it belongs to, can set and get the values of a node attribute.

Finally, it is also possible to define the scope of an attribute as **global** using Type#GlobalType as the parent type identifier argument when calling the Graph #newAttribute method. Global attributes may be used for any object (node or

edge) in the graph. Therefore, any object of the graph, no matter which type it belongs to, can set and get the values of a global attribute.

Whereas attributes and their values are persistent in the graph database (as types or objects), it is possible to create temporary or **session** attributes. Session attributes are a special type of attribute that are exclusively associated to the session:

- They can only be seen and used within the session. Therefore other concurrent sessions cannot set, get or perform any operation with that attribute.
- They are temporary and are automatically removed when the session is closed.

As they cannot be accessed outside the scope of a session or after it finishes, they are anonymous and do not require a name. Session attributes are created using the Graph#newSessionAttribute which returns the attribute identifier. Despite the restrictions, these attributes can be used in any method just like any other attribute.

Use

The class Value is used to set and get attribute values for an object. This class is just the container for a specific value belonging to a specific data type (the domain). This class helps create a more simplified API. Thus, instead of having a different set/get method for each data type (for example: setAttributeInteger , setAttributeString, and so on), the user only manages two methods: one for setting the value and another one for getting it. As it is only a container, the value can be reused as many times as necessary. Actually, it is highly recommended that a Value instance is created only once and then different values set as many times as required. This behavior is illustrated in the following examples:

Java

```
Graph graph = sess.getGraph();
...
int nameAttrId = graph.findAttribute(peopleTypeId, "Name");
if (Attribute.InvalidAttribute == nameAttrId)
{
    nameAttrId = graph.newAttribute(peopleTypeId, "Name", DataType.String,
        AttributeKind.Indexed);
}
long people1 = graph.newNode(peopleTypeId);
long people2 = graph.newNode(peopleTypeId);
Value v = new Value();
graph.setAttribute(people1, nameAttrId, v.setString("Scarlett Johansson"));
graph.setAttribute(people2, nameAttrId, v.setString("Woody Allen"));
```

[C#]

```
Graph graph = sess.GetGraph();
...
int nameAttrId = graph.FindAttribute(peopleTypeId, "Name");
if (Attribute.InvalidAttribute == nameAttrId)
{
    nameAttrId = graph.NewAttribute(peopleTypeId, "Name", DataType.String,
        AttributeKind.Indexed);
}
long people1 = graph.NewNode(peopleTypeId);
long people2 = graph.NewNode(peopleTypeId);
Value v = new Value();
graph.SetAttribute(people1, nameAttrId, v.SetString("Scarlett Johansson"));
graph.SetAttribute(people2, nameAttrId, v.SetString("Woody Allen"));
```

C++

```
Graph * graph = sess->GetGraph();
...
attr_t nameAttrId = graph->FindAttribute(peopleTypeId, L"Name");
if (InvalidAttribute == nameAttrId)
{
    nameAttrId = graph->NewAttribute(peopleTypeId, L"Name", String, Indexed);
}
oid_t people1 = graph->NewNode(peopleTypeId);
oid_t people2 = graph->NewNode(peopleTypeId);
Value v;
graph->SetAttribute(people1, nameAttrId, v.SetString(L"Scarlett Johansson"));
graph->SetAttribute(people2, nameAttrId, v.SetString(L"Woody Allen"));
```

Python

```
graph = sess.get_graph()
...
name_attr_id = graph.find_attribute(people_type_id, u"Name")
if sparksee.Attribute.INVALID_ATTRIBUTE == name_attr_id:
    name_attr_id = graph.new_attribute(people_type_id, u"Name", sparksee.DataType.
        STRING, sparksee.AttributeKind.INDEXED)

people1 = graph.new_node(people_type_id)
people2 = graph.new_node(people_type_id)
v = sparksee.Value()
graph.set_attribute(people1, name_attr_id, v.set_string(u"Scarlett Johansson"))
graph.set_attribute(people2, name_attr_id, v.set_string(u"Woody Allen"))
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
int nameAttrId = [graph findAttribute: peopleTypeId name: @"name"];
if ([STSAttribute getInvalidAttribute] == nameAttrId)
{
    nameAttrId = [graph createAttribute: peopleTypeId name: @"name" dt: STSString
        kind: STSIndexed];
}
long long people1 = [graph createNode: peopleTypeId];
long long people2 = [graph createNode: peopleTypeId];
STSValue *v = [[STSValue alloc] init];
[graph setAttribute: people1 attr: nameAttrId value: [v setString: @"Scarlett
    Johansson"]];
[graph setAttribute: people2 attr: nameAttrId value: [v setString: @"Woody Allen"
    ]];
```

Additionally to Graph#setAttribute and Graph#getAttribute, there are other methods to manage attributes:

- Graph#removeAttribute removes the attribute and its values.
- AttributeList Graph#findAttributes(int type) retrieves all attributes defined for the given type.
- AttributeList Graph#getAttributes(long object) retrieves all the attributes for the given object which have a non-null value.
- AttributeStatistics Graph#getAttributeStatistics(long attr) and long Graph#getAttributeIntervalCount retrieve some statistics for the given attribute (this is explained more in detail in the 'Statistics' section of the 'Maintenance and monitoring' chapter).
- Graph#indexAttribute updates the index-capability of the given attribute.
- Attribute Graph#getAttribute retrieves all the metadata associated with an existing attribute.

Text attributes String attribute values are restricted to a maximum length of 2048 characters, thus in case of storing larger strings, text attributes should be used instead. However, it is important to notice that whereas string attributes are set and got using the Value class, text attributes are operated using a stream pattern.

Text attributes can be considered a special case for the following reasons:

- They cannot be indexed.
- They cannot be used in query operations, which are explained in the 'Query operations' section of this chapter. They only can get and set the value for a given object identifier.
- Graph#getAttributeText gets the value for a given object identifier by means of a TextStream instance instead of the Graph#getAttribute method and a Value instance.
- Graph#setAttributeText sets the value for a given object identifier by means of a TextStream instance instead of the Graph#setAttribute method and a Value instance.

The TextStream class implements the stream pattern for Sparksee text attributes as follows:

1. It retrieves a read/write/append stream for a text attribute and object identifier.
2. It reads or writes chunks of data on the stream as many times as required.
3. It closes the stream to ensure all data is correctly flushed.

The following examples show how to set a text attribute:

Java

```
Graph graph = sess.getGraph();
...
long oid = ...              // object identifier
int textAttrId = ...       // text attribute identifier
...
String str1 = "This is the first chunk of the text stream";
String str2 = "This is the second chunk of the text stream";
...
TextStream tstrm = new TextStream(false);
graph.setAttributeText(oid, textAttrId, tstrm);
...
char[] buff = str1.toCharArray();
tstrm.write(buff, buff.length);
buff = str2.toCharArray();
tstrm.write(buff, buff.length);
tstrm.close();
```

[C#]

```
Graph graph = sess.GetGraph();
...
long oid = ...              // object identifier
int textAttrId = ...       // text attribute identifier
...
string str1 = "This is the first chunk of the text stream";
string str2 = "This is the second chunk of the text stream";
...
TextStream tstrm = new TextStream(false);
graph.SetAttributeText(oid, textAttrId, tstrm);
...
char[] buff = str1.ToCharArray();
tstrm.Write(buff, buff.Length);
buff = str2.ToCharArray();
tstrm.Write(buff, buff.Length);
tstrm.Close();
```

C++

```
Graph * graph = sess->GetGraph();
...
oid_t oid = ...             // object identifier
attr_t textAttrId = ...     // text attribute identifier
...
std::wstring str1(L"This is the first chunk of the text stream");
std::wstring str2(L"This is the second chunk of the text stream");
...
TextStream tstrm(false);
graph->SetAttributeText(oid, textAttrId, &tstrm);
...
tstrm.Write(str1.c_str(), str1.size());
tstrm.Write(str2.c_str(), str2.size());
tstrm.Close();
```

Python

```
graph = sess.get_graph()
...
oid = ...                # object identifier
text_attribute_id = ...  # text attribute identifier
...
str1 = u"This is the first chunk of the text stream"
str2 = u"This is the second chunk of the text stream"
...
```

```
tstrm = sparksee.TextStream(False)
graph.set_attribute_text(oid, text_attribute_id, tstrm)
...
tstrm.write(str1, len(str1))
tstrm.write(str2, len(str2))
tstrm.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
long long oid = ...; // object identifier
int textAttrId = ...;   // text attribute identifier
...
NSString * const str1 = @"This is the first chunk of the text stream";
NSString * const str2 = @"This is the second chunk of the text stream";
...
STSTextStream *tstrm = [[STSTextStream alloc] initWithAppend: false];
[graph setAttributeText: oid attr: textAttrId tstream: tstrm];
...
[tstrm writeString: str1];
[tstrm writeString: str2];
[tstrm close]; // The stream must be closed
//[tstrm release]; // You may need to release it here.
```

The following code blocks show an example of how the previously written text could be retrieved:

It's important to always close the TextStream object retrieved, even if it's content is Null.

Java

```
Graph graph = sess.getGraph();
...
long oid = ...              // object identifier
int textAttrId = ...       // text attribute identifier
...
TextStream tstrm = graph.getAttributeText(oid, textAttrId);
if (!tstrm.isNull())
{
    int read;
    StringBuffer str = new StringBuffer();
    do
    {
        char[] buff = new char[10];
        read = tstrm.read(buff, 10);
        str.append(buff, 0, read);
    }
    while (read > 0);
    System.out.println(str);
}
tstrm.close();
```

[C#]

```
Graph graph = sess.GetGraph();
...
long oid = ...              // object identifier
int textAttrId = ...       // text attribute identifier
...
TextStream tstrm = graph.GetAttributeText(oid, textAttrId);
```

```
if (!tstrm.IsNull())
{
    int read;
    System.Text.StringBuilder str = new System.Text.StringBuilder();
    do
    {
        char[] buff = new char[10];
        read = tstrm.Read(buff, 10);
        str.Append(buff, 0, read);
    }
    while (read > 0);
    System.Console.WriteLine(str);
}
tstrm.Close();
```

C++

```
Graph * graph = sess->GetGraph();
...
oid_t oid = ...             // object identifier
attr_t textAttrId = ...     // text attribute identifier
...
TextStream *tstrm = graph->GetAttributeText(oid, textAttrId);
if (!tstrm->IsNull())
{
    int read;
    std::wstring str;
    do
    {
        wchar_t * buff = new wchar_t[10];
        read = tstrm->Read(buff, 10);
        str.append(buff, read);
    }
    while (read > 0);
    std::wcout << str << std::endl;
}
tstrm->Close();
delete tstrm;
```

Python

```
Graph * graph = sess->GetGraph();
...
oid = ...                # object identifier
text_attribute_id = ...  # text attribute identifier
...
tstrm = graph.get_attribute_text(oid, text_attribute_id)
if not tstrm.is_null():
    readed_character = tstrm.read(1)
    readedStr = readed_character
    while len(readed_character) > 0:
        readed_character = tstrm.read(10)
        readedStr = readedStr + readed_character
    print readedStr
tstrm.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
long long oid = ...; // object identifier
int textAttrId = ...;   // text attribute identifier
...
STSTextStream *tstrm = [graph getAttributeText: oid attr: textAttrId];
```

```
if (![tstrm IsNull])
{
    int readedSize;
    NSMutableString * str = [[NSMutableString alloc] init];
    do
    {
        NSString *next10chars = [tstrm readString: 10];
        [str appendString: next10chars];
        readedSize = [next10chars length];
    }
    while (readedSize > 0);
    NSLog(@"Readed text:\n%@\n", str);
}
[tstrm close];
//[tstrm release];
```

Objects

As we are going to see later, most of the query or navigational operations return a collection of object identifiers or OIDs as the result of the operation. The Objects class is used for the management of these collections of object identifiers. Actually, Objects is considered a set, as duplicated elements are not allowed and it does not follow a defined order.

The user can create as many Objects instances as may be required for use by calling Objects Session#newObjects. Please note that the Objects class has been designed to store a large collection of object identifiers. Therefore, for smaller collections it is strongly recommended that a common class provided by the chosen language is used.

Also, it is important to note that this class is not thread-safe, so it cannot be used by two different threads at the same time.

The Objects class can add object identifiers to a collection (Objects#add), check if an object identifier exists (Objects#exists), remove an object identifier from the set (Objects#remove) or retrieve the number of elements of a collection (Objects #count). Take a look at the Objects class reference documentation to get a comprehensive list of all the available methods.

The following code blocks show some examples of the use of the Objects class:

Java

```
Objects objs = sess.newObjects();
assert objs.add(1) && objs.add(2);
assert !objs.add(1);
assert objs.exists(1) && objs.exists(2) && !objs.exists(3);
assert objs.count() == 2 && !objs.isEmpty();
...
objs.close();
```

[C#]

```
Objects objs = sess.NewObjects();
System.Diagnostics.Debug.Assert(objs.Add(1) && objs.Add(2));
System.Diagnostics.Debug.Assert(!objs.Add(1));
```

```
System.Diagnostics.Debug.Assert(objs.Exists(1) && objs.Exists(2) && !objs.Exists
    (3));
System.Diagnostics.Debug.Assert(objs.Count() == 2);
...
objs.Close();
```

C++

```
Objects * objs = sess->NewObjects();
assert(objs->Add(1) && objs->Add(2));
assert(!objs->Add(1));
assert(objs->Exists(1) && objs->Exists(2) && !objs->Exists(3));
assert(objs->Count() == 2);
...
delete objs;
```

Python

```
objs = sess.new_objects()
assert(objs.add(1) and objs.add(2))
assert(not objs.add(1))
assert(objs.exists(1) and objs.exists(2) and not objs.exists(3))
assert(objs.count() == 2)
...
objs.close()
```

Objective-C

```
STSObjects * objs = [sess createObjects];
assert([objs add: 1] && [objs add: 2]);
assert(![objs add: 1]);
assert([objs exists: 1] && [objs exists: 2] && ![objs exists: 3]);
assert([objs count] == 2);
[objs close];
```

As seen in the previous examples, all Objects instances need to be closed (deleted in the case of C++) just like Sparksee, Database and Session instances. Moreover, collections must be closed as soon as possible to free internal resources and ensure a higher performance of the application.

Furthermore, as these collections are retrieved from the Session they are only valid while the parent Session instance remains open. In fact, when the Session is closed (deleted in the case of C++), it checks out if there are still any non-closed Objects instances, and if detected an exception is thrown.

Objects Iterator

The ObjectsIterator class is used to construct an *iterator* instance for traversing a collection. The traversal can be performed by calling the Objects#hastNext and Objects#next methods:

Java

```
Objects objs = sess.newObjects();
...
ObjectsIterator it = objs.iterator();
while (it.hasNext())
{
    long currentOID = it.next();
}
it.close();
...
objs.close();
```

[C#]

```
Objects objs = sess.NewObjects();
...
ObjectsIterator it = objs.Iterator();
while (it.HasNext())
{
    long currentOID = it.Next();
}
it.Close();
...
objs.Close();
```

C++

```
Objects * objs = sess->NewObjects();
...
ObjectsIterator * it = objs->Iterator();
while (it->HasNext())
{
    oid_t currentOID = it->Next();
}
delete it;
...
delete objs;
```

Python

```
objs = sess.new_objects()
...
for oid in objs:
    print oid
...
objs.close()
```

Objective-C

```
STSObjects * objs = [sess createObjects];
...
STSObjectsIterator * it = [objs iterator];
while ([it hasNext])
{
    long long currentOID = [it next];
}
[it close];
...
[objs close];
```

It is important to notice that ObjectsIterator instances must be closed (deleted in the case of C++) as soon as possible to ensure better performance. Nevertheless, non-closed iterators will be automatically closed when the collection is closed.

When traversing Objects instances it is important to have in mind that the Objects instance cannot be updated; elements cannot be added or removed from the collection.

Combination

Objects instances can be efficiently combined with the following methods. Note that there are two versions for the same call, because whereas the instance method performs the resulting operation on the calling instance, the static method creates a new instance as a result of the operation.

- Union.

  Objects#union(Objects objs) & static Objects Objects#combineUnion(Objects objs1, Objects objs2)

  The resulting collection contains the union of both collections with no repeated objects. It contains all the object identifiers from the first collection together with all the object identifiers from the second one.

- Intersection.

  Objects#intersection(Objects objs) & static Objects Objects#combineIntersection (Objects objs1, Objects objs2)

  The resulting collection contains the intersection of both collections. It contains only those object identifiers from the first collection that also belong to the second one.

- Difference.

  Objects#difference(Objects objs) & static Objects Objects#combineDifference (Objects objs1, Objects objs2)

  The resulting collection contains the difference of both collections. It contains those object identifiers from the first collection that do not belong to the second one.

Blow are some examples of the use of the three combination methods:

Java

```
Objects objsA = sess.newObjects();
objsA.add(1);
objsA.add(2);
Objects objsB = sess.newObjects();
objsB.add(2);
objsB.add(3);

Objects union = Objects.combineUnion(objsA, objsB);
assert union.exists(1) && union.exists(2) && union.exists(3) && union.count() ==
     3;
union.close();

Objects intersec = Objects.combineIntersection(objsA, objsB);
```

```
assert intersec.exists(2) && intersec.count() == 1;
intersec.close();

Objects diff = Objects.combineDifference(objsA, objsB);
assert diff.exists(1) && diff.count() == 1;
diff.close();

objsA.close();
objsB.close();
```

[C#]

```
Objects objsA = sess.NewObjects();
objsA.Add(1);
objsA.Add(2);
Objects objsB = sess.NewObjects();
objsB.Add(2);
objsB.Add(3);

Objects union = Objects.CombineUnion(objsA, objsB);
System.Diagnostics.Debug.Assert(union.Exists(1) && union.Exists(2) && union.Exists
    (3) && union.Count() == 3);
union.Close();

Objects intersec = Objects.CombineIntersection(objsA, objsB);
System.Diagnostics.Debug.Assert(intersec.Exists(2) && intersec.Count() == 1);
intersec.Close();

Objects diff = Objects.CombineDifference(objsA, objsB);
System.Diagnostics.Debug.Assert(diff.Exists(1) && diff.Count() == 1);
diff.Close();

objsA.Close();
objsB.Close();
```

C++

```
Objects * objsA = sess->NewObjects();
objsA->Add(1);
objsA->Add(2);
Objects * objsB = sess->NewObjects();
objsB->Add(2);
objsB->Add(3);

Objects * unnion = Objects::CombineUnion(objsA, objsB);
assert(unnion->Exists(1) && unnion->Exists(2) && unnion->Exists(3) && unnion->
    Count() == 3);
delete unnion;

Objects * intersec = Objects::CombineIntersection(objsA, objsB);
assert(intersec->Exists(2) && intersec->Count() == 1);
delete intersec;

Objects * diff = Objects::CombineDifference(objsA, objsB);
assert(diff->Exists(1) && diff->Count() == 1);
delete diff;

delete objsA;
delete objsB;
```

Python

```
objsA = sess.new_objects()
objsA.add(1)
```

```
objsA.add(2)
objsB = sess.new_objects()
objsB.add(2)
objsB.add(3)

union = sparksee.Objects.combine_union(objsA, objsB)
assert(union.exists(1) and union.exists(2) and union.exists(3) and union.count()
    == 3)
union.close()

intersec = sparksee.Objects.combine_intersection(objsA, objsB)
assert(intersec.exists(2) and intersec.count() == 1)
intersec.close()

diff = sparksee.Objects.combine_difference(objsA, objsB)
assert(diff.exists(1) and diff.count() == 1)
diff.close()

objsA.close()
objsB.close()
```

Objective-C

```
STSObjects * objsA = [sess createObjects];
[objsA add: 1];
[objsA add: 2];
STSObjects * objsB = [sess createObjects];
[objsB add: 2];
[objsB add: 3];

STSObjects * unnion = [STSObjects combineUnion: objsA objs2: objsB];
assert([unnion exists: 1] && [unnion exists: 2] && [unnion exists: 3] && [unnion
    count] == 3);
[unnion close];

STSObjects * intersec = [STSObjects combineIntersection: objsA objs2: objsB];
assert([intersec exists: 2] && [intersec count] == 1);
[intersec close];

STSObjects * diff = [STSObjects combineDifference: objsA objs2: objsB];
assert([diff exists: 1] && [diff count] == 1);
[diff close];

[objsA close];
[objsB close];
```

Query operations

Sparksee has different methods to retrieve data from the graph. Most of them return an instance of the Objects class.

The most simple query method is the select operation Objects Graph#select(int type), which retrieves all the objects belonging to the given node or edge type, so it is a type-based operation.

The method Objects Graph#select(int attribute, Condition cond, Value v) is more specific than the previous selection, being able to retrieve all the objects satisfying a condition for a given attribute. This select is an attribute-based operation.

Note that the second select operation requires the datatype of the given value to be the same as the datatype of the attribute.

This is the list of possible conditions to be specified in a select operation, all of them defined in the Condition enum class:

- Equal: retrieves those objects that have a value for the attribute equal to the given value.
- NotEqual: retrieves those objects that have a value for the attribute not equal to the given value.
- GreaterEqual: retrieves those objects that have a value for the attribute greater than or equal to the given value.
- GreaterThan: retrieves those objects that have a value for the attribute strictly greater than the given value.
- LessEqual: retrieves those objects that have a value for the attribute less than or equal to the given value.
- LessThan: retrieves those objects that have a value for the attribute strictly less than the given value.
- Between: retrieves those objects that have a value for the attribute within the range defined for the two given values. This condition can only be used in the specific signature of the selected method where there are two argument values instead of one: Objects Graph#select(int attribute, Condition cond, Value lower, Value higher). Note that lower and higher define an inclusive range [lower,higher], also null values cannot be used in the specification of the range.

There are a few further conditions which can only be used for string attributes:

- Like: retrieves those objects having a value which is a substring of the given value.
- LikeNoCase: retrieves those objects having a value which is a substring (not case sensitive) of the given value.
- RegExp: retrieves those objects having a value for the given attribute that matches a regular expression defined by the given string value. In fact, Like and LikeNoCase are simply two specific cases of regular expressions. More details on how Sparksee regular expressions work are given in a later section.

The following code blocks are examples of the select queries. Notice the use of the conditions:

Java

```
Graph graph = sess.getGraph();
Value v = new Value();
...
// retrieve all 'people' node objects
Objects peopleObjs1 = graph.select(peopleTypeId);
...
// retrieve Scarlett Johansson from the graph, which is a "PEOPLE" node
```

```
Objects peopleObjs2 = graph.select(nameAttrId, Condition.Equal, v.setString("
    Scarlett Johansson"));
...
// retrieve all 'PEOPLE' node objects having "Allen" in the name. It would
    retrieve
// Woody Allen, Tim Allen or Allen Leech, or other similar if they are present in
    the graph.
Objects peopleObjs3 = graph.select(nameAttrId, Condition.Like, v.setString("Allen"
    ));
...
peopleObjs1.close();
peopleObjs2.close();
peopleObjs3.close();
```

[C#]

```
Graph graph = sess.GetGraph();
Value v = new Value();
...
// retrieve all 'people' node objects
Objects peopleObjs1 = graph.Select(peopleTypeId);
...
// retrieve Scarlett Johansson from the graph, which is a "PEOPLE" node
Objects peopleObjs2 = graph.Select(nameAttrId, Condition.Equal, v.SetString("
    Scarlett Johansson"));
...
// retrieve all 'PEOPLE' node objects having "Allen" in the name. It would
    retrieve
// Woody Allen, Tim Allen or Allen Leech, or other similar if they are present in
    the graph.
Objects peopleObjs3 = graph.Select(nameAttrId, Condition.Like, v.SetString("Allen"
    ));
...
peopleObjs1.Close();
peopleObjs2.Close();
peopleObjs3.Close();
```

C++

```
Graph * graph = sess->GetGraph();
Value v;
...
// retrieve all 'people' node objects
Objects * peopleObjs1 = graph->Select(peopleTypeId);
...
// retrieve Scarlett Johansson from the graph, which is a "PEOPLE" node
Objects * peopleObjs2 = graph->Select(nameAttrId, Equal, v.SetString(L"Scarlett
    Johansson"));
...
// retrieve all 'PEOPLE' node objects having "Allen" in the name. It would
    retrieve
// Woody Allen, Tim Allen or Allen Leech, or other similar if they are present in
    the graph.
Objects * peopleObjs3 = graph->Select(nameAttrId, Like, v.SetString(L"Allen"));
...
delete peopleObjs1;
delete peopleObjs2;
delete peopleObjs3;
```

Python

```
graph = sess.get_graph()
v = sparksee.Value()
...
```

```
# retrieve all 'people' node objects
people_objs1 = graph.select(people_type_id)
...
# retrieve Scarlett Johansson from the graph, which is a "PEOPLE" node
people_objs2 = graph.select(name_attr_id, sparksee.Condition.EQUAL, v.set_string(u
    "Scarlett Johansson"))
...
# retrieve all 'PEOPLE' node objects having "Allen" in the name. It would retrieve
# Woody Allen, Tim Allen or Allen Leech, or other similar if they are present in
    the graph.
people_objs3 = graph.select(name_attr_id, sparksee.Condition.LIKE, v.set_string("
    Allen"))
...
people_objs1.close()
people_objs2.close()
people_objs3.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
STSValue *v = [[STSValue alloc] init];
...
// retrieve all 'people' node objects
STSObjects * peopleObjs1 = [graph selectWithType: peopleTypeId];
...
// retrieve Scarlett Johansson from the graph, which is a "PEOPLE" node
STSObjects * peopleObjs2 = [graph selectWithAttrValue: nameAttrId cond: STSEqual
    value: [v setString: @"Scarlett Johansson"]];
...
// retrieve all 'PEOPLE' node objects having "Allen" in the name. It would
    retrieve
// Woody Allen, Tim Allen or Allen Leech, or other similar if they are present in
    the graph.
STSObjects * peopleObjs3 = [graph selectWithAttrValue: nameAttrId cond: STSLike
    value: [v setString: @"Allen"]];
...
[peopleObjs1 close];
[peopleObjs2 close];
[peopleObjs3 close];
```

The method long Graph#findObject(long attr, Value v) is a special case of the attribute-based select operation. In this case, instead of returning a collection of objects, this method returns a single object identifier. Moreover, in this case, findObject assumes the condition Equal. Thus, it randomly returns the object identifier of any of the objects having the given value for the given attribute. Although it can be used with any kind of attribute (Basic, Indexed, Unique) it may be better to use it with Unique attributes, as they ensure that two objects will not have the same attribute value (except for the null value).

When retrieving an Objects from the graph and traversing the collection, as it is not a copy but directly access to Sparksee internal structures, any object cannot be removed or added from that collection. For example, if we traverse a collection of objects belonging to a certain type, we cannot remove elements from that type at the same time.

Regular expressions

As explained previously, a regular expression can be used in query operations to search for objects having a value for an attribute matching the given regular

expression. RegExp is the Condition value to set a matching regular expression condition in an attribute-based select operation. Of course, this condition can only be applied for string attributes.

Regular expression format conforms to most of the POSIX Extended Regular Expressions and therefore they are case sensitive.

A detailed reference for the syntax of the regular expressions can be found in the POSIX Extended Regular Expressions reference guide.

For instance the following regular expressions will have these matches:

- A+B*C+ matches AAABBBCCCD
- B*C+ matches AAACCCD
- B+C+ does not match AAACCCD
- ^A[^]*D$ matches AAACCCD
- B*C+$ does not match AAACCCD

The following examples search for people with names that start with an 'A' and end with a 'D', using regular expressions:

Java

```
Graph graph = sess.getGraph();
Value v = new Value();
...
// retrieve all 'people' node objects having a value for the 'name' attribute
// satisfying the the '^A[^]*D$' regular expression
Objects peopleObjs = graph.select(nameAttrId, Condition.RegExp, v.setString("^A
    [^]*D$"));
...
peopleObjs.close();
```

[C#]

```
Graph graph = sess.GetGraph();
Value v = new Value();
...
// retrieve all 'people' node objects having a value for the 'name' attribute
// satisfying the the '^A[^]*D$' regular expression
Objects peopleObjs = graph.Select(nameAttrId, Condition.RegExp, v.SetString("^A
    [^]*D$"));
...
peopleObjs.Close();
```

C++

```
Graph * graph = sess->GetGraph();
Value v;
...
// retrieve all 'people' node objects having a value for the 'name' attribute
// satisfying the the '^A[^]*D$' regular expression
Objects * peopleObjs = graph->Select(nameAttrId, RegExp, v.SetString(L"^A[^]*D$"))
    ;
...
delete peopleObjs;
```

Python

```
graph = sess.get_graph()
v = sparksee.Value()
...
# retrieve all 'people' node objects having a value for the 'name' attribute
# satisfying the the '^A[^]*D$' regular expression
people_objs = graph.select(name_attr_id, sparksee.Condition.REG_EXP, v.set_string(
     "^A[^]*D$"))
...
people_objs.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
STSValue *v = [[STSValue alloc] init];
...
// retrieve all 'people' node objects having a value for the 'name' attribute
// satisfying the the '^A[^]*D$' regular expression
STSObjects * peopleObjs = [graph selectWithAttrValue: nameAttrId cond: STSRegExp
     value: [v setString: @"^A[^]*D$"]];
...
[peopleObjs close];
```

Navigation operations

There are two basic set of methods for the navigation through the graph:

- Explode-based methods: these methods visit the edges of a given node identifier. Check out Graph#explode in the reference documentation for details.
- Neighbor-based: these methods visit the neighbor nodes of a given node identifier. A neighbor is an adjacent node 1-hop away from the source node (distance 1). Check out Graph#neighbor in the reference documentation for details.

Both methods require the source node identifier, the edge type and the direction of navigation. The edge type restricts the edge instances to be considered for the navigation, so edge instances that do not belong to the given edge type will be ignored for the operation. And the direction restricts the direction of navigation through those edges. The EdgesDirection enum class defines the following directions:

- Outgoing: Only out-going edges, starting from the source node, will be valid for the navigation.
- Ingoing: Only in-going edges will be valid for the navigation.
- Any: Both out-going and in-going edges will be valid for the navigation.

Note that in case of undirected edges the direction restriction has no effect as we may consider undirected edges as bidirectional edges, which do not have a restriction on the direction of the navigation.

Although both methods return an Objects instance as a result, for *explode* the resulting Objects instance contains edge identifiers whereas for *neighbors* it contains node identifiers.

The following examples show the navigational methods in use, assuming a database where PEOPLE nodes are related by means of undirected FRIEND edges and directed LOVES edges:

Java

```
Graph graph = sess.getGraph();
...
long node = ... // a PEOPLE node has been retrieved somehow
int friendTypeId = graph.findType("FRIEND");
int lovesTypeId = graph.findType("LOVES");
...
// retrieve all in-comming LOVES edges
Objects edges = graph.explode(node, lovesTypeId, EdgesDirection.Ingoing);
...
// retrieve all nodes through FRIEND edges
Objects friends = graph.neighbors(node, friendTypeId, EdgesDirection.Any);
...
edges.close();
friends.close();
```

[C#]

```
Graph graph = sess.GetGraph();
...
long node = ... // a PEOPLE node has been retrieved somehow
int friendTypeId = graph.FindType("FRIEND");
int lovesTypeId = graph.FindType("LOVES");
...
// retrieve all in-comming LOVES edges
Objects edges = graph.Explode(node, lovesTypeId, EdgesDirection.Ingoing);
...
// retrieve all nodes through FRIEND edges
Objects friends = graph.Neighbors(node, friendTypeId, EdgesDirection.Any);
...
edges.Close();
friends.Close();
```

C++

```
Graph * graph = sess->GetGraph();
...
oid_t node = ... // a PEOPLE node has been retrieved somehow
type_t friendTypeId = graph->FindType(L"FRIEND");
type_t lovesTypeId = graph->FindType(L"LOVES");
...
// retrieve all in-comming LOVES edges
Objects * edges = graph->Explode(node, lovesTypeId, Ingoing);
...
// retrieve all nodes through FRIEND edges
Objects * friends = graph->Neighbors(node, friendTypeId, Any);
...
delete edges;
delete friends;
```

Python

```
graph = sess.get_graph()
...
node = ... # a PEOPLE node has been retrieved somehow
friend_type_id = graph.find_type(u"FRIEND")
loves_type_id = graph.find_type(u"LOVES")
...
# retrieve all in-comming LOVES edges
edges = graph.explode(node, loves_type_id, sparksee.EdgesDirection.INGOING)
...
# retrieve all nodes through FRIEND edges
friends = graph.neighbors(node, friend_type_id, sparksee.EdgesDirection.ANY)
...
edges.close()
friends.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
long long node = ...; // a PEOPLE node has been retrieved somehow
int friendTypeId = [graph findType: @"FRIEND"];
int lovesTypeId = [graph findType: @"LOVES"];
...
// retrieve all in-comming LOVES edges
STSObjects * edges = [graph explode: node etype: lovesTypeId dir: STSIngoing];
...
// retrieve all nodes through FRIEND edges
STSObjects * friends = [graph  neighbors: node etype: friendTypeId dir: STSAny];
...
[edges close];
[friends close];
```

Note that in the previous examples we have performed the *neighbors* call setting Any as the direction because FRIEND is an undirected relationship. In fact, any other direction (Ingoing or Outgoing) would have retrieved the same result.

For both navigation methods there is a more general implementation where instead of having a source node identifier as the first argument, there is an Objects instance. This Objects instance is the collection of object identifiers to perform the operation. The result will be an Objects instance with the union of all the results of performing the operation for each of the objects in the argument instance.

The following examples show how to use this alternative version of the *neighbors* method to perform a friend-of-a-friend query:

Java

```
Graph graph = sess.getGraph();
...
long node = ... // a PEOPLE node has been retrieved somehow
int friendTypeId = graph.findType("FRIEND");
...
// 1-hop friends
Objects friends = graph.neighbors(node, friendTypeId, EdgesDirection.Any);
// friends of friends (2-hop)
Objects friends2 = graph.neighbors(friends, friendTypeId, EdgesDirection.Any);
...
friends.close();
friends2.close();
```

[C#]

```
Graph graph = sess.GetGraph();
...
long node = ... // a PEOPLE node has been retrieved somehow
int friendTypeId = graph.FindType("FRIEND");
...
// 1-hop friends
Objects friends = graph.Neighbors(node, friendTypeId, EdgesDirection.Any);
// friends of friends (2-hop)
Objects friends2 = graph.Neighbors(friends, friendTypeId, EdgesDirection.Any);
...
friends.Close();
friends2.Close();
```

C++

```
Graph * graph = sess->GetGraph();
...
oid_t node = ... // a PEOPLE node has been retrieved somehow
type_t friendTypeId = graph->FindType(L"FRIEND");
...
// 1-hop friends
Objects * friends = graph->Neighbors(node, friendTypeId, Any);
// friends of friends (2-hop)
Objects * friends2 = graph->Neighbors(friends, friendTypeId, Any);
...
delete friends;
delete friends2;
```

Python

```
graph = sess.get_graph()
...
node = ... # a PEOPLE node has been retrieved somehow
friend_type_id = graph.find_type(u"FRIEND")
...
# 1-hop friends
friends = graph.neighbors(node, friend_type_id, sparksee.EdgesDirection.ANY)
# friends of friends (2-hop)
friends2 = graph.neighbors(friends, friend_type_id, sparksee.EdgesDirection.ANY)
...
friends.close()
friends2.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
long long node = ...; // a PEOPLE node has been retrieved somehow
int friendTypeId = [graph findType: @"FRIEND"];
...
// 1-hop friends
STSObjects * friends = [graph neighbors: node etype: friendTypeId dir: STSAny];
// friends of friends (2-hop)
STSObjects * friends2 = [graph neighborsWithObjects: friends etype: friendTypeId
     dir: STSAny];
...
[friends close];
[friends2 close];
```

Neighbor index

Actually, by default the *neighbors* method is solved internally by performing an explode-based implementation, firstly visiting the edges themselves and then visiting the *other side* of the edge. Additionally a specific index may be created to improve the performance of the *neighbors* query. More details are in the 'Indexing' section of the 'Graph database' chapter.

This index can be set when creating an edge type (Graph#newEdgeType). Using the index, all neighbors-based operations involving that edge type will be internally performed faster. As expected, the management of an index introduces a small penalty when creating new edge instances. Nevertheless, it is strongly recommended to set an index for those applications making intensive use or critical use of the *neighbors* method, as the small penalty is more than compensated for the improvement in performance.

Transactions

The 'Processing' section in the 'Graph database' chapter explains the execution model and Sparksee transactions in detail.

To make explicit use of transactions, the Session class provides three methods:

- begin - which starts a transaction.

  The transaction starts as a read transaction and will become a write transaction when:

  - the transaction executes a write operation on the graph
  - there are no other read or write transactions running

- beginUpdate - starts directly as a write transaction
- commit - which ends a transaction.
- rollback - which ends the transaction aborting all the changes enclosed in the transaction.

Take in to accunt that the rollback mechanism can be disabled to improve the performance when there is not any active transaction. The following example shows how to enable and disable the rollback.

Java

```
SparkseeConfig cfg = new SparkseeConfig();
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.create("HelloSparksee.gdb", "HelloSparksee");
db.disableRollback();  // Rollback is now disabled
Session sess = db.newSession();
Graph graph = sess.getGraph();
//Use 'graph' to perform operations on the graph database without rollbacks
db.enableRollback(); // Rollback is now enabled
//Use 'graph' to perform operations on the graph database
sess.close();
db.close();
sparksee.close();
```

[C#]

```
SparkseeConfig cfg = new SparkseeConfig();
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.Create("HelloSparksee.gdb", "HelloSparksee");
db.disableRollback();  // Rollback is now disabled
Session sess = db.NewSession();
Graph graph = sess.GetGraph();
//Use 'graph' to perform operations on the graph database without rollbacks
db.EnableRollback(); // Rollback is now enabled
//Use 'graph' to perform operations on the graph database
sess.Close();
db.Close();
sparksee.Close();
```

C++

```
SparkseeConfig cfg;
Sparksee *sparksee = new Sparksee(cfg);
Database * db = sparksee->Create(L"HelloSparksee.gdb", L"HelloSparksee");
db->DisableRollback();  // Rollback is now disabled
Session * sess = db->NewSession();
Graph * graph = sess->GetGraph();
//Use 'graph' to perform operations on the graph database without rollbacks
db->EnableRollback(); // Rollback is now enabled
//Use 'graph' to perform operations on the graph database
delete sess;
delete db;
delete sparksee;
return EXIT_SUCCESS;
```

Python

```
cfg = sparksee.SparkseeConfig()
sparks = sparksee.Sparksee(cfg)
db = sparks.create(u"Hellosparks.gdb", u"HelloSparksee")
db.disable_rollback()  # Rollback is now disabled
sess = db.new_session()
graph = sess.get_graph()
# Use 'graph' to perform operations on the graph database without rollbacks
db.enable_ollback(); # Rollback is now enabled
# Use 'graph' to perform operations on the graph database
sess.close()
db.close()
sparks.close()
```

Objective-C

```
STSSparkseeConfig *cfg = [[STSSparkseeConfig alloc] init];
//The license key is required for mobile versions.
//[cfg setLicense: @"THE_LICENSE_KEY"];
STSSparksee *sparksee = [[STSSparksee alloc] initWithConfig: cfg];
// If you are not using Objective-C Automatic Reference Counting , you
// may want to release the cfg here, when it's no longer needed.
//[cfg release];
STSDatabase *db = [sparksee create: @"HelloSparksee.gdb" alias: @"HelloSparksee"];
[db disableRollback];  // Rollback is now disabled
STSSession *sess = [db createSession];
STSGraph *graph = [sess getGraph];
//Use 'graph' to perform operations on the graph database without rollbacks
[db enableRollback]; // Rollback is now enabled
//Use 'graph' to perform operations on the graph database
[sess close];
[db close];
```

```
[sparksee close];
// If you are not using Objective-C Automatic Reference Counting , you
// may want to release the sparksee here , when it's closed.
//[sparksee release];
```

The following examples illustrate the fomer explained behavior when a transaction starts as a read transaction but when the first write method (in this case, the newNode method) is executed, it becomes a write transaction:

Java

```
Value v = new Value();
sess.begin(); // Start a Transaction as a read transaction
int peopleTypeId = graph.findType("PEOPLE");
int nameAttrId = graph.findAttribute(peopleTypeId, "NAME");
// In the following newNode method the transaction becomes a write transaction
long billMurray = graph.newNode(peopleTypeId);
graph.setAttribute(billMurray, nameAttrId, v.setString("Bill Murray"));
// Create a birth year attribute
int birthYearAttrId = graph.newAttribute(peopleTypeId, "BIRTH YEAR", DataType.
    Integer, AttributeKind.Basic);
// Set Bill Murray's birth year
graph.setAttribute(billMurray, birthYearAttrId, v.setInteger(1950));
// Commit all the changes
sess.commit();
...
// Start a new transaction to change the birth year of Bill Murray
sess.begin();
graph.setAttribute(billMurray, birthYearAttrId, v.setInteger(2050));
// That change was a mistake, so use the rollback method
sess.rollback();
...
// Check that the attribute is still 1950
// We don't use a transaction, so the next method is in autocommit
graph.getAttribute(billMurray, birthYearAttrId, v);
Assert(v.getInteger() == 1950);
```

[C#]

```
Value v = new Value();
sess.Begin(); // Start a Transaction as a read transaction
int peopleTypeId = graph.FindType("PEOPLE");
int nameAttrId = graph.FindAttribute(peopleTypeId, "NAME");
// In the following NewNode method the transaction becomes a write transaction
long billMurray = graph.NewNode(peopleTypeId);
graph.SetAttribute(billMurray, nameAttrId, v.SetString("Bill Murray"));
// Create a birth year attribute
int birthYearAttrId = graph.NewAttribute(peopleTypeId, "BIRTH YEAR", DataType.
    Integer, AttributeKind.Basic);
// Set Bill Murray's birth year
graph.SetAttribute(billMurray, birthYearAttrId, v.SetInteger(1950));
// Commit all the changes
sess.Commit();
...
// Start a new transaction to change the birth year of Bill Murray
sess.Begin();
graph.SetAttribute(billMurray, birthYearAttrId, v.SetInteger(2050));
// That change was a mistake, so use the rollback method
sess.Rollback();
...
// Check that the attribute is still 1950
// We don't use a transaction, so the next method is in autocommit
graph.GetAttribute(billMurray, birthYearAttrId, v);
Assert(v.GetInteger() == 1950);
```

C++

```
Value v;
sess->Begin(); // Start a Transaction as a read transaction
type_t peopleTypeId = graph->FindType(L"PEOPLE");
attr_t nameAttrId = graph->FindAttribute(peopleTypeId, L"NAME");
// In the following NewNode method the transaction becomes a write transaction
oid_t billMurray = graph->NewNode(peopleTypeId);
graph->SetAttribute(billMurray, nameAttrId, v.SetString(L"Bill Murray"));
// Create a birth year attribute
int birthYearAttrId = graph->NewAttribute(peopleTypeId, L"BIRTH YEAR", Integer,
     Basic);
// Set Bill Murray's birth year
graph->SetAttribute(billMurray, birthYearAttrId, v.SetInteger(1950));
// Commit all the changes
sess->Commit();
...
// Start a new transaction to change the birth year of Bill Murray
sess->Begin();
graph->SetAttribute(billMurray, birthYearAttrId, v.SetInteger(2050));
// That change was a mistake, so use the rollback method
sess->Rollback();
...
// Check that the attribute is still 1950
// We don't use a transaction, so the next method is in autocommit
graph->GetAttribute(billMurray, birthYearAttrId, v);
assert(v.GetInteger() == 1950);
```

Python

```
v = sparksee.Value()
sess.begin() # Start a Transaction as a read transaction
peopleTypeId == graph.find_type("PEOPLE")
nameAttrId = graph.find_attribute(peopleTypeId, "NAME")
# In the following new_node method the transaction becomes a write transaction
billMurray = graph.new_node(peopleTypeId)
graph.set_attribute(billMurray, nameAttrId, v.set_string("Bill Murray"))
# Create a birth year attribute
birthYearAttrId = graph.new_attribute(peopleTypeId, "BIRTH YEAR", sparksee.
     DataType.INTEGER, sparksee.AttributeKind.BASIC);
# Set Bill Murray's birth year
graph.set_attribute(billMurray, birthYearAttrId, v.set_integer(1950));
# Commit all the changes
sess.commit()
...
# Start a new transaction to change the birth year of Bill Murray
sess.begin()
graph.set_attribute(billMurray, birthYearAttrId, v.set_integer(2050));
# That change was a mistake, so use the rollback method
sess.rollback();
...
# Check that the attribute is still 1950
# We don't use a transaction, so the next method is in autocommit
graph.get_attribute(billMurray, birthYearAttrId, v);
assert(v.get_integer() == 1950);
```

Objective-C

```
STSValue *v = [[STSValue alloc] init];
[sess begin]; // Start a Transaction as a read transaction
int peopleTypeId = [graph findType: @"PEOPLE"];
int nameAttrId = [graph findAttribute: peopleTypeId name: @"NAME"];
[v setString: @"Bill Murray"];
// In the following createNode method the transaction becomes a write transaction
long long billMurray = [graph createNode: peopleTypeId];
[graph setAttribute: billMurray attr: nameAttrId value: v];
// Create a birth year attribute
```

```
int birthYearAttrId = [graph createAttribute: peopleTypeId name: @"BIRTH YEAR" dt:
      STSInteger kind: STSBasic];
// Set Bill Murray's birth year
[graph setAttribute: billMurray attr: birthYearAttrId value: [v setInteger:
     1950]];
// Commit all the changes
[sess commit];
...
// Start a new transaction to change the birth year of Bill Murray
[sess begin];
[graph setAttribute: billMurray attr: birthYearAttrId value: [v setInteger:
     2050]];
// That change was a mistake, so use the rollback method
[sess rollback];
...
// Check that the attribute is still 1950
// We don't use a transaction, so the next method is in autocommit
[graph getAttributeInValue: billMurray attr: birthYearAttrId value: v];
assert([v getInteger] == 1950);
```

Note that the previous codes are also valid without the begin/commit calls because they are automatically executed in autocommit mode, where a transaction is created for each one of the calls.

Import/export data

Sparksee provides utilities to export a graph into visual format or import/export data from external data sources.

Visual export

Data stored in a Sparksee graph database can be exported to a visual-oriented representation format.

The available visual exports formats for Sparksee are defined in the ExportType class:

- Graphviz

  this is an open-source graph visualization software. The Graphviz layouts take descriptions of graphs in a text language and make diagrams in useful formats, such as images and SVG for web pages or PDF and Postscript for inclusion in other documents. The resulting formats can also be displayed in an interactive graph browser (also supports GXL, an XML dialect). Graphviz has many useful features to personalize the diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks and rolland custom shapes.

- GraphML

  GraphML is a file format for graphs based on XML, so it is especially suitable as a common denominator for all kinds of services that generate, archive, or process graphs. This is the reason it is considered a standard for graph exportation. Its core language describes the structural properties of a graph while the extensions add application-specific data. Its

main features include support of directed, undirected, and mixed graphs, hypergraphs, hierarchical graphs, graphical representations, references to external data, application-specific attribute data, and light-weight parsers.

- YGraphML

  YGraphML is a format based in GraphML that has some exclusive extensions for yEd Graph Editor. yED is a desktop application that can be used to generate graphs or diagrams which can be browsed with the application and then imported or exported to different data formats.

Sparksee includes the method Graph#export to export the stored data in any of the previously explained formats. The method has the following parameters:

- The file where the data is to be exported.
- The export type format.
- An implementation of the ExportManager class.

The ExportManager class defines the properties to be exported in the following classes:

- GraphExport: a global label can be defined for the graph.
- NodeExport: label, label color, color, shape, width, font, font size, as well as other properties can be set for a node.
- EdgeExport: label, label color, width, font, font size, arrows, as well as other properties can be set for an edge.

Note that only some of these properties are meaningful for some of the available export type formats.

The DefaultExport is an implementation of the ExportManager class already provided in the Sparksee library which performs a default exportation of the settings for the whole database. Examples of a default exportation:

Java

```
Graph graph = sess.getGraph();
...
ExportManager expMngr = new DefaultExport();
graph.export("test.dot", ExportType.Graphviz, expMngr);
```

[C#]

```
Graph graph = sess.GetGraph();
...
ExportManager expMngr = new DefaultExport();
graph.Export("test.dot", ExportType.Graphviz, expMngr);
```

C++

```
Graph * graph = sess->GetGraph();
...
ExportManager * expMngr = new DefaultExport();
graph->Export(L"test.dot", Graphviz, expMngr);
delete expMngr;
```

Python

```
graph = sess.get_graph()
...
expMngr = sparksee.DefaultExport()
graph.export("test.dot", sparksee.ExportType.GRAPHVIZ, expMngr)
```

Objective-C

```
STSGraph *graph = [sess getGraph];
...
STSExportManager *expMngr = [[STSDefaultExport alloc] init];
[graph exportGraph: @"test.dot" type: STSGraphviz em: expMngr];
//[expMngr release];
```

The ExportManager class contains the following methods:

- **getNodeType:** the user can set the properties to export all nodes belonging to a given type by setting the NodeExport in/out parameter.
- **getNode:** the user can override the previous NodeExport definition for the specific given node instance.
- **getEdgeType:** the same as getNodeType but for edge types.
- **getEdge:** the same as getNode but for edge instances.
- **enableType:** the user can skip a whole node or edge type from the export process.
- **getGraph:** the user can set certain global properties for the graph.
- **prepare:** this is called when the export process starts.
- **release:** this is called when the export process ends (the method name is **close** in Objective-C).

The following codes are an example of how to export the graph from Figure-2.1 using the previously explained methods from the ExportManager class:

Java

```
import com.sparsity.sparksee.gdb.*;

public class MyExport extends ExportManager {

    private Graph g = null;

    private int peopleTypeId = Type.InvalidType;
    private int nameAttrId = Attribute.InvalidAttribute;
    private int moviesTypeId = Type.InvalidType;
    private int titleAttrId = Attribute.InvalidAttribute;
    private int castTypeId = Type.InvalidType;
    private int directsTypeId = Type.InvalidType;
```

```
    private Value v = new Value();

    @Override
    public void prepare(Graph graph) {
        g = graph;
        peopleTypeId = g.findType("PEOPLE");
        nameAttrId = g.findAttribute(peopleTypeId, "Name");
        moviesTypeId = g.findType("MOVIES");
        titleAttrId = g.findAttribute(moviesTypeId, "Title");
        castTypeId = g.findType("CAST");
        directsTypeId = g.findType("DIRECTS");
    }

    @Override
    public void release() {
    }

    @Override
    public boolean getGraph(GraphExport ge) {
        ge.setLabel("Hollywood");
        return true;
    }

    @Override
    public boolean getNodeType(int nodetype, NodeExport ne) {
        // default node type export:
        // - PEOPLE in RED nodes
        // - MOVIES in ORANGE nodes

        if (nodetype == peopleTypeId) {
            ne.setColor(java.awt.Color.RED);
        } else if (nodetype == moviesTypeId) {
            ne.setColor(java.awt.Color.ORANGE);
        } else {
            assert false;
        }
        return true;
    }

    @Override
    public boolean getEdgeType(int edgetype, EdgeExport ee) {
        // default edge type export:
        // - CAST in YELLOW lines
        // - DIRECTS in BLUE lines

        if (edgetype == castTypeId) {
            ee.setColor(java.awt.Color.YELLOW);
        } else if (edgetype == directsTypeId) {
            ee.setColor(java.awt.Color.BLUE);
        } else {
            assert false;
        }
        return true;
    }

    @Override
    public boolean getNode(long nodeOID, NodeExport ne) {
        // specific node export:
        // - PEOPLE: use the Name attribute as label
        // - MOVIES: use the Title attribute as label

        int nodetype = g.getObjectType(nodeOID);

        if (nodetype == peopleTypeId) {
            g.getAttribute(nodeOID, nameAttrId, v);
        } else if (nodetype == moviesTypeId) {
            g.getAttribute(nodeOID, titleAttrId, v);
        } else {
            assert false;
        }

        ne.setLabel("[" + nodeOID + "]" + v.toString());
        return true;
```

```
    }

    @Override
    public boolean getEdge(long edgeOID, EdgeExport ee) {
        // default edge type export is enough

        return false;
    }

    @Override
    public boolean enableType(int type) {
        // enable all node and edge types
        return true;
    }
}
```

[C#]

```
using com.sparsity.sparksee.gdb.*;

public class MyExport : ExportManager {

    private Graph g = null;

    private int peopleTypeId = Type.InvalidType;
    private int nameAttrId = Attribute.InvalidAttribute;
    private int moviesTypeId = Type.InvalidType;
    private int titleAttrId = Attribute.InvalidAttribute;
    private int castTypeId = Type.InvalidType;
    private int directsTypeId = Type.InvalidType;

    private Value v = new Value();

    public override void Prepare(Graph graph) {
        g = graph;
        peopleTypeId = g.FindType("PEOPLE");
        nameAttrId = g.FindAttribute(peopleTypeId, "Name");
        moviesTypeId = g.FindType("MOVIES");
        titleAttrId = g.FindAttribute(moviesTypeId, "Title");
        castTypeId = g.FindType("CAST");
        directsTypeId = g.FindType("DIRECTS");
    }

    public override void Release() {
    }

    public override bool GetGraph(GraphExport ge) {
        ge.SetLabel("Hollywood");
        return true;
    }

    public override bool GetNodeType(int nodetype, NodeExport ne) {
        // default node type export:
        // - PEOPLE in RED nodes
        // - MOVIES in ORANGE nodes

        if (nodetype == peopleTypeId) {
            ne.SetColor(System.Drawing.Color.Red);
        } else if (nodetype == moviesTypeId) {
            ne.SetColor(System.Drawing.Color.Orange);
        } else {
            System.Diagnostics.Debug.Assert(false);
        }
        return true;
    }

    public override bool GetEdgeType(int edgetype, EdgeExport ee) {
        // default edge type export:
        // - CAST in YELLOW lines
        // - DIRECTS in BLUE lines
```

```
        if (edgetype == castTypeId) {
            ee.SetColor(System.Drawing.Color.Yellow);
        } else if (edgetype == directsTypeId) {
            ee.SetColor(System.Drawing.Color.Blue);
        } else {
            System.Diagnostics.Debug.Assert(false);
        }
        return true;
    }

    public override bool GetNode(long nodeOID, NodeExport ne) {
        // specific node export:
        // - PEOPLE: use the Name attribute as label
        // - MOVIES: use the Title attribute as label

        int nodetype = g.GetObjectType(nodeOID);

        if (nodetype == peopleTypeId) {
            g.GetAttribute(nodeOID, nameAttrId, v);
        } else if (nodetype == moviesTypeId) {
            g.GetAttribute(nodeOID, titleAttrId, v);
        } else {
            System.Diagnostics.Debug.Assert(false);
        }

        ne.SetLabel("[" + nodeOID + "]" + v.ToString());
        return true;
    }

    public override bool GetEdge(long edgeOID, EdgeExport ee) {
        // default edge type export is enough

        return false;
    }

    public override bool EnableType(int type) {
        // enable all node and edge types
        return true;
    }
}
```

C++

```
#include <stdio.h>
#include "gdb/Sparksee.h"
#include "gdb/Database.h"
#include "gdb/Session.h"
#include "gdb/Graph.h"
#include "gdb/Objects.h"
#include "gdb/ObjectsIterator.h"
#include "gdb/Stream.h"
#include "gdb/Export.h"
#include <assert.h>

using namespace sparksee::gdb;

class MyExport : ExportManager {

private:
    Graph * g;

    type_t peopleTypeId;
    attr_t nameAttrId;
    type_t moviesTypeId;
    attr_t titleAttrId;
    type_t castTypeId;
    type_t directsTypeId;

    Value v;

public:
```

```
    MyExport()
    : g(NULL)
    , peopleTypeId(Type::InvalidType)
    , nameAttrId(Attribute::InvalidAttribute)
    , moviesTypeId(Type::InvalidType)
    , titleAttrId(Attribute::InvalidAttribute)
    , castTypeId(Type::InvalidType)
    , directsTypeId(Type::InvalidType) {
    }

    virtual void Prepare(Graph * graph) {
        g = graph;
        peopleTypeId = g->FindType(L"PEOPLE");
        nameAttrId = g->FindAttribute(peopleTypeId, L"Name");
        moviesTypeId = g->FindType(L"MOVIES");
        titleAttrId = g->FindAttribute(moviesTypeId, L"Title");
        castTypeId = g->FindType(L"CAST");
        directsTypeId = g->FindType(L"DIRECTS");
    }

    virtual void Release() {
    }

    virtual bool GetGraph(GraphExport * ge) {
        ge->SetLabel(L"Hollywood");
        return true;
    }

    virtual bool GetNodeType(int nodetype, NodeExport * ne) {
        // default node type export:
        // - PEOPLE in RED nodes
        // - MOVIES in ORANGE nodes

        if (nodetype == peopleTypeId) {
            ne->SetColorRGB(16711680); // red
        } else if (nodetype == moviesTypeId) {
            ne->SetColorRGB(16744192); // ORANGE
        } else {
            assert(false);
        }
        return true;
    }

    virtual bool GetEdgeType(int edgetype, EdgeExport * ee) {
        // default edge type export:
        // - CAST in YELLOW lines
        // - DIRECTS in BLUE lines

        if (edgetype == castTypeId) {
            ee->SetColorRGB(16776960); // yellow
        } else if (edgetype == directsTypeId) {
            ee->SetColorRGB(255); // blue
        } else {
            assert(false);
        }
        return true;
    }

    virtual bool GetNode(long nodeOID, NodeExport * ne) {
        // specific node export:
        // - PEOPLE: use the Name attribute as label
        // - MOVIES: use the Title attribute as label

        int nodetype = g->GetObjectType(nodeOID);

        if (nodetype == peopleTypeId) {
            g->GetAttribute(nodeOID, nameAttrId, v);
        } else if (nodetype == moviesTypeId) {
            g->GetAttribute(nodeOID, titleAttrId, v);
        } else {
            assert(false);
        }
```

```
        std::wstring aux2;
        std::wstringstream aux;
        aux << L"[" << nodeOID << L"]" << v.ToString(aux2);
        ne->SetLabel(aux.str());
        return true;
    }

    virtual bool GetEdge(long edgeOID, EdgeExport * ee) {
        // default edge type export is enough

        return false;
    }

    virtual bool EnableType(int type) {
        // enable all node and edge types
        return true;
    }
};
```

Python

```
import sparksee

class MyExport(sparksee.ExportManager):

    def __init__(self):
        sparksee.ExportManager.__init__(self)

    def prepare(self, g):
        self.graph = g;
        self.people_type_id = self.graph.find_type(u"PEOPLE")
        self.name_attribute_id = self.graph.find_attribute(self.people_type_id, u"
            Name")
        self.movies_type_id = self.graph.find_type(u"MOVIES")
        self.title_attribute_id = self.graph.find_attribute(self.movies_type_id, u
            "Title")
        self.cast_type_id = self.graph.find_type(u"CAST")
        self.directs_type_id = self.graph.find_type(u"DIRECTS")

    def release(self):
        self.graph = None

    def get_graph(self, graphExport):
        graphExport.set_label("Hollywood")
        return True

    def get_node_type(self, node_type, nodeExport):
        # default node type export:
        # - PEOPLE in RED nodes
        # - MOVIES in ORANGE nodes
        if node_type == self.people_type_id:
            nodeExport.set_color_rgb(16711680)
        elif node_type == self.movies_type_id:
            nodeExport.set_color_rgb(65535)
        else:
            assert False
        return True

    def get_edge_type(self, edge_type, edgeExport):
        # default edge type export:
        # - CAST in YELLOW lines
        # - DIRECTS in BLUE lines

        if edge_type == self.cast_type_id:
            edgeExport.set_color_rgb(16776960)
        elif edge_type == self.directs_type_id:
            edgeExport.set_color_rgb(255)
        else:
            assert False
        return True
```

```
    def get_node(self, node_oid, nodeExport):
        # specific node export:
        # - PEOPLE: use the Name attribute as label
        # - MOVIES: use the Title attribute as label

        v = sparksee.Value()
        node_type = self.graph.get_object_type(node_oid)
        if node_type == self.people_type_id:
            self.graph.get_attribute(node_oid, self.name_attribute_id, v)
        elif node_type == self.movies_type_id:
            self.graph.get_attribute(node_oid, self.title_attribute_id, v)
        else:
            assert False

        nodeExport.set_label(u"[" + str(node_oid) + u"]" + v.get_string())
        return True

    def get_edge(self, edge, edgeExport):
        # default edge type export is enough
        return False

    def enable_type(self, my_type):
        # enable all node and edge types
        return True
```

Objective-C

```
#import <Sparksee/Sparksee.h>

@interface MyExport : STSExportManager
{
    STSGraph *g;
    int peopleTypeId;
    int nameAttrId;
    int moviesTypeId;
    int titleAttrId;
    int castTypeId;
    int directsTypeId;
    STSValue *value;
}
-(id)init;
-(void)prepare: (STSGraph*)graph;
-(void)close;
-(BOOL)getGraph: (STSGraphExport*)graphExport;
-(BOOL)getNodeType: (int)type nodeExport: (STSNodeExport*)nodeExport;
-(BOOL)getEdgeType: (int)type edgeExport: (STSEdgeExport*)edgeExport;
-(BOOL)getNode: (long long)node nodeExport: (STSNodeExport*)nodeExport;
-(BOOL)getEdge: (long long)edge edgeExport: (STSEdgeExport*)edgeExport;
-(BOOL)enableType: (int)type;
@end


@implementation MyExport
- (id)init
{
    self = [super init];
    return self;
}
-(void)prepare: (STSGraph*)graph
{
    g = graph;
    peopleTypeId = [g findType: @"PEOPLE"];
    nameAttrId = [g findAttribute: peopleTypeId name: @"Name"];
    moviesTypeId = [g findType: @"MOVIES"];
    titleAttrId = [g findAttribute: moviesTypeId name: @"Title"];
    castTypeId = [g findType: @"CAST"];
    directsTypeId = [g findType: @"DIRECTS"];
    value = [[STSValue alloc] init];
}
-(void)close
{
```

```
    g = NULL;
    //[value release];
}
-(BOOL)getGraph: (STSGraphExport*)graphExport
{
    [graphExport setLabel: @"Hollywood"];
    return TRUE;
}
-(BOOL)getNodeType: (int)type nodeExport: (STSNodeExport*)nodeExport
{
    // default node type export:
    // - PEOPLE in RED nodes
    // - MOVIES in ORANGE nodes
    if (type == peopleTypeId) {
        [nodeExport setColorRGB: 16711680]; // red == 0xFFFF00
    } else if (type == moviesTypeId) {
        [nodeExport setColorRGB: 0xFF7F00]; // ORANGE == 16744192
    } else {
        NSLog(@"Unknown type");
    }
    return TRUE;
}
-(BOOL)getEdgeType: (int)type edgeExport: (STSEdgeExport*)edgeExport
{
    // default edge type export:
    // - CAST in YELLOW lines
    // - DIRECTS in BLUE lines
    if (type == castTypeId) {
        [edgeExport setColorRed: 1.0 green: 1.0 blue: 0.0 alpha: 0.0]; // yellow
            == 16776960
    } else if (type == directsTypeId) {
        [edgeExport setColorRGB: 255]; // blue
    } else {
        NSLog(@"Unknown type");
    }
    return TRUE;
}
-(BOOL)getNode: (long long)node nodeExport: (STSNodeExport*)nodeExport
{
    // specific node export:
    // - PEOPLE: use the Name attribute as label
    // - MOVIES: use the Title attribute as label
    int nodetype = [g getObjectType: node];

    if (nodetype == peopleTypeId) {
        [g getAttributeInValue: node attr: nameAttrId value: value];
    } else if (nodetype == moviesTypeId) {
        [g getAttributeInValue: node attr: titleAttrId value: value];
    } else {
        NSLog(@"Unknown type");
    }

    [nodeExport setLabel: [NSString stringWithFormat: @"[%lld]%@", node, [value
        getString]]];
    return TRUE;
}
-(BOOL)getEdge: (long long)edge edgeExport: (STSEdgeExport*)edgeExport
{
    // default edge type export is enough
    return FALSE;
}
-(BOOL)enableType: (int)type
{
    // enable all node and edge types
    return TRUE;
}
@end
```

Data import

Instead of manually creating node and edge objects, data can be imported following a certain schema, in bulk.

This functionality is provided by classes in the com.sparsity.sparksee.io package for Sparkseejava, the com.sparsity.sparksee.io namespace for Sparkseenet, and in the sparksee::io namespace in Sparkseecpp.

The main classes for importing data into Sparksee are the RowReader class, which reads data from external data sources and shows this data to the user with a row-based logical format, and the TypeLoader class, which imports data into a Sparksee graph.

To read data Sparksee has a CSVReader class that is a RowReader implementation to retrieve data from a CSV file. Alternatively, users could implement a different RowReader in order to retrieve data from other specific data sources.

The most important method defined by the RowReader is the bool RowReader#read(StringList row) method which returns true if a row has been read or false otherwise. If a row has been read, the output argument row is a list of strings, each string being a column within the row. The RowReader#close method must be called once the processing ends.

The CSVReader class extends the parent class with extra functionality:

- CSVReader#open sets the path of the CSV file to be read.
- Set the locale of the file by means of a case in-sensitive string argument with the following format: *an optional language, a dot ('.') character and a coding specification.*
  - Valid languages are: en_US, ca_ES, es_ES
  - Valid coding is: UTF8, ISO88591

  Thus, valid locale arguments are: "en_US.UTF8", "es_ES.ISO88591", ".UTF8", etc. If not specified, "en_US.iso88591" is the default locale.
- Set a separator character (to limit columns) with CSVReader#setSeparator or set the quote character for strings with CSVReader#setQuotes. For quoted strings it is also possible to set a restriction for them to be single-lined or allow them to be multi-lined. In the second case the user should limit the maximum number of lines.
- Finally, it is possible to skip some lines at the beginning of the file (CSVReader#setStartLine) or fix a maximum number of lines to be read (CSVReader#setNumLines).

Let's look at an example of the use of the CSVReader class and its functionalities, for a csv such as the following:

```
ID; NAME; SALARY; AGE
1; name1; 1800; 18
2; name2; 1600; 16
3; name3; 2000; 20
```

```
4; name4; 2200; 22
```

It could be read using a CSVReader as follows:

Java

```
CSVReader csv = new CSVReader();
csv.setSeparator(";");
csv.setStartLine(1);
csv.open("people.csv");
StringList row = new StringList();
while (csv.read(row))
{
    System.out.println(">> Reading line num " + csv.getRow());
    StringListIterator it = row.iterator();
    while (it.hasNext())
    {
        System.out.println(it.next());
    }
}
csv.close();
```

[C#]

```
CSVReader csv = new CSVReader();
csv.SetSeparator(";");
csv.SetStartLine(1);
csv.Open("people.csv");
StringList row = new StringList();
while (csv.Read(row))
{
    System.Console.WriteLine(">> Reading line num " + csv.GetRow());
    StringListIterator it = row.Iterator();
    while (it.HasNext())
    {
        System.Console.WriteLine(it.Next());
    }
}
csv.Close();
```

C++

```
CSVReader csv;
csv.SetSeparator(L";");
csv.SetStartLine(1);
csv.Open(L"people.csv");
StringList row;
while (csv.Read(row))
{
    std::cout << ">> Reading line num " << csv.GetRow() << std::endl;
    StringListIterator * it = row.Iterator();
    while (it->HasNext())
    {
        std::wcout << it->Next() << std::endl;
    }
    delete it;
}
csv.Close();
```

Python

```
csv = sparksee.CSVReader()
csv.set_separator(u";")
csv.set_start_line(1)
csv.open(u"people.csv")
row = sparksee.StringList()
while csv.read(row):
    print ">> Reading line num " + str(csv.get_row())
    for elem in row:
        print elem
csv.close()
```

Objective-C

```
STSCSVReader *csv = [[STSCSVReader alloc] init];
[csv setSeparator: @";"];
[csv setStartLine: 1];
[csv open: @"people.csv"];
STSStringList *row = [[STSStringList alloc] init];
while ([csv read: row])
{
    NSLog(@">> Reading line num %d\n", [csv getRow]);
    STSStringListIterator * it = [row iterator];
    while ([it hasNext])
    {
        NSLog(@"%@\n", [it next]);
    }
}
[csv close];
//[csv release];
//[row release];
```

Note that the separator and the first line to be read have been set in the first place.

There are two implementations for the TypeLoader class, regarding whether the element to be included in the graph is a node, the NodeTypeLoader class, or an edge, the EdgeTypeLoader class.

The functionality of the TypeLoader class includes:

- Execution mode:
  - TypeLoader#run. The CSV file is read once and the objects are created in the graph.
  - TypeLoader#runTwoPhases. The CSV file is read twice: once to create the objects and once again to set the attribute values.
  - TypeLoader#runNPhases(int num). The CSV file is read N times: once to create the objects and then once for each attribute column. Each attribute column is logically partitioned, so the data column is loaded in num partitions.
- It is possible to register listeners to receive progress events from the TypeLoader (TypeLoader#register). This requires the user to implement the TypeLoaderListener class. The frequency with which these events are generated can also be configured (TypeLoader#setFrequency).

- In case of parsing a string column to load a Timestamp attribute, the timestamp format can be set by the user.

  Valid format fields are:

  - yyyy: Year
  - yy: Year without century interpreted. Within 80 years before or 20 years after the current year. For example, if the current year is 2007, the pattern MM/dd/yy for the value 01/11/12 parses to January 11, 2012, while the same pattern for the value 05/04/64 parses to May 4, 1964.
  - MM: Month [1..12]
  - dd: Day of month [1..31]
  - hh: Hour [0..23]
  - mm: Minute [0..59]
  - ss: Second [0..59]
  - SSS: Millisecond [0..999]

  Therefore, valid timestamp formats may be: MM/dd/yy or MM/dd/yyyy −hh.mm. If not specified, the parser automatically tries to match any of the following timestamp formats:

  - yyyy−MM−dd hh:mm:ss.SSS
  - yyyy−MM−dd hh:mm:ss
  - yyyy−MM−dd

The following examples show how to use the CSVReader and the TypeLoader classes in order to execute a basic import of the nodes “PEOPLE” that have two attributes “Name and”Age” into a graph:

Java

```
Graph graph = sess.getGraph();
int peopleTypeId = graph.findType("PEOPLE");
...
// configure CSV reader
CSVReader csv = new CSVReader();
csv.setSeparator(";");
csv.setStartLine(1);
csv.open("people.csv");
// set attributes to be loaded and their positions
AttributeList attrs = new AttributeList();
Int32List attrPos = new Int32List();
// NAME attribute in the second column
attrs.add(graph.findAttribute(peopleTypeId, "Name"));
attrPos.add(1);
// AGE attribute in the fourth column
attrs.add(graph.findAttribute(peopleTypeId, "Age"));
attrPos.add(3);
// import PEOPLE node type
NodeTypeLoader ntl = new NodeTypeLoader(csv, graph, peopleTypeId, attrs, attrPos);
ntl.setLogError("people.csv.log");
ntl.run();
csv.close();
```

[C#]

```
Graph graph = sess.GetGraph();
int peopleTypeId = graph.FindType("PEOPLE");
...
// configure CSV reader
CSVReader csv = new CSVReader();
csv.SetSeparator(";");
csv.SetStartLine(1);
csv.Open("people.csv");
// set attributes to be loaded and their positions
AttributeList attrs = new AttributeList();
Int32List attrPos = new Int32List();
// NAME attribute in the second column
attrs.Add(graph.FindAttribute(peopleTypeId, "NAME"));
attrPos.Add(1);
// AGE attribute in the fourth column
attrs.Add(graph.FindAttribute(peopleTypeId, "AGE"));
attrPos.Add(3);
// import PEOPLE node type
NodeTypeLoader ntl = new NodeTypeLoader(csv, graph, peopleTypeId, attrs, attrPos);
ntl.SetLogError("people.csv.log");
ntl.Run();
csv.Close();
```

C++

```
Graph * graph = sess->GetGraph();
type_t peopleTypeId = graph->FindType(L"PEOPLE");
...
// configure CSV reader
CSVReader csv;
csv.SetSeparator(L";");
csv.SetStartLine(1);
csv.Open(L"people.csv");
// set attributes to be loaded and their positions
AttributeList attrs;
Int32List attrPos;
// NAME attribute in the second column
attrs.Add(graph->FindAttribute(peopleTypeId, L"NAME"));
attrPos.Add(1);
// AGE attribute in the fourth column
attrs.Add(graph->FindAttribute(peopleTypeId, L"AGE"));
attrPos.Add(3);
// import PEOPLE node type
NodeTypeLoader ntl(csv, *graph, peopleTypeId, attrs, attrPos);
ntl.SetLogError(L"people.csv.log");
ntl.Run();
csv.Close();
```

Python

```
graph = sess.get_graph()
people_type_id = graph.new_node_type(u"PEOPLE")
...
# configure CSV reader
csv = sparksee.CSVReader()
csv.set_separator(u";")
csv.set_start_line(1)
csv.open(u"people.csv")
# set attributes to be loaded and their positions
attrs = sparksee.AttributeList()
attrPos = sparksee.Int32List()
# NAME attribute in the second column
attrs.add(graph.find_attribute(people_type_id, u"Name"))
attrPos.add(1)
# AGE attribute in the fourth column
attrs.add(graph.find_attribute(people_type_id, u"Age"))
attrPos.add(3)
```

```
# import PEOPLE node type
ntl = sparksee.NodeTypeLoader(csv, graph, people_type_id, attrs, attrPos)
ntl.set_log_error(u"people.csv.log")
ntl.run()
csv.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
peopleTypeId = [graph findType: @"PEOPLE"];
...
// configure CSV reader
csv = [[STSCSVReader alloc] init];
[csv setSeparator: @";"];
[csv setStartLine: 1];
[csv open: @"people.csv"];
// set attributes to be loaded and their positions
STSAttributeList *attrs = [[STSAttributeList alloc] init];
STSInt32List *attrPos = [[STSInt32List alloc] init];
// NAME attribute in the second column
[attrs add: [graph findAttribute: peopleTypeId name: @"name"]];
[attrPos add: 1];
// AGE attribute in the fourth column
[attrs add: [graph findAttribute: peopleTypeId name: @"AGE"]];
[attrPos add: 3];
// import PEOPLE node type
STSNodeTypeLoader *ntl = [[STSNodeTypeLoader alloc] initWithRowReader: csv graph:
     graph type: peopleTypeId attrs: attrs attrsPos: attrPos];
[ntl setLogError: @"people.csv.log"];
[ntl run];
[csv close];
//[csv release];
//[attrs release];
//[ntl release];
```

Specifically the EdgeTypeLoader has two set of methods to specify the source and the target nodes of the edge:

- setHeadPosition and setTailPosition set which columns from the RowReader contain information about the target and source nodes respectively.
- For the referenced columns, setHeadAttribute and setTailAttribute indicate the attribute identifier to be used to find the target and source node respectively. Thus, the value in that column will be used to find an object with that value for that attribute. That node will be the target or the source node, accordingly, for the new edge.

Data export

Analogously to the data import, this functionality is provided by classes in the com.sparsity.sparksee.io package for Sparkseejava, the com.sparsity.sparksee.io namespace for Sparkseenet, and in the sparksee::io namespace in Sparkseecpp.

The main classes for exporting data from Sparksee are the RowWriter class, which writes data from a row-based logical format to an external data source, and the TypeExporter class, which exports data from a Sparksee graph.

The CSVWriter class is a RowWriter implementation for writing data from a Sparksee graph into a CSV file. Alternatively, users may implement a different RowWriter to export data into other data sources.

The most important method defined by the RowWriter is the bool RowWriter #write(StringList row) method which writes data to an external storage. The RowWriter#close method must be called once the processing ends.

The CSVWriter class extends its parent class with extra functionality:

- CSVWriter#open sets the path of the CSV file to be written.
- The locale of the file is set by means of a case in-sensitive string argument with the following format: *an optional language, a dot ('.') character and a coding specification.*
  - Valid languages are: en_US, ca_ES, es_ES
  - Valid coding is: UTF8, ISO88591

  Thus, valid locale arguments are: "en_US.UTF8", "es_ES.ISO88591", ".UTF8", etc. If not specified, "en_US.iso88591" is the default locale.
- A separator character is set (to limit columns) with CSVWriter# setSeparator or the quote character is set for strings with CSVWriter #setQuotes. Quoted strings can also be forced with the CSVwriter# setForcedQuotes method, or managed automatically with the CSVwriter# setAutoQuotesmethod.

The following examples show the use of the CSVWriter class, in order to add "PEOPLE" nodes with their attributes from the graph into a csv file:

Java

```
CSVWriter csv = new CSVWriter();
csv.setSeparator("|");
csv.open("people.csv");
StringList row = new StringList();
// write header
row.add("ID");
row.add("NAME");
row.add("AGE");
csv.write(row);
// write rows
BooleanList quotes = new BooleanList();
quotes.add(false);
quotes.add(true); // force second column to be quoted
quotes.add(false);
csv.setForcedQuotes(quotes); // enables de quoting rules
row.clear();
row.add("1");
row.add("Woody Allen");
row.add("77");
csv.write(row);
row.clear();
row.add("2");
row.add("Scarlett Johansson");
row.add("28");
csv.write(row);
csv.close();
```

[C#]

```
CSVWriter csv = new CSVWriter();
csv.SetSeparator("|");
csv.Open("people.csv");
StringList row = new StringList();
// write header
row.Add("ID");
row.Add("NAME");
row.Add("AGE");
csv.Write(row);
// write rows
BooleanList quotes = new BooleanList();
quotes.Add(false);
quotes.Add(true); // force second column to be quoted
quotes.Add(false);
csv.SetForcedQuotes(quotes); // enables de quoting rules
row.Clear();
row.Add("1");
row.Add("Woody Allen");
row.Add("18");
csv.Write(row);
row.Clear();
row.Add("2");
row.Add("Scarlett Johansson");
row.Add("28");
csv.Write(row);
csv.Close();
```

C++

```
CSVWriter csv;
csv.SetSeparator(L"|");
csv.Open(L"people.csv");
StringList row;
// write header
row.Add(L"ID");
row.Add(L"NAME");
row.Add(L"AGE");
csv.Write(row);
// write rows
BooleanList quotes;
quotes.Add(false);
quotes.Add(true); // force second column to be quoted
quotes.Add(false);
csv.SetForcedQuotes(quotes); // enables de quoting rules
row.Clear();
row.Add(L"1");
row.Add(L"Woody Allen");
row.Add(L"18");
csv.Write(row);
row.Clear();
row.Add(L"2");
row.Add(L"Scarlett Johansson");
row.Add(L"28");
csv.Write(row);
csv.Close();
```

Python

```
csv = sparksee.CSVWriter()
csv.set_separator("|")
csv.open("peopleWritten.csv")
row = sparksee.StringList()
# write header
row.add("ID")
row.add("NAME")
```

```
row.add("AGE")
csv.write(row)
# write rows
quotes = sparksee.BooleanList()
quotes.add(False)
quotes.add(True) # force second column to be quoted
quotes.add(False)
csv.set_forced_quotes(quotes)
row.clear()
row.add("1")
row.add("Woody Allen")
row.add("77")
csv.write(row)
row.clear()
row.add("2")
row.add("Scarlett Johansson")
row.add("28")
csv.write(row)
csv.close()
```

Objective-C

```
STSCSVWriter *csv = [[STSCSVWriter alloc] init];
[csv setSeparator: @"|"];
[csv open: @"people_out.csv"];
STSStringList *row = [[STSStringList alloc] init];
// write header
[row add: @"ID"];
[row add: @"NAME"];
[row add: @"AGE"];
[csv write: row];
// write rows
STSBooleanList *quotes = [[STSBooleanList alloc] init];
[quotes add: FALSE];
[quotes add: TRUE]; // force second column to be quoted
[quotes add: FALSE];
[csv setForcedQuotes: quotes]; // enables de quoting rules
[row clear];
[row add: @"1"];
[row add: @"Woody Allen"];
[row add: @"18"];
[csv write: row];
[row clear];
[row add: @"2"];
[row add: @"Scarlett Johansson"];
[row add: @"28"];
[csv write: row];
[csv close];
//[csv release];
//[row release];
//[quotes release];
```

The output for the previous examples would be the following csv file:

```
ID | NAME | AGE
1 | "Woody Allen" | 77
2 | "Scarlett Johansson" | 28
```

The TypeExporter class exports a specific type of the graph to a external storage. There are two implementations for the TypeExporter class, regarding whether the element to be exported from the graph is a node, the NodeTypeExporter class, or an edge, the EdgeTypeExporter class. A TypeExporter retrieves all the

data from the instances belonging to a node or edge type and writes it using a RowWriter.

The TypeExporter class includes the following functionalities:

- TypeExporter#run executes the export process.
- It is possible to register listeners to receive progress events from the TypeExporter (TypeExporter#register). This requires the user to implement the TypeExporterListener class. The frequency with which these events are generated can also be configured (TypeExporter#setFrequency).
- TypeExporter#setAttributes sets the list of attributes to be exported.

The following examples export PEOPLE nodes from the graph with their attributes Name and Age, and write the information into a csv file:

Java

```
Graph graph = sess.getGraph();
int peopleTypeId = graph.findType("PEOPLE");
int nameAttrId = graph.findAttribute(peopleTypeId, "Name");
int ageAttrId = graph.findAttribute(peopleTypeId, "Age");
...
// configure CSV writer
CSVWriter csv = new CSVWriter();
csv.setSeparator("|");
csv.setAutoQuotes(true);
csv.open("people.csv");
// export PEOPLE node type: Name and Age attributes
AttributeList attrs = new AttributeList();
attrs.add(nameAttrId);
attrs.add(ageAttrId);
NodeTypeExporter nte = new NodeTypeExporter(csv, graph, peopleTypeId, attrs);
nte.run();
csv.close();
```

[C#]

```
Graph graph = sess.GetGraph();
int peopleTypeId = graph.FindType("PEOPLE");
int nameAttrId = graph.FindAttribute(peopleTypeId, "Name");
int ageAttrId = graph.FindAttribute(peopleTypeId, "Age");
...
// configure CSV writer
CSVWriter csv = new CSVWriter();
csv.SetSeparator("|");
csv.SetAutoQuotes(true);
csv.Open("people.csv");
// export PEOPLE node type: Name and Age attributes
AttributeList attrs = new AttributeList();
attrs.Add(nameAttrId);
attrs.Add(ageAttrId);
NodeTypeExporter nte = new NodeTypeExporter(csv, graph, peopleTypeId, attrs);
nte.Run();
csv.Close();
```

C++

```
Graph * graph = sess->GetGraph();
type_t peopleTypeId = graph->FindType(L"PEOPLE");
attr_t nameAttrId = graph->FindAttribute(peopleTypeId, L"Name");
attr_t ageAttrId = graph->FindAttribute(peopleTypeId, L"Age");
...
// configure CSV writer
CSVWriter csv;
csv.SetSeparator(L"|");
csv.SetAutoQuotes(true);
csv.Open(L"people.csv");
// export PEOPLE node type: Name and Age attributes
AttributeList attrs;
attrs.Add(nameAttrId);
attrs.Add(ageAttrId);
NodeTypeExporter nte(csv, *graph, peopleTypeId, attrs);
nte.Run();
csv.Close();
```

Python

```
graph = sess.get_graph()
people_type_id = graph.new_node_type(u"PEOPLE")
name_attr_id = graph.new_attribute(people_type_id, u"Name", sparksee.DataType.
    STRING, sparksee.AttributeKind.INDEXED)
age_attr_id = graph.new_attribute(people_type_id, u"Age", sparksee.DataType.
    INTEGER, sparksee.AttributeKind.BASIC
...
# configure CSV writer
csv = sparksee.CSVWriter()
csv.set_separator(u"|")
csv.set_auto_quotes(True)
csv.open(u"people.csv")
# export PEOPLE node type: Name and Age attributes
attrs = sparksee.AttributeList()
attrs.add(name_attr_id)
attrs.add(age_attr_id)
nte = sparksee.NodeTypeExporter(csv, graph, people_type_id, attrs)
nte.run()
csv.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
int peopleTypeId = [graph findType: @"people"];
int nameAttrId = [graph findAttribute: peopleTypeId name: @"name"];
int ageAttrId = [graph findAttribute: peopleTypeId name: @"AGE"];

// configure CSV writer
STSCSVWriter *csv = [[STSCSVWriter alloc] init];
[csv setSeparator: @"|"];
[csv setAutoQuotes: TRUE];
[csv open: @"people_out.csv"];
// export PEOPLE node type: Name and Age attributes
STSAttributeList *attrs = [[STSAttributeList alloc] init];
[attrs add: nameAttrId];
[attrs add: ageAttrId];
STSNodeTypeExporter *nte = [[STSNodeTypeExporter alloc] initWithRowWriter: csv
    graph: graph type: peopleTypeId attrs: attrs];
[nte run];
[csv close];
//[csv release];
//[attrs release];
//[nte release];
```

Specifically the EdgeTypeExporter has two set of methods to specify the source and target nodes to be exported:

- setHeadPosition and setTailPosition set which columns from the RowWriter will be used to export the target and source node respectively.
- For the referenced columns, setHeadAttribute and setTailAttribute indicate the attribute identifier to be used to export the source and target node respectively. Thus, the value in that column of the current row will correspond to the value of that object for that attribute.

Scripting

This functionality is provided by classes in the com.sparsity.sparksee.script package for Sparkseejava, the com.sparsity.sparksee.script namespace for Sparkseenet, and in the sparksee::script namespace in Sparkseecpp.

Users can also interact with Sparksee through a script file, mainly to create or delete objects. Although the grammar and examples of Sparksee script language are explained in the 'Scripting' chapter, this section explains how to execute those scripts.

ScriptParser class

The ScriptParser class is in the com.sparsity.sparksee.script package or namespace in Sparkseejava or Sparkseenet respectively, or in the sparksee::script namespace in Sparkseecpp.

Once instantiated, this class allows the parsing and/or executing of a Sparksee script file with the method ScriptParser#parse.

The ScriptParser class sets the output and error log paths, and also sets the locale of the file to be processed. More information about the locale formats can be found in the 'Data import' and 'Data export' sections of this chapter.

Finally, the static method ScriptParser::generateSchemaScript generates and dumps the schema of a Sparksee database into an output Sparksee script file path.

Interactive execution

The same ScriptParser class is the one that allows an interactive command-line execution:

- In the case of Sparkseejava, the *jar* library includes a static main method together with the ScriptParser class. It just calls the ScriptParser class to provide a command-line interface to that class.
- In the case of Sparkseenet and Sparkseecpp, the distribution includes a ScriptParser executable that calls the ScriptParser class to provide a command-line interface to that class.

In all cases, the application has one required argument, a Sparksee script file path, and one optional argument, a case-insensitive boolean (true or false) to force the execution of the Sparksee script file or to just parse it; true (execute) is the default.

Help is shown in case the user provides no arguments:

Java

```
$ java -cp sparkseejava.jar com.sparsity.sparksee.script.ScriptParser
Wrong number of arguments.
Usage: java -cp sparkseejava.jar com.sparsity.sparksee.script.ScriptParser <
     script_file.des> [bool_run]
Where:
    script_file.des Is the required file containing the script commands.
    bool_run: True (default) = run the commands / False = just check the script.
```

[C#]

```
$ ScriptParser
Wrong number of arguments.
Usage: ScriptParser.exe <script_file.des> [bool_run]
Where:
    script_file.des Is the required file containing the script commands.
    bool_run: True (default) = run the commands / False = just check the script.
```

C++

```
$ ./ScriptParser
Wrong number of arguments.
Usage: ScriptParser <script_file.des> [bool_run]
Where:
    script_file.des Is the required file containing the script commands.
    bool_run: True (default) = run the commands / False = just check the script.
```

Python

```
$ python ScriptParser.py
Wrong number of arguments.
Usage:  ScriptParser.py  <script_file.des> [bool_run]
Where:
    script_file.des Is the required file containing the script commands.
    bool_run: True (default) = run the commands / False = just check the script.
```

Algorithms

This functionality is provided by classes in the com.sparsity.sparksee.algorithms package for Sparkseejava, the com.sparsity.sparksee.algorithms namespace for Sparkseenet, and in the sparksee::algorithms namespace in Sparkseecpp.

Sparksee API includes a set of generalist graph algorithms which can be categorized as follows:

- Traversal: for visiting the nodes in a graph.
- Shortest path: for computing the shortest path between two nodes in a graph.
- Connectivity: for computing the connected components, those connected to each other forming a subgraph.
- Community detection: for computing communities of nodes densely connected.

Traversal

To traverse a graph is to visit its nodes starting from one of them. Several filters and restrictions can be specified for the traversal:

- *Restrict which node types can be traversed.* The user can specify which node types are valid for the traversal, so nodes belonging to other node types will be ignored.
- *Restrict which edge types can be traversed and the direction.* The user can specify which edge types are valid for the traversal plus the direction of the navigation, so the user can set to navigate only through out-going, in-coming or both types of edges.
- *Exclude a collection of node identifiers.* The user can specify a collection of node identifiers to be ignored during the traversal.
- *Exclude a collection of edge identifiers.* The user can specify a collection of edge identifiers to be ignored during the traversal.
- *Set a maximum number of hops.* The user can specify the maximum distance to be visited, so nodes further from that will be ignored.

There are two implementations available for this algorithm:

- TraversalBFS class implements the traversal using a breadth-first search strategy.
- TraversalDFS class implements the traversal using a depth-first search strategy.

All previous traversal classes have an iterator pattern. Once instantiated and configured, the user must call Traversal#hasNext and Traversal#next in order to visit the next node. Take into account that is strongly recommended to close (delete for Sparkseecpp) the traversal instance as soon as it is no longer needed.

The following examples navigate three hops through outgoing edges visiting only PEOPLE nodes:

Java

```
Graph graph = sess.getGraph();
long src = ... // source node identifier
...
TraversalDFS dfs = new TraversalDFS(sess, src);
dfs.addAllEdgeTypes(EdgesDirection.Outgoing);
dfs.addNodeType(graph.findType("PEOPLE"));
dfs.setMaximumHops(3);
```

```
while (dfs.hasNext())
{
    System.out.println("Current node " + dfs.next()
                       + " at depth " + dfs.getCurrentDepth());
}
dfs.close();
```

[C#]

```
Graph graph = sess.GetGraph();
long src = ... // source node identifier
...
TraversalDFS dfs = new TraversalDFS(sess, src);
dfs.AddAllEdgeTypes(EdgesDirection.Outgoing);
dfs.AddNodeType(graph.FindType("PEOPLE"));
dfs.SetMaximumHops(3);
while (dfs.HasNext())
{
    System.Console.WriteLine("Current node " + dfs.Next()
                       + " at depth " + dfs.GetCurrentDepth());
}
dfs.Close();
```

C++

```
Graph * graph = sess->GetGraph();
oid_t src = ... // source node identifier
...
TraversalDFS dfs(*sess, src);
dfs.AddAllEdgeTypes(Outgoing);
dfs.AddNodeType(graph->FindType(L"PEOPLE"));
dfs.SetMaximumHops(3);
while (dfs.HasNext())
{
    std::cout << "Current node " << dfs.Next()
                  << " at depth " << dfs.GetCurrentDepth() << std::endl;
}
```

Python

```
graph = sess.get_graph()
src = ... # source node identifier
...
dfs = sparksee.TraversalDFS(sess, src)
dfs.add_all_edge_types(sparksee.EdgesDirection.OUTGOING)
dfs.add_node_type(graph.find_type(u"PEOPLE"))
dfs.set_maximum_hops(3)
while dfs.has_next():
    print "Current node ", dfs.next(), " at depth ", dfs.get_current_depth()
dfs.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
long long src = ... // source node identifier
...
STSTraversalDFS *dfs = [[STSTraversalDFS alloc] initWithSession: sess node: src];
[dfs addAllEdgeTypes: STSOutgoing];
[dfs addNodeType: [graph findType: @"PEOPLE"]];
[dfs setMaximumHops: 3];
```

```
while ([dfs hasNext])
{
    NSLog(@"Current node %lld at depth %d\n", [dfs next], [dfs getCurrentDepth]);
}
[dfs close];
//[dfs release];
```

Context

The Context is a complementary class that has a very similar interface and provides the same functionality as the Traversal class. Instead of visiting each node using an iterator pattern, Context class returns an Objects instance which contains all the "visited" nodes. Similarly to what happens with the Traversal instances, all Context instances must be closed (or deleted in the case of Sparksee cpp) when they are no longer in use.

The following examples repeat the three-hop navigation but this time using the Context class:

Java

```
Graph graph = sess.getGraph();
long src = ... // source node identifier
...
Context ctx = new Context(sess, src);
ctx.addAllEdgeTypes(EdgesDirection.Outgoing);
ctx.addNodeType(graph.findType("PEOPLE"));
ctx.setMaximumHops(3, true);
Objects objs = ctx.compute();
...
objs.close();
ctx.close();
```

[C#]

```
Graph graph = sess.GetGraph();
long src = ... // source node identifier
...
Context ctx = new Context(sess, src);
ctx.AddAllEdgeTypes(EdgesDirection.Outgoing);
ctx.AddNodeType(graph.FindType("PEOPLE"));
ctx.SetMaximumHops(3, true);
Objects objs = ctx.Compute();
...
objs.Close();
ctx.Close();
```

C++

```
Graph * graph = sess->GetGraph();
oid_t src = ... // source node identifier
...
Context ctx(*sess, src);
ctx.AddAllEdgeTypes(Outgoing);
ctx.AddNodeType(graph->FindType(L"PEOPLE"));
ctx.SetMaximumHops(3, true);
Objects * objs = ctx.Compute();
...
```

```
delete objs;
```

Python

```
graph = sess.get_graph()
src = ... # source node identifier
...
ctx = sparksee.Context(sess, src)
ctx.add_all_edge_types(sparksee.EdgesDirection.OUTGOING)
ctx.add_node_type(graph.find_type(u"PEOPLE"))
ctx.set_maximum_hops(3, True)
objs = sparksee.Context.compute(ctx)
...
objs.close();
```

Objective-C

```
STSGraph *graph = [sess getGraph];
long long src = ... // source node identifier
...
STSContext *ctx = [[STSContext alloc] initWithSession: sess node: src];
[ctx addAllEdgeTypes: STSOutgoing];
[ctx addNodeType: [graph findType: @"people"]];
[ctx setMaximumHops: 3 include: TRUE];
STSObjects * objs = [ctx compute];
...
[objs close];
[ctx close];
//[ctx release];
```

Shortest path

To find a shortest path in a graph is to discover which edges and nodes should be visited in order to go from one node to another in the fastest way. Several filters and restrictions can be specified in order to find the most appropriate path:

- *Restrict which node types can be traversed:* the user can specify which node types are valid for the computation of the shortest path, so nodes belonging to other node types will be ignored.
- *Restrict which edge types can be traversed and the direction:* the user can specify which edge types are valid for the shortest path plus the direction of the navigation, so the user can set to navigate only through out-going, in-coming or both type of edges.
- *Exclude a collection of node identifiers:* the user can specify a collection of node identifiers to be ignored during the computation of the shortest path.
- *Exclude a collection of edge identifiers:* the user can specify a collection of edge identifiers to be ignored during the computation of the shortest path.

- *Set a maximum number of hops:* the user can specify the maximum distance to be visited, so paths longer than this distance will be discarded.

All Sparksee shortest path implementations inherit from a specific ShortestPath subclass called SinglePairShortestPath. Additionally, this class defines the following methods to retrieve the results:

- SinglePairShortestPath#getPathAsEdges: returns the computed shortest path as an edge identifier sequence.
- SinglePairShortestPath#getPathAsNodes: returns the computed shortest path as a node identifier sequence.

In fact, there are two specific implementations of SinglePairShortestPath:

- SinglePairShortestPathBFS class which solves the problem using a breadth-first search strategy. This implementation solves the problem of unweighted graphs, as it assumes all edges have the same cost.
- SinglePairShortestPathDijkstra class which solves the problem based on the well-known Dijkstra's algorithm. This algorithm solves the shortest path problem for a graph with positive edge path costs, producing a shortest path tree.

  With the same class the user may add weights to the edges of the graph using the addWeightedEdgeType method. That attribute will be used during the computation of the algorithm to retrieve the cost of that edge. The class also includes the getCost method to retrieve the cost of the computed shortest path.

The ShortestPath class and all its subclasses have a close method which must be called once the instances are no longer in use in order to free internal resources (or delete them in the case of Sparkseecpp).

The following examples show the use of the Dijkstra shortest path:

Java

```
Graph graph = sess.getGraph();
long src = ... // source node identifier
long dst = ... // destination node identifier
...
SinglePairShortestPathDijkstra spspd = new SinglePairShortestPathDijkstra(sess,
     src, dst);
spspd.addAllNodeTypes();
int roadTypeId = graph.findType("ROAD");
int distanceAttrId = graph.findAttribute(roadTypeId, "DISTANCE");
spspd.addWeightedEdgeType(roadTypeId, EdgesDirection.Outgoing, distanceAttrId);
spspd.setMaximumHops(4);
spspd.run();
if (spspd.exists())
{
    double cost = spspd.getCost();
    OIDList nodes = spspd.getPathAsNodes();
    OIDList edges = spspd.getPathAsEdges();
}
spspd.close();
```

[C#]

```
Graph graph = sess.GetGraph();
long src = ... // source node identifier
long dst = ... // destination node identifier
...
SinglePairShortestPathDijkstra spspd = new SinglePairShortestPathDijkstra(sess,
    src, dst);
spspd.AddAllNodeTypes();
int roadTypeId = graph.FindType("ROAD");
int distanceAttrId = graph.FindAttribute(roadTypeId, "DISTANCE");
spspd.AddWeightedEdgeType(roadTypeId, EdgesDirection.Outgoing, distanceAttrId);
spspd.SetMaximumHops(4);
spspd.Run();
if (spspd.Exists())
{
    double cost = spspd.GetCost();
    OIDList nodes = spspd.GetPathAsNodes();
    OIDList edges = spspd.GetPathAsEdges();
}
spspd.Close();
```

C++

```
Graph * graph = sess->GetGraph();
oid_t src = ... // source node identifier
oid_t dst = ... // destination node identifier
...
SinglePairShortestPathDijkstra spspd(*sess, src, dst);
spspd.AddAllNodeTypes();
type_t roadTypeId = graph->FindType(L"ROAD");
attr_t distanceAttrId = graph->FindAttribute(roadTypeId, L"DISTANCE");
spspd.AddWeightedEdgeType(roadTypeId, Outgoing, distanceAttrId);
spspd.SetMaximumHops(4);
spspd.Run();
if (spspd.Exists())
{
    double cost = spspd.GetCost();
    OIDList * nodes = spspd.GetPathAsNodes();
    OIDList * edges = spspd.GetPathAsEdges();
    ...
    delete nodes;
    delete edges;
}
```

Python

```
graph = sess.get_graph()
src = ... # source node identifier
dst = ... # destination node identifier
...
spspd = sparksee.SinglePairShortestPathDijkstra(sess, src, dst)
spspd.add_all_node_types()
road_type_id = graph.find_type("ROAD")
distance_attr_id = graph.find_attribute(road_type_id, "DISTANCE")
spspd.add_weighted_edge_type(road_type_id, sparksee.EdgesDirection.OUTGOING,
    distance_attr_id)
spspd.set_maximum_hops(4)
spspd.run()
if spspd.exists():
    cost = spspd.get_cost()
    nodes = spspd.get_path_as_nodes()
    edges = spspd.get_path_as_edges()
    ...
    nodes.close()
    edges.close()
spspd.close()
```

Objective-C

```
STSGraph *graph = [sess getGraph];
long long src = ... // source node identifier
long long dst = ... // destination node identifier
...
STSSinglePairShortestPathDijkstra *spspd = [[STSSinglePairShortestPathDijkstra
     alloc] initWithSession: sess src: src dst: dst];
[spspd addAllNodeTypes];
int roadTypeId = [graph findType: @"ROAD"];
int distanceAttrId = [graph findAttribute: roadTypeId name: @"DISTANCE"];
[spspd addWeightedEdgeType: roadTypeId dir: STSOutgoing attr: distanceAttrId];
[spspd setMaximumHops: 4];
[spspd run];
if ([spspd exists])
{
    double cost = [spspd getCost];
    STSOidList * nodes = [spspd getPathAsNodes];
    STSOidList * edges = [spspd getPathAsEdges];
    ...
}
[spspd close];
[spspd release];
}
```

Connectivity

Discovering the connected components is a common problem in graph theory. A connected component is a subgraph where any two nodes are connected to each other by paths, whilst at the same time it is not connected to any additional node in the supergraph.

Connectivity is the basic class for all the different implementations. Several filters and restrictions can be specified in order to find the connected components:

- *Restrict which node types can be traversed:* the user can specify which node types are valid for the computation, so nodes belonging to other node types will be ignored.
- *Exclude a collection of node identifiers:* the user can specify a collection of node identifiers to be ignored during the computation.
- *Exclude a collection of edge identifiers:* the user can specify a collection of edge identifiers to be ignored during the computation.
- *Attribute for the materialization of the computation:* the user can specify an attribute where the results of the computation are persistently store. The identifier of the connected component will be stored for each node as the value of that attribute. The connected component identifier is a numeric value in the range $[0..C-1]$ where C is the number of different connected components. Note that the results are not automatically updated in the case that the graph topology is updated (node or edge object added or removed).

Depending on whether the graph is managed as a directed or undirected graph, there are two types of Connectivity subclasses:

- StrongConnectivity class solves the connected components problem for directed graphs. This class provides methods for restricting which edge types can be traversed during the computation as well as the direction of those edges. The implementation for this class is the StrongConnectivityGabow class that uses the Gabow's algorithm strategy.
- WeakConnectivity class solves the connected components problem for undirected graphs, where all edges are considered as undirected. Note that if the graph contains directed edge types, these edges will be managed as if they were simply undirected edge types. The implementation for this class is the WeakConnectivityDFS class that use the DFS strategy.

Additionally the ConnectedComponents class helps the user manage the result of a connectivity computation. It retrieves the number of connected components, the connected component identifier for each node, and the size of a certain connected component or all the elements in a connected component.

All the connectivity classes and subclasses have a close method in order to free resources as soon as the instances are no longer in use (deleted in the case of Sparkseecpp).

Java

```
StrongConnectivityGabow scg = new StrongConnectivityGabow(sess);
scg.addAllNodeTypes();
scg.addAllEdgeTypes(EdgesDirection.Outgoing);
scg.run();
ConnectedComponents cc = scg.getConnectedComponents();
for (int i = 0; i < cc.getCount(); i++)
{
    System.out.println("# component: " + i + " size: " + cc.getSize(i));
    Objects objs = cc.getNodes(i);
    ...
    objs.close();
}
cc.close();
scg.close();
```

[C#]

```
StrongConnectivityGabow scg = new StrongConnectivityGabow(sess);
scg.AddAllNodeTypes();
scg.AddAllEdgeTypes(EdgesDirection.Outgoing);
scg.Run();
ConnectedComponents cc = scg.GetConnectedComponents();
for (int i = 0; i < cc.GetCount(); i++)
{
    System.Console.WriteLine("# component: " + i + " size: " + cc.GetSize(i));
    Objects objs = cc.GetNodes(i);
    ...
    objs.Close();
}
cc.Close();
scg.Close();
```

C++

```
StrongConnectivityGabow scg(*sess);
scg.AddAllNodeTypes();
scg.AddAllEdgeTypes(Outgoing);
scg.Run();
ConnectedComponents * cc = scg.GetConnectedComponents();
for (int i = 0; i < cc->GetCount(); i++)
{
    std::cout << "# component: " << i << " size: " << cc->GetSize(i) << std::endl;
    Objects * objs = cc->GetNodes(i);
    ...
    delete objs;
}
delete cc;
```

Python

```
scg = sparksee.StrongConnectivityGabow(sess)
scg.add_all_node_types()
scg.add_all_edge_types(sparksee.EdgesDirection.OUTGOING)
scg.run()
cc = scg.get_connected_components()
for i in range(0, cc.get_count()):
    print "# component: ", i, " size: ", cc.get_size(i)
    objs = cc.get_nodes(i)
    ...
    objs.close()
cc.close()
scg.close()
```

Objective-C

```
STSStrongConnectivityGabow *scg = [[STSStrongConnectivityGabow alloc]
     initWithSession: sess];
[scg addAllNodeTypes];
[scg addAllEdgeTypes: STSOutgoing];
[scg run];
STSConnectedComponents * cc = [scg getConnectedComponents];
for (int ii = 0; ii < [cc getCount]; ii++)
{
    NSLog(@"# component: %d size: %lld\n", ii, [cc getSize: ii]);
    STSObjects * objs = [cc getNodes: ii];
    ...
    [objs close];
}
[cc close];
[scg close];
//[scg release];
```

Community detection

Detecting communities is a common problem in graph theory. A community is a subgraph where the set of nodes are densely connected.

CommunityDetection is the basic class for all the different implementations. Several filters and restrictions can be specified in order to find the communities:

- *Restrict which node types can be traversed:* the user can specify which node types are valid for the computation, so nodes belonging to other node types will be ignored.
- *Exclude a collection of node identifiers:* the user can specify a collection of node identifiers to be ignored during the computation.
- *Exclude a collection of edge identifiers:* the user can specify a collection of edge identifiers to be ignored during the computation.

In addition the abstract class, DisjointCommunityDetection inherits from 'CommunityDetection' and also adds the specific operations for not overlapping community detection:

- *Attribute for the materialization of the computation:* the user can specify an attribute where the results of the computation are persistently stored. The identifier of the community will be stored for each node as the value of that attribute. The community identifier is a numeric value in the range $[0..C-1]$ where C is the number of different communities. Note that the results are not automatically updated in the case that the graph topology is updated (node or edge objects added or removed).

The only community detection algorithm currently implemented is an algorithm for undirected graphs. As a result, the operations to set the valid EdgeTypes does not have a direction argument. All the added types will be used in both directions even when the edge type is directed.

- CommunitiesSCD class solves the not overlapping community detection problem for undirected graphs. This class provides methods for restricting which edge types can be traversed during the computation. The implementation for this class uses the Scalable Community Detection algorithm based on the paper High quality, scalable and parallel community detection for large real graphs by Arnau Prat-Perez, David Dominguez-Sal, Josep-Lluis Larriba-Pey - WWW 2014.

Additionally the DisjointCommunities class helps the user manage the result of a disjoint community detection algorithm. It retrieves the number of communities, the community identifier for each node, and the size of a certain community or all the elements in a community.

All the community classes and subclasses have a close method in order to free resources as soon as the instances are no longer in use (deleted in the case of Sparkseecpp).

Java

```
CommunitiesSCD commSCD = new CommunitiesSCD(sess);
commSCD.addAllEdgeTypes();
commSCD.addAllNodeTypes();
commSCD.run();
DisjointCommunities dcs = commSCD.getCommunities();
for (long ii = 0; ii < dcs.getCount(); ii++)
{
    System.out.println("Community "+ii+" has "+dcs.getSize(ii)+" nodes.");
```

```
    Objects dcsNodes = dcs.getNodes(ii);
    ...
    dcsNodes.close();
}
dcs.close();
commSCD.close();
```

[C#]

```
CommunitiesSCD commSCD = new CommunitiesSCD(sess);
commSCD.AddAllEdgeTypes();
commSCD.AddAllNodeTypes();
commSCD.Run();
DisjointCommunities dcs = commSCD.GetCommunities();
for (long ii = 0; ii < dcs.GetCount(); ii++)
{
    System.Console.WriteLine("Community "+ii+" has "+dcs.GetSize(ii)+" nodes.");
    Objects dcsNodes = dcs.GetNodes(ii);
    ...
    dcsNodes.Close();
}
dcs.Close();
commSCD.Close();
```

C++

```
CommunitiesSCD commSCD(*sess);
commSCD.AddAllEdgeTypes();
commSCD.AddAllNodeTypes();
commSCD.Run();
DisjointCommunities *dcs = commSCD.GetCommunities();
for (sparksee::gdb::int64_t ii = 0; ii < dcs->GetCount(); ii++)
{
    std::cout << "# community: " << ii << " size: " << cc->GetSize(ii) << std::
        endl;
    Objects *dcsNodes = dcs->GetNodes(ii);
    ...
    delete dcsNodes;
}
delete dcs;
```

Python

```
commmSCD = sparksee.CommunitiesSCD(sess)
commmSCD.add_all_edge_types()
commmSCD.add_all_node_types()
commmSCD.run()
dcs = commmSCD.get_communities()
for ii in range(0, dcs.get_count()):
    print "Community ", ii, " has ", dcs.get_size(ii), " nodes."
    dcsNodes = dcs.get_nodes(ii)
    ...
    dcsNodes.close()
dcs.close()
commmSCD.close()
```

Objective-C

```
STSCommunitiesSCD *commSCD = [[STSCommunitiesSCD alloc] initWithSession: sess];
[commSCD addAllEdgeTypes];
```

```
[commSCD addAllNodeTypes];
[commSCD run];
STSDisjointCommunities *dcs = [commSCD getCommunities];
for (long long ii = 0; ii < [dcs getCount]; ii++)
{
    NSLog(@"# Community: %d size: %lld\n", ii, [dcs getSize: ii]);
    STSObjects *dcsNodes = [dcs getNodes: ii];
    ...
    [dcsNodes close];
}
[dcs close];
[commSCD close];
//[commSCD release];
```

Configuration

Sparksee allows some configuration variables to be set in order to update or monitorize the behavior of the system, as well as to make the deployment of Sparksee-based applications easier.

Set-up

There are two alternatives for setting-up the configuration:

1. Using the SparkseeConfig class.
2. Using a properties file.

Both can even be used at the same time. In case of conflict, the settings from the SparkseeConfig class have a higher priority.

SparkseeConfig class

The SparkseeConfig class defines a setter and a getter for each of the variables that can be specified for Sparksee configuration. If not changed, all the variables are loaded with default values. All these values can be overwritten by the user calling the corresponding setter or creating a properties file.

In order to set up a configuration the user must create an instance of the SparkseeConfig class, which will have the default values. All the variables may be set if the user needs to change them. We strongly recommend understanding each of the variables prior to changing their default values:

Java

```
SparkseeConfig cfg = new SparkseeConfig();
cfg.setCacheMaxSize(2048); // 2 GB
cfg.setLogFile("HelloSparksee.log");
...
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.create("HelloSparksee.gdb", "HelloSparksee");
```

[C#]

```
SparkseeConfig cfg = new SparkseeConfig();
cfg.SetCacheMaxSize(2048); // 2 GB
cfg.SetLogFile("HelloSparksee.log");
...
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.Create("HelloSparksee.gdb", "HelloSparksee");
```

C++

```
SparkseeConfig cfg;
cfg.SetCacheMaxSize(2048); // 2 GB
cfg.SetLogFile(L"HelloSparksee.log");
...
Sparksee * sparksee = new Sparksee(cfg);
Database * db = sparksee->Create(L"HelloSparksee.gdb", L"HelloSparksee");
```

Python

```
cfg = sparksee.SparkseeConfig()
cfg.set_cache_max_size(2048) # 2 GB
cfg.set_log_file("Hellosparksee.log")
...
sparks = sparksee.Sparksee(cfg)
db = sparks.create(u"Hellosparksee.gdb", u"HelloSparksee")
```

Objective-C

```
STSSparkseeConfig *cfg = [[STSSparkseeConfig alloc] init];
[cfg setCacheMaxSize: 2048]; // 2 GB
[cfg setLogFile: @"HelloSparksee.log"];
...
STSSparksee *sparksee = [[STSSparksee alloc] initWithConfig: cfg];
//[cfg release];
STSDatabase *db = [sparksee create: @"HelloSparksee.gdb" alias: @"HelloSparksee"];
```

Properties file

As previously explained, as an alternative to SparkseeConfig methods, the user can load Sparksee configuration variables from a properties file in order to make development of their applications using Sparksee easier.

A properties file is a plain-text file where there is one line per property. Each property is defined by a key and a value as follows: key=value.

This is an example of a Sparksee configuration properties file:

```
sparksee.license="XXXXX-YYYYY-ZZZZZ-PPPPP"
sparksee.io.cache.maxsize=2048
sparksee.log.file="HelloSparksee.log"
```

By default, SparkseeConfig tries to load all the variables defined in the ./sparksee.cfg file (in the execution directory). If the user has not created this file, the default values will be assumed.

The SparkseeProperties class also contains a method for specifying a different file name and path to be loaded.

To load Sparksee configuration variables from a different file rather than the default, the method load must be called before the SparkseeConfig class is instantiated, as we can see in the following examples where the mysparksee.cfg file is used to load Sparksee configuration variables:

Java

```
SparkseeProperties.load("sparksee/config/dir/mysparksee.cfg");
SparkseeConfig cfg = new SparkseeConfig();
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.create("HelloSparksee.gdb", "HelloSparksee");
```

[C#]

```
SparkseeProperties.Load("sparksee/config/dir/mysparksee.cfg");
SparkseeConfig cfg = new SparkseeConfig();
Sparksee sparksee = new Sparksee(cfg);
Database db = sparksee.Create("HelloSparksee.gdb", "HelloSparksee");
```

C++

```
SparkseeProperties::Load(L"sparksee/config/dir/mysparksee.cfg");
SparkseeConfig cfg;
Sparksee * sparksee = new Sparksee(cfg);
Database * db = sparksee->Create(L"HelloSparksee.gdb", L"HelloSparksee");
```

Python

```
sparksee.SparkseeProperties.load("sparksee/config/dir/mysparksee.cfg");
cfg = sparksee.SparkseeConfig()
sparks = sparksee.Sparksee(cfg)
db = sparks.create(u"Hellosparksee.gdb", u"HelloSparksee")
```

Objective-C

```
[STSSparkseeProperties load: @"sparksee/config/dir/mysparksee.cfg"];
STSSparkseeConfig *cfg = [[STSSparkseeConfig alloc] init];
STSSparksee *sparksee = [[STSSparksee alloc] initWithConfig: cfg];
//[cfg release];
STSDatabase *db = [sparksee create: @"HelloSparksee.gdb" alias: @"HelloSparksee"];
```

Variables

Here is the list of the existing Sparksee configuration variables divided by categories, with an identifier name, a description, the valid format for the value, the default value and the corresponding method to set each variable in the SparkseeConfig class. Please note that the identifier name is the key to be used in the properties file.

Valid formats for the variables values are:

- A string, optionally quoted.
- A number.
- A *time unit*: <X>[D|H|M|S|s|m|u] where:

  - <X> is a number
  - Followed by an optional character which represents the unit: D for days, H for hours, M for minutes, S or s for seconds, m for milliseconds and u for microseconds. If no unit character is given, seconds are assumed.

License

- **sparksee.license**: License user code. If no license code is provided, by default the evaluation restrictions are in place.

  Value format: String. Default value: ""

  This variable can be set by calling SparkseeConfig#setLicense

Log

Details about logging capabilities & value information are given in the 'Logging' section of the 'Maintenance and monitoring' chapter.

- **sparksee.log.level**: logging level. Allowed values are (case insensitive): off, severe, warning, info, config, fine.

  Value format: String. Default value: info.

  This variable can be set by calling SparkseeConfig#setLogLevel

- **sparksee.log.file**: log file. This is ignored when the log level is 'off'.

  Value format: String. Default value: "sparksee.log"

  This variable can be set by calling SparkseeConfig#setLogFile

Rollback

Details about the use of transactions in Sparksee are given in the 'Transactions' section of the 'Graph Database' chapter.

- **sparksee.io.rollback**: enable ("true") or disable ("false") the rollback functionality.

  Value format: String. Default value: "true"

  This variable can be set by calling SparkseeConfig#setRollbackEnabled

Recovery

Details about the recovery module are given in the 'Recovery' section of the 'Maintenance and monitoring' chapter.

- **sparksee.io.recovery**: enable ("true") or disable ("false") the recovery functionality. If disabled, all the other related variables are ignored.

  Value format: String. Default value: "false"

  This variable can be set by calling SparkseeConfig#setRecoveryEnabled

- **sparksee.io.recovery.logfile**: recovery log file. Empty string means that the recovery log file will be the same as the database path with the format ".log". For example, if the database is at "database.gdb" then the recovery log file will be "database.gdb.log".

  Value format: String. Default value: ""

  This variable can be set by calling SparkseeConfig#setRecoveryLogFile

- **sparksee.io.recovery.cachesize**: maximum size for the recovery cache in extents.

  Value format: number. Default value: 256 (for the default extent size)

  This variable can be set by calling SparkseeConfig#setRecoveryCacheMaxSize

- **sparksee.io.recovery.checkpointTime**: checkpoint frequency for the recovery cache.

  Value format: time unit. Values must be greater than or equal to 1 second. Default value: 60

  This variable can be set by calling SparkseeConfig#setRecoveryCheckpointTime

Storage

- **sparksee.storage.extentsize**: extent size in KB. An extent is the unit for I/O operations. It must be the same for the whole life of the database. Therefore, it can be set when the database is created and it cannot be updated subsequently. Recommended sizes are: 4, 8, 16, 32 or 64. Note large extent sizes will have a penalty performance if recovery is enabled.

  Value format: number. Default value: 4

  This variable can be set by calling SparkseeConfig#setExtentSize

- **sparksee.storage.extentpages**: number of pages per extent. A page is the logical unit of allocation size for the internal data structures of the system. Therefore, it can be set when the database is created and it cannot be updated subsequently.

  Value format: number. Default value: 1

  This variable can be set by calling SparkseeConfig#setExtentPages

Cache

The cache is split into different areas, the most important being for the persistent data whilst the rest are for the running user's sessions. The size of these areas is automatic, dynamic and adaptive depending on the requirements of each of the parts. Thus, there is constant negotiation between them. Although default values might be the best configuration, the user can adapt them for very specific requirements, taking into account the fact that negotiation between areas only happens when the pool is using memory in the range [minimum, maximum]. Therefore pools are always guaranteed to have at least the minimum memory of the configuration and never be larger than the maximum.

- **sparksee.io.cache.maxsize**: maximum size for all the cache pools in MBs (persistent pools as well as all session & other temporary pools). A value for this variable of zero means unlimited, which is in fact all the available memory (when the system starts) minus 512 MB (which is a naive estimate of the minimum memory to be used by the OS).

  Value format: number. Default value: 0

  This variable can be set by calling SparkseeConfig#setCacheMaxSize

- **sparksee.io.pool.frame.size**: number of extents per frame. Whereas the extent size determines the I/O unit, the frame size is the allocation/deallocation unit in number of extents. Thus, values higher than 1 allow the pre-allocation of extents in bulk which might make sense when loading large data sets, especially if working with a small extent size.

  Value format: number. Default value: 1

  This variable can be set by calling SparkseeConfig#setPoolFrameSize

- **sparksee.io.pool.persistent.minsize**: minimum size in number of frames for the persistent pool where zero means unlimited.

  Value format: number. Default value: 64

  This variable can be set by calling SparkseeConfig#setPoolPersistentMinSize

- **sparksee.io.pool.persistent.maxsize**: maximum size in number of frames for the persistent pool where zero means unlimited.

  Value format: number. Default value: 0

  This variable can be set by calling SparkseeConfig#setPoolPersistentMaxSize

- **sparksee.io.pool.temporal.minsize**: minimum size in number of frames for the session/temporal pools where zero means unlimited. It makes sense to have a larger minimum size in case of highly memory-consuming sessions.

  Value format: number. Default value: 16

  This variable can be set by calling SparkseeConfig#setPoolTemporaryMinSize

- **sparksee.io.pool.temporal.maxsize**: maximum size in number of frames for the session/temporal pool where zero means unlimited.

  Value format: number. Default value: 0

  This variable can be set by calling SparkseeConfig#setPoolTemporaryMaxSize

For development and debugging purposes it may be useful to enable the cache statistics. More details about this can be found in the 'Statistics' section of the 'Maintenance and monitoring' chapter.

- **sparksee.cache.statistics**: enable ("true") or disable ("false") storing of cache statistics.

  Value format: String. Default value: false

  This variable can be set by calling SparkseeConfig#setCacheStatisticsEnabled

- **sparksee.cache.statisticsFile**: file where cache statistics are stored. This is ignored when the cache statistics are disabled.

  Value format: String. Default value: statistics.log

  This variable can be set by calling SparkseeConfig#setCacheStatisticsFile

- **sparksee.cache.statisticsSnapshotTime**: frequency for storing the cache statistics. This is ignored when the cache statistics are disabled.

  Value format: time unit. Default value: 1000

  This variable can be set by calling SparkseeConfig#setCacheStatisticsSnapshotTime

SparkseeHA

High availability configuration is explained in detail in the 'High availability' chapter.

- **sparksee.ha**: enables or disables HA mode. If disabled, all other HA variables are ignored.

  Value format: String. Default value: false

  This variable can be set by calling SparkseeConfig#setHighAvailabilityEnabled

- **sparksee.ha.ip**: IP address and port for the instance. It must be given as ip:port

  Value format: String. Default value: localhost:7777

  This variable can be set by calling SparkseeConfig#setHighAvailabilityIP

- **sparksee.ha.coordinators**: comma-separated list of the ZooKeeper instances. For each instance, the IP address and the port must be given as: ip:port. Moreover, the port must correspond to the given clientPort in the ZooKeeper configuration file.

  Value format: String. Default value: “”

  This variable can be set by calling SparkseeConfig#setHighAvailabilityCoordinators

- **sparksee.ha.sync**: synchronization polling time. If 0, polling is disabled and synchronization is only performed when the slave receives a write request, otherwise the parameter fixes the frequency the slaves poll the master asking for writes. The polling timer is reset if the slaves receive a write request, as it synchronizes then.

  Value format: time unit. Default value: 0.

  This variable can be set by calling SparkseeConfig#setHighAvailabilitySynchronization

- **sparksee.ha.master.history**: master's history log. The history log is limited to a certain period of time, so writes occurring beyond that period of time will be removed from it and the master will not accept requests from those deleted Sparksee slaves. For example, in case of 12H, the master will store all write operations during the last 12 hours in the history log. It will reject requests from a slave which has not been up dated in the last 12 hours.

  Value format: time unit. Default value: 1D

  This variable can be set by calling SparkseeConfig#setHighAvailabilityMasterHistory

Basic configuration

In most cases default values are the best option. In fact, non-advanced users should consider setting only the following variables for a basic configuration:

- License (sparksee.license).
- Maximum size of the cache (sparksee.io.cache.maxsize).
- Recovery (sparksee.io.recovery, sparksee.io.recovery.logfile).
- Log (sparksee.log.level and sparksee.log.file).

Scripting

Very common operations such as creating the schema or loading data from a CSV file can alternatively be performed by using Sparksee scripts.

A Sparksee script is a plain-text file which contains a set of commands. Each of these commands is interpreted and executed by a parser. Different alternatives for processing a Sparksee script are explained in the 'Scripting' section of the 'API' chapter.

A Sparksee script has the following features:

- Although it is case sensitive, the **grammar keywords are case insensitive**.
- ** Strings can be quoted** in order to use them as grammar keywords. It is also a way to use blanks or tabs in the grammar keywords. Strings can be quoted using single-quote (') or double-quoted (") characters. Examples of quoted strings: "this is a user string", 'this is a user string'
- Everything between the '#' character and the end of line character is considered a **comment** and therefore ignored by the parser.

This chapter explains valid Sparksee script commands and gives examples for each of them.

Schema definition

To define the schema of a Sparksee database the scripts include commands in order to create every element on the schema, such as the database itself and the types and attributes.

Create and open

These are the commands to create and open a Sparksee database:

```
CREATE GDB alias INTO filename

OPEN GDB alias INTO filename
```

- alias is the name for the new Sparksee graph database.
- filename corresponds to the file path where the Sparksee graph database will be physically stored.

Note that CREATE also opens the database after its creation. Also, both commands close any other previously opened database.

Examples of use, take into account that everything after the '#' character is a description of the command:

```
open gdb "My graph database" into "database.gdb" # opens 'database.gdb'
create gdb MyGDB into '/data/mygdb.dbg' # closes 'database.gdb' and opens 'mygdb.
    dbg'
```

The rest of commands require having a database open so this should be the first step in the scripts.

Types and attributes

The user can create a new node type with the following command. Attributes may also be created in this step, or added later with the specific commands for them:

```
CREATE NODE type_name "("
    [attribute_name
        (INTEGER|LONG|DOUBLE|STRING|BOOLEAN|TIMESTAMP|TEXT)
        [BASIC|INDEXED|UNIQUE]
        [DEFAULT value],
    ...]
")"
```

- type_name is the name for the new node type.
- An optional attribute list can be specified:
  - attribute_name is the name for the new attribute.
  - A valid data type must be given.
  - An index specification is optional. Basic is assumed by default.For more details about indexes see the 'Indexing' section of the 'Graph Database' chapter.
  - A default value is also optional. A null value is assumed by default.

This is an example of a valid node type creation:

```
create node PEOPLE (
    ID string unique,
    NAME string indexed,
    BIRTH timestamp,
    AGE integer basic default 0
)
```

This is the command to create an edge type and, optionally, its attributes:

```
CREATE [UNDIRECTED] EDGE type_name
[FROM tail_node_type_name TO head_node_type_name] "("
    [attribute_name
        (INTEGER|LONG|DOUBLE|STRING|BOOLEAN|TIMESTAMP|TEXT)
        [BASIC|INDEXED|UNIQUE]
        [DEFAULT value],
    ...]
") [MATERIALIZE NEIGHBORS]"
```

Edge type creation has the same features as in the case of node types but in addition it includes the following parameters:

- UNDIRECTED sets the new edge type as undirected. If not given, directed is assumed.
- FROM tail_node_type_name TO head_node_type_name sets the new edge type as restricted from a tail or source node type to a head or destination node type. If not given, non-restricted edge type is assumed. Given node type names must have been created first.
- MATERIALIZE NEIGHBORS forces the creation of the neighbor index. For more details about indexing, check the 'Indexing' section of the 'Graph database' chapter

This is an example of a valid edge type creation:

```
create edge KNOWS
from PEOPLE to PEOPLE (
    SINCE timestamp indexed,
    WHERE string default "Barcelona, ES"
) materialize neighbors
```

An alternative to specifying attributes would be creating them afterwards, using the specific command for that:

```
CREATE ATTRIBUTE [GLOBAL|NODES|EDGES] [type_name.]attribute_name
    (INTEGER|LONG|DOUBLE|STRING|BOOLEAN|TIMESTAMP|TEXT)
    [BASIC|INDEXED|UNIQUE]
    [DEFAULT value]
```

In this case, type_name references the parent node or edge type for the new attribute. When it is omitted and the scope is not specified the attribute is created as global.

The same attributes from the previous examples could have also been created as follows:

```
create node PEOPLE
create attribute PEOPLE.ID string unique
create attribute PEOPLE.NAME string indexed
create attribute PEOPLE.BIRTH timestamp
create attribute PEOPLE.AGE  integer basic default 0

create edge KNOWS from PEOPLE to PEOPLE materialize neighbors
create edge KNOWS.SINCE timestamp indexed
create edge KNWOS.WHERE string default "Barcelona, ES"
```

And with the following commands, they could be created as attributes defined for a node (for all the nodes of the graph), edge (for all the edges of the graph) or global scope (for all nodes and edges of the graph):

```
create attribute IDENT string unique
create attribute global DESCRIPTION string unique
create attribute nodes NICKNAME string unique
```

```
create attribute edges WEIGHT double
```

In addition, the default value or the index specification can be updated after the attribute definition with the following commands:

```
SET ATTRIBUTE [GLOBAL|NODES|EDGES] [type_name.]attribute_name DEFAULT value

INDEX [type_name.]attribute_name [INDEXED|UNIQUE|BASIC]
```

For example, the default value for the previous IDENT global attribute and the index specification of the previous WHERE attribute could be updated as follows:

```
set attribute IDENT default ""
set attribute global DESCRIPTION default ""
set attribute nodes NICKNAME default ""
set attribute edges WEIGHT default 0.0

index KNOWS.WHERE basic
```

Moreover, existing node or edge types and attributes can be removed from the database with the following commands respectively:

```
DROP (NODE|EDGE) type_name

DROP ATTRIBUTE [GLOBAL|NODES|EDGES] [type_name.]attribute_name
```

Thus, we could remove all previously added types and attributes as follows:

```
drop attribute IDENT
drop attribute global DESCRIPTION
drop attribute nodes NICKNAME
drop attribute edges WEIGHT

drop attribute PEOPLE.ID
drop attribute PEOPLE.NAME
drop attribute PEOPLE.BIRTH
drop attribute PEOPLE.AGE
drop node PEOPLE

drop attribute KNOWS.SINCE
drop attribute KNWOS.WHERE
drop edge KNOWS
```

Load

Sparksee provides a couple of commands to easily load data from a CSV file into node or edge types that have been previously created in the graph.

The processing of the CSV file assumes that:

- Each row of the CSV file corresponds to a new object. Thus, each row creates a new node or edge object.
- All elements of the schema required and referenced by LOAD commands must previously exist.
- All previously existing instances and their values will not be removed from the graph.

Load nodes

This is the command to load data from a CSV file into a node type:

```
LOAD NODES file_name
    [LOCALE locale_name]
    COLUMNS column_name [alias_name], ...
    INTO node_type_name
    [IGNORE (column_name|alias_name), ....]
    [FIELDS
        [TERMINATED char]
        [ENCLOSED char]
        [ALLOW_MULTILINE [num]]]
    [FROM num]
    [MAX num]
    [MODE (ROWS|COLUMNS [SPLIT [PARTITIONS num]])]
    [LOG (OFF|ABORT|logfile)]
```

- file_name is the CSV file path.
- LOCALE allows the user to set a specific locale for the CSV file by means of a case in-sensitive string argument. Locale formats are described in the 'Data import' and 'Data export' sections of the 'API' chapter
- COLUMNS sets a list of names and optionally an alias for each column of the CSV file. For example, the first element of the list matches the first column of the CSV file, and so on. In case that the name of a column corresponds to an attribute name, that column is used to set values for that attribute, otherwise the column is ignored.

  The character '*' can be used to ignore a column, too.
- node_type_name is the name of the node type to be populated.
- IGNORE sets a list of columns to be ignored. The column can be referenced by the name or the alias.

  By default, no column is ignored.
- FIELDS allows the user to specify certain features for reading data from the CSV file:
  - TERMINATED sets the character to separate fields (columns) in the CSV file. Comma (',') character is the default value.

- ENCLOSED sets the character to quote strings. Double-quoted (‘”’) character is the default value.

  Inner quotes of a quoted string can be escaped if they are doubled. Thus, the string ”my string has a ” in the middle” can be processed if the inner quoted is doubled as follows: ”my string has a ”” in the middle”.

- ALLOW_MULTILINE specifies that a string may be in multiple lines of characters. Note that those strings must be quoted. num specifies a maximum number of lines allowed. By default, strings are considered as single-lined.

- FROM sets the index row to start loading from, skipping all text previous to that point. By default all rows are loaded. Note that the row index starts at 0, which is the default value.

- MAX sets the maximum number of rows to load. A value of 0, which is the default, means unlimited.

- MODE sets the load mode, with the following options:

  - ROWS. The CSV file is read once. This is the default value.
  - COLUMNS. The CSV file is read twice: once to create the objects and once to set the attribute values.
  - COLUMNS SPLIT. The CSV file is read N times: once to create the objects and then, once time for each attribute column.
  - COLUMNS SPLIT PARTITIONS num. The CSV file is read N times: once to create the objects and then, once for each attribute column. Also, each attribute column is logically partitioned, so data column is loaded in num partitions.

- LOG sets how the errors should be managed:

  - logfile . Errors are dumped into a file, so it does not stop the processing of the CSV file. This is the default value assuming logfile as the type name.
  - ABORT. If an error is raised it interrupts the processing of the CSV file.
  - OFF. Errors are not dumped anywhere and the processing of the CSV file is never interrupted.

With this CSV file:

```
ID; NAME; SALARY; AGE
1; name1; 1800; 18
2; name2; 1600; 16
3; name3; 2000; 20
4; name4; 2200; 22
```

The loading of “PEOPLE” nodes using Sparksee scripts would look like the example below:

```
create gdb FRIENDS into "friends.dbg"

create node PEOPLE
create attribute PEOPLE.ID string unique
create attribute PEOPLE.NAME string indexed
create attribute PEOPLE.AGE  integer basic default 0

load nodes "people.csv"
    columns ID, NAME, *, AGE
    into PEOPLE
    fields terminated ;
    from 1
    log "people.csv.log"
```

Load edges

This is the command to load data from a CSV file into an edge type:

```
LOAD EDGES file_name
    [LOCALE locale_name]
    COLUMNS column_name [alias_name], ...
    INTO node_type_name
    [IGNORE (column_name|alias_name), ....]
    WHERE
        TAIL (column_name|alias_name) = node_type_name.attribute_name
        HEAD (column_name|alias_name) = node_type_name.attribute_name
    [FIELDS
        [TERMINATED char]
        [ENCLOSED char]
        [ALLOW_MULTILINE [num]]]
    [FROM num]
    [MAX num]
    [MODE (ROWS|COLUMNS [SPLIT [PARTITIONS num]])]
    [LOG (OFF|ABORT|logfile)]
```

Most of the features are the same as in the case of loading nodes, with the exception of the following WHERE elements, which are required to specify the source (tail) and destination (head) of the edges:

- TAIL shows which column name (or the alias) from the edge corresponds to values of the specified attribute of a node (node_type_name.attribute_name).

  That value would be used to retrieve a node using this edge, so it is strongly advisable to define that node attribute as unique.

  For instance, TAIL tailColumn = PEOPLE.ID means:

  - tailColumn shows the column to be used to retrieve the tail nodes.
  - PEOPLE.ID shows the ID attribute of the PEOPLE node type.
  - When processing each row of the CSV file the value of the tailColumn is used to retrieve a PEOPLE node object using the ID attribute.

- HEAD is analogous to TAIL but in this case the retrieved node will be the head (destination) of the new edge.

This is an example of a Sparksee script to load edges from a CSV file with Sparksee scripts. It assumes that "PEOPLE" nodes have already been created:

```
1;2;2012-01-01;'Barcelona'
2;3;2011-05-01;'NYC,USA'
3;4;2012-01-01;'Paris,France'
2;1;2012-01-01;'Barcelona'
3;2;2011-05-01;'NYC,USA'
```

In the example, 'tail tailId = PEOPLE.ID' sets the value referenced by the 'TailId' column as the "PEOPLE" "ID" attribute. The value is used to find a "PEOPLE" node which is the tail of the edge (because the attribute "ID" matches). For instance, the first row created would be an edge that goes from a "PEOPLE" node with ID=2 to the "PEOPLE" node with ID=1.

```
open gdb FRIENDS into "friends.dbg"

create edge KNOWS from PEOPLE to PEOPLE materialize neighbors
create edge KNOWS.SINCE timestamp indexed
create edge KNWOS.WHERE string default "Barcelona, ES"

load edges "knows.csv"
    columns headId, tailId, SINCE, WHERE
    into KNOWS
    ignore headId, tailId
    where
        tail tailId=PEOPLE.ID
        head headId=PEOPLE.ID
    fields terminated ; enclosed '
    log off
```

Other

In order to be able to create attributes with a timestamp the following command must be specified in the scripts:

```
SET TIMESTAMP FORMAT timestamp_format_string
```

Valid format fields are:

- yyyy: Year
- yy: Year without century interpreted. Within 80 years before or 20 years after the current year. For example, if the current year is 2007, the pattern MM/dd/yy for the value 01/11/12 parses to January 11, 2012, while the same pattern for the value 05/04/64 parses to May 4, 1964.
- MM: Month [1..12]
- dd: Day of month [1..31]
- hh: Hour [0..23]
- mm: Minute [0..59]
- ss: Second [0..59]
- SSS: Millisecond [0..999]

So, valid timestamp formats may be: MM/dd/yy or MM/dd/yyyy−hh.mm. If not specified, the parser automatically tries to match any of the following timestamp formats:

- yyyy−MM−dd hh:mm:ss.SSS
- yyyy−MM−dd hh:mm:ss
- yyyy−MM−dd

Valid commands to set a timestamp would be:

```
set timestamp format "MM/dd/yy"
set timestamp format "MM/dd/yyyy-hh.mm"
```

High availability

As of version 4.7 Sparksee high-performance graph database comes with high-availability features which are best suited for those applications with large petition load.

In fact, Sparksee high-availability (SparkseeHA) enables multiple replicas working together, allowing the highest scalability for Sparksee applications.

This chapter covers the architecture for these SparkseeHA features, the configuration details for enabling it, and examples of typical usage scenarios.

SparkseeHA allows to horizontal scaling of read operations whilst writes are managed centrally. Future work on SparkseeHA will provide fault tolerance and master re-election.

SparkseeHA is a software feature which is enabled through the license. Sparksee free evaluation does not provide it by default. More information about the licenses at the 'Introduction' section.

Architecture

Design

SparkseeHA provides a horizontally scaling architecture that allows Sparksee-based applications to handle larger read-mostly workloads.

SparkseeHA has been thought to minimize developers' work to go from a single node installation to a multiple node HA-enabled installation. In fact, it does not require any change in the user application because it is simply a question of configuration.

To achieve this, several Sparksee slave databases work as replicas of a single Sparksee master database, as seen in the figure below. Thus, read operations can be performed locally on each node and write operations are replicated and synchronized through the master.

Figure 7.1 shows all components in a basic SparkseeHA installation:

- **Sparksee master**

  This is responsible for receiving write requests from a slave and redirecting them to the other slave instances. At the same time the master itself also plays the role of a slave.

  Only a single node of the cluster can be configured to be the master. The election of the master is automatically done by the coordinator service when the system starts.

  The master is in charge of the synchronization of write operations with the slaves. To do this task it manages a history log where all writes are serialized. The size of this log is limited, and it can be configured by the user.

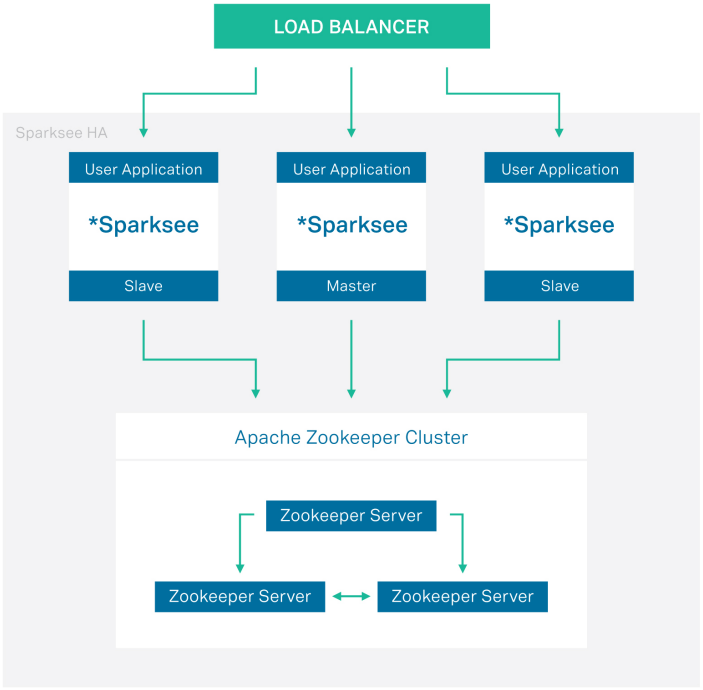


Figure 7.1: SparkseeHA Architecture

- **Sparksee slave**

  Slaves are exact replicas of the master database; they can therefore locally perform read operations on their own without requiring synchronization.

  However, for write operations the synchronization with the master in order to preserve data consistency is a must. These writes are eventually propagated from the master to other slaves. Therefore, the result of a write operation is not immediately visible in all slaves. These synchronizations are made by default made during a write operation; however there is optional polling to force synchronization that can be configured by the user.

  It is not mandatory to have a slave in the architecture, as the master can work as a standalone.

- **Coordinator service: Apache ZooKeeper**

  A ZooKeeper cluster is required to perform the coordination tasks, such as the election of the master when the system starts.

  The size of the ZooKeeper cluster depends on the number of Sparksee instances. In every case, the size of the ZooKeeper cluster must be an odd number.

  Sparksee v4.7 works with Apache ZooKeeper v3.4. All our tests have been performed using v3.4.3.

- **User application**

As Sparksee is an embedded graph database, a user application is required for each instance. As it has already been mentioned, moving to SparkseeHA mode does not require any update in the user application.

Note that the user application can be developed for all the platforms and languages supported by Sparksee. For the current version will be running on Windows, Linux or MacOSX and using Java, .NET or C++

- **Load balancer**

  The load balancer redirects the requests to each of the running applications (instances).

  The load balancer is not part of the Sparksee technology, therefore it must be provided by the user.

  In order to achieve the horizontal scalability, this redistribution of the application requests must be done efficiently. A round-robin approach would be a good starting solution but depending on the application requirements smarter solutions may be required. In fact, using existing third-party solutions is advisable.

  More information about load balancing strategies & available solutions in [this article][URL-Load_balancing].

How it works

Now that the pieces of the architecture are clear, let's see how SparkseeHA works in different scenarios or acts in typical operations using these components. Below is an explanation of how the system acts in the described situations.

Master election

The first time a Sparksee instance goes up, it registers itself into the coordinator service. The first instance registered which becomes the *master*. If a master already exists, it becomes a *slave*.

Reads

As all Sparksee slave databases are replicas of the Sparksee master database, slaves can answer read operations by performing the operation locally. They do not need to synchronize with the master.

Writes

In order to preserve data consistency, write operations require slaves to be synchronized with the master. A write operation is as follows:

1. A slave wishes to perform a write operation and sends it to the master.
2. The master serializes the operation in the history log, performs the write, and replies to the slave when it has been successfully achieved.
3. From the master the slave receives a fully updated list of write operations, which are extracted from the history log, and records them in addition to its original write. This operation preserves the consistency of the database.

If two slaves perform a write operation on the same object at the same time, it may result in a *lost update* in the same way as may happen in a Sparksee single instance installation if two different sessions want to write the same object at the same time.

Slave goes down

A failure in a slave during a regular situation does not affect the rest of the system. However if it goes down in the middle of a write operation the behavior of the rest of the system will depend on the use of transactions:

- If we are in an **auto-committed** mode (the user does not explicitly start/end transactions), the system remains operational when the slave fails.
- If the write operation **is enclosed within a transaction**, the master will not be able to *rollback* the operation and will therefore be blocked for any more write operations. This behavior can be explained by the fact that the master keeps waiting to finish the transaction, which will never be received as the slave has crashed.

Slave goes up

When a Sparksee instance goes up, it registers itself with the coordinator. The instance will become a slave if there is already a master in the cluster.

If **polling** is **enabled** for the slave, it will immediately synchronize with the master to receive all pending writes. On the other hand, if **polling** is **disabled**, the slave will synchronize when a write is requested (as explained previously).

Future work

This is a first version of SparkseeHA, so although it is fully operational some important functionality is not available which will assure a complete high-availability of the system. Subsequent versions will focus on the following features:

Master goes down

A failure in the master leaves the system non-operational. In future versions this scenario will be correctly handled automatically converting one of the slaves into a master.

Fault tolerance

A failure during the synchronization of a write operation between a master and a slave leaves the system non-operational. For instance, a slave could fail during the performance of a write operation enclosed in a transaction, or there could be a general network error.

This scenario requires that the master should be able to abort (*rollback*) a transaction. As Sparksee does not offer that functionality, these scenarios cannot currently be solved. SparkseeHA will be able to react when Sparksee implements the required functionality.

Configuration

Installation

A complete installation includes all the elements previously described in the architecture: Sparksee (SparkseeHA configuration), the coordination service (ZooKeeper) and the load balancer. The last one is beyond the scope of this document because, as has been previously stated, it is developers' decision which is the best to use for their specific system.

SparkseeHA is included in all distributed Sparksee packages. Thus, it is not necessary to install any extra package to make the application HA-enabled it is only a matter of configuration. Sparksee can be downloaded as usual from Sparsity's website. Use Sparksee to develop your application. Plus, visit Sparksee documentation site to learn how to use Sparksee.

SparkseeHA requires Apache ZooKeeper as the coordination service. Latest version of ZooKeeper v3.4.3 should be downloaded from their website. Once downloaded, it must be installed on all the nodes of the cluster where the coordination service will run. Please note that Apache ZooKeeper requires Java to work, we recommend consulting the Apache ZooKeeper documentation for requirements details.

ZooKeeper

The configuration of Apache ZooKeeper can be a complex task, so we refer the user to the Apache ZooKeeper documentation for more detailed instructions.

This section does, however, cover the configuration of the basic parameters to be used with SparkseeHA, to serve as an introduction for the configuration of the ZooKeeper.

Basic ZooKeeper configuration can be performed in the $ZOOKEEPER_HOME /conf/zoo.cfg file. This configuration file must be installed on each of the nodes which is part of the coordination cluster.

- **clientPort**: This is the port that listens for client connections, to which the clients attempt to connect.
- **dataDir**: This shows the location where ZooKeeper will store the in-memory database snapshots and, unless otherwise specified, the transaction log of updates to the database. Please be aware that the device where the log is located strongly affects the performance. A dedicated transaction log device is a key to a consistently good performance.
- **tickTime**: The length of a single tick, which is the basic time unit used by ZooKeeper, as measured in milliseconds. It is used to regulate heartbeats, and timeouts. For example, the minimum session timeout will be two ticks.
- **server.x=[hostname]:nnnnn[:nnnnn]**: There must be one parameter of this type for each server in the ZooKeeper ensemble. When a server goes

up, it determines which server number it is by looking for the myid file in the data directory. This file contains the server number in ASCII, and should match the x in server.x of this setting. Please take into account the fact that the list of ZooKeeper servers used by the clients must exactly match the list in each one of the Zookeper servers.

For each server there are two port numbers nnnnn. The first port is mandatory because it is used for the Zookeeper servers, assigned as followers, to connect to the leader. However, the second one is only used when the leader election algorithm requires it. To test multiple servers on a single machine, different ports should be used for each server.

This is an example of a valid $ZOOKEEPER_HOME/conf/zoo.cfg configuration file:

```
tickTime=2000
dataDir=/var/lib/zookeeper/
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

SparkseeHA

As previously explained, enabling HA in a Sparksee-based application does not require any update of the user's application nor the use of any extra packages. Instead, just a few variables must be defined in the Sparksee configuration.

- **sparksee.ha**: Enables or disables HA mode.

  Default value: false

- **sparksee.ha.ip**: IP address and port for the instance. This must be given as follows: ip:port.

  Default value: localhost:7777

- **sparksee.ha.coordinators**: Comma-separated list of the ZooKeeper instances. For each instance, the IP address and the port must be given as follows: ip:port. Moreover, the port must correspond to that given as clientPort in the ZooKeeper configuration file.

  Default value: ""

- **sparksee.ha.sync**: Synchronization polling time. If 0, polling is disabled and synchronization is only performed when the slave receives a write request, otherwise the parameter fixes the frequency the slaves poll the master asking for writes. The polling timer is reset if the slave receives a write request, at that moment it is (once again) synchronized.

  The time is given in *time-units*: <X>[D|H|M|S|s|m|u] where <X> is a number followed by an optional character representing the unit; D for days, H

for hours, M for minutes, S or s for seconds, m for milliseconds and u for microseconds. If no unit character is given, seconds are assumed.

Default value: 0

- **sparksee.ha.master.history**: The history log is limited to a certain period of time, so writes occurring after that period of time will be removed and the master will not accept requests from those deleted Sparksee slaves. For example, in case of 12H, the master will store in the history log all write operations performed during the previous 12 hours. It will reject requests from a slave which has not been updated in the last 12 hours.

  This time is given in *time-units*, as with the previous variable.

  Default value: 1D

Please, take into account the fact that slaves should synchronize before the master's history log expires. This will happen if the write ratio of the user's application is high enough, otherwise you should set a polling value, which must be shorter than the master's history log time.

These variables must be defined in the Sparksee configuration file (sparksee.cfg) or set using the SparkseeConfig class. More details on how to configure Sparksee can be found on the documentation site.

Example

Figure 7.2 is an example of a simple SparkseeHA installation containing:

- **HAProxy** as the **load balancer** which redirects application requests to the replicated user application instances.
- 2 Apache Tomcat servers running a **web Java user application**. In this case, those applications would be using Sparkseejava. Both servers run the same user application as they are replicas.
- A single-node **ZooKeeper coordinator service**.

HAProxy

HAProxy is a free, fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. Check their documentation site for more details about the installation and configuration of this balancer.

The configuration file for the example would look like this:

```
global
    daemon
    maxconn 500

defaults
    mode http
    timeout connect 10000ms
    timeout client 50000ms
```

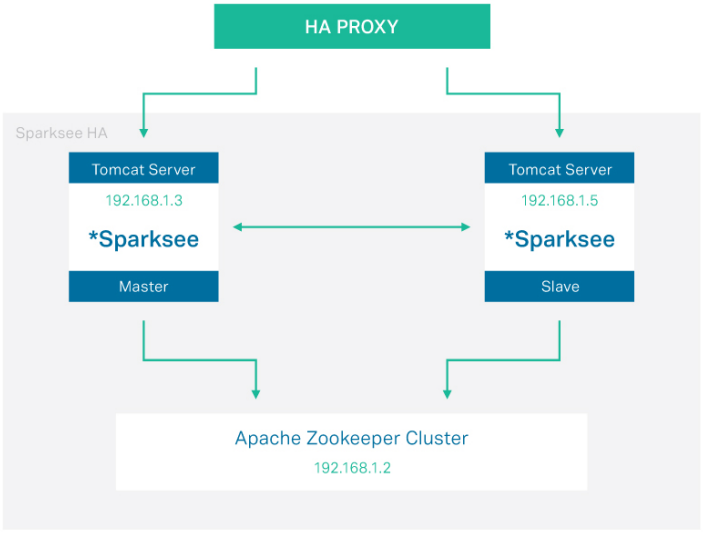


Figure 7.2: SparkseeHA example

```
    timeout server 50000ms

frontend http-in
    bind *:80
    default_backend sparksee

backend sparksee
    server s1 192.168.1.3:8080
    server s2 192.168.1.5:8080

listen admin
    bind *:8080
    stats enable
```

ZooKeeper

In this example, the $ZOOKEEPER_HOME/conf/zoo.cfg configuration file for the ZooKeeper server would be:

```
tickTime=2000
dataDir=$ZOOKEEPER_HOME/var
clientPort=2181
initLimit=10
syncLimit=5
```

Please note that, as it is running a single-node ZooKeeper cluster, `server.x` variable is not necessary.

SparkseeHA

The Sparksee configuration file for the first instance (the master) would look like this:

```
sparksee.ha=true
sparksee.ha.ip=192.168.1.3:7777
sparksee.ha.coordinators=192.168.1.2:2181
sparksee.ha.sync=600s
sparksee.ha.master.history=24H
```

And this would be the content for the file in the second instance (the slave):

```
sparksee.ha=true
sparksee.ha.ip=192.168.5.3:7777
sparksee.ha.coordinators=192.168.1.2:2181
sparksee.ha.sync=600s
sparksee.ha.master.history=24H
```

The only difference between these two files is the value of the sparksee.ha.ip variable.

As seen in the ['Architecture' chapter][doc:Architecture] the role of the master is given to the first starting instance, so to make sure the instance master is that designated in the example, the order of the operations is as follows:

1. Start the master server by starting first the server with the 192.168.1.3 IP address.
2. Once the master has been started, start all the slave instances.
3. Finally, start the HAProxy.

Likewise, to shut down the system it is highly recommended that the slaves are stopped first, followed by the master.

Maintenance and monitoring

In this chapter the database administrator can learn about the functionalities that Sparksee offers in order to maintain and monitorize Sparksee databases.

We would like to place particular emphasis on the Recovery functionality that will help the administrator to always keep an automatic copy of the database stored and safe.

Backup

Sparksee provides functionality for performing a cold backup and restoring a database which has been previously backed up.

During a cold backup, the database is closed or locked and not available to users. The data files do not change during the backup process so the database is in a consistent state when it is returned to normal operation.

The method Graph#backup performs a full backup by writing all the content of the database into a given file path and Sparksee#restore creates a new Database instance from a backup file.

Next code-blocks provide an example of this functionality:

Java

```
// perform backup

Graph graph = sess.getGraph();
...
graph.backup("database.gdb.back");
...
sess.close();

// restore backup

Sparksee sparksee = new Sparksee(new SparkseeConfig());
Database db = sparksee.restore("database.gdb", "database.gdb.back");
Session sess = db.newSession();
Graph graph = sess.getGraph();
...
sess.close();
db.close();
sparksee.close();
```

[C#]

```
// perform backup

Graph graph = sess.GetGraph();
...
graph.Backup("database.gdb.back");
...
sess.Close();

// restore backup

Sparksee sparksee = new Sparksee(new SparkseeConfig());
Database db = sparksee.Restore("database.gdb", "database.gdb.back");
Session sess = db.NewSession();
Graph graph = sess.GetGraph();
```

```
...
sess.Close();
db.Close();
sparksee.Close();
```

C++

```
// perform backup

Graph * graph = sess->getGraph();
...
graph->Backup(L"database.gdb.back");
...
delete sess;

// restore backup

SparkseeConfig cfg;
Sparksee * sparksee = new Sparksee(cfg);
Database * db = sparksee.Restore(L"database.gdb", L"database.gdb.back");
Session * sess = db->NewSession();
Graph * graph = sess->GetGraph();
...
delete db;
delete sess;
delete sparksee;
```

Python

```
# perform backup

graph = sess.get_graph()
...
graph.backup("database.gdb.back")
...
sess.close;

# restore backup

sparks = sparksee.Sparksee(sparksee.SparkseeConfig())
db = sparks.restore("database.gdb", "database.gdb.back")
sess = db.new_session()
graph = sess.get_graph()
...
db.close()
sess.close()
sparks.close()
```

Objective-C

```
// perform backup
STSGraph * graph = [sess getGraph];
...
[graph backup: @"database.gdb.back"];
...
[sess close];
[db close];
[sparksee close];
//[sparksee release];

// restore backup
STSSparkseeConfig * cfg = [[STSSparkseeConfig alloc] init];
STSSparksee * sparksee = [[STSSparksee alloc] initWithConfig: cfg];
//[cfg release];
```

```
STSDatabase * db = [sparksee restore: @"database.gdb" backupFile: @"database.gdb.
    back"];
STSSession * sess = [db createSession];
STSGraph * graph = [sess getGraph];
...
[sess close];
[db close];
[sparksee close];
//[sparksee release];
```

Note that OIDs (object identifiers) for both node and edge objects will be the same when the database is restored, however type or attribute identifiers may differ.

Take into consideration that although it does not update the database it works as a writing method. As Sparksee's concurrency model only accepts 1 writer transaction at a time (see more details about this in the 'Processing' section of the 'Graph database' chapter), this operation blocks any other transaction.

Recovery

Sparksee includes an automatic recovery manager which keeps the database safe for any eventuality. In case of application or system failures, the recovery manager is able to bring the database to a consistent state in the next restart.

By default, the recovery functionality is disabled so in order to use it, the user must enable and configure the manager. The recovery manager introduces a small penalty in the performance, so there is always a trade-off between the functionality it provides and a minor decrease in performance.

The configuration includes:

- **Log file:** the recovery log file stores all data pages that have not been flushed to disk. It is used for the recovery in the next restart after a failure.
- **Cache:** the maximum size for the recovery cache. Some parts of the recovery log file are stored in this cache. In case the cache is too small some extra I/O will be required. Anyway, a small cache should be enough to work properly.
- **Checkpoint:** the checkpoint frequency for the recovery cache. A checkpoint is a safe point which guarantees database consistency. On one hand a high frequency increases the number of writes to disk slowing the process. On the other, a low frequency requires a larger recovery log file and increases the risk of lost information.

This configuration can be performed with the SparkseeConfig class or by setting the values in a Sparksee configuration file. This is explained in detail in the 'Recovery' section of the 'Configuration' chapter.

Runtime information

Logging

It is possible to enable the logging of Sparksee activity. The log configuration requires both the level and the log file path.

This configuration can be performed with the SparkseeConfig class or by setting the values in a Sparksee configuration file. This is explained in detail in the 'Log' section of the 'Configuration' chapter.

Current valid Sparksee log levels are defined in the LogLevel enum class. This is the list of values ordered from the least verbose and increasing:

1. Off

   Log is disabled.

2. Severe

   The log only stores errors.

3. Warning

   Log errors and situations which may require special attention are included in the log file.

4. Info

   Log errors, warnings and information messages are always stored.

5. Config

   Log includes configuration details of the different components.

6. Fine

   This is the most complete log level; it includes the previous levels of logging plus additional platform details.

7. Debug

   Log debug information. It only works for a debug version of the library, so it can only be used by developers.

Dumps

There are two methods to dump a summary of the content from a Sparksee database.

- Graph#dumpData writes a summary of the user's data. It contains attributes and values for each of the database objects as well as other type of user-oriented information.
- Graph#dumpStorage writes a summary of the internal structures. This type of dump is useful for developers.

Both files are written using YAML, a human-readable data serialization format.

Statistics

Sparksee offers a set of runtime statistics available for different Sparksee components. In order to use each statistical method it is recommended checking the class in the reference manuals of the chosen programming language.

Database statistics The class DatabaseStatistics provides general information about the database:

- Database size.
- Temporary storage database size.
- Current number of concurrent sessions.
- Cache size.
- Total read data since the start.
- Total write data since the start.

Use the Database#getStatistics method to retrieve this information.

Platform statistics The class PlatformStatistics provides general information about the platform where Sparksee is running:

- Physical memory size.
- Free physical memory size.
- Number of CPUS.
- The *epoch time*.
- CPU user and system time.

Use the Platform#getStatistics method to retrieve this information.

Attribute statistics The class AttributeStatistics provides information about a certain attribute:

- Number of distinct values.
- Number of objects with null and non-null values.
- Minimum and maximum values.
- Mode value and the number of objects having the mode value.

For numerical attributes (integer, long and double) it also includes:

- Mean value.
- Variance.
- Median.

For string attributes it also includes:

- Maximum length.
- Minimum length.
- Average length.

Use the Graph#getAttributeStatistics method to retrieve this information. The user should take into account the fact that the method has a boolean argument in order to specify if basic (TRUE value) or complete statistics (FALSE value) for that datatype must be retrieved. Check in the reference manual for those statistics which are considered to be basic.

The administrator may also want to check which attributes have a value in a certain range, in which case the method Graph#getAttributeIntervalCount would be the most appropriate.

Note that both methods do not work for *Basic* attributes, statistics can only be retrieved for *Indexed* or *Unique* attributes. See 'API' chapter for more details on the attribute types.

Cache statistics Finally, it is also possible to enable the logging of the cache to monitorize its activity. By default, the logging of the cache is disabled, so it should be enabled and configured first. This configuration can be performed with the SparkseeConfig class or by setting the values in a Sparksee configuration file. This is explained in detail in the 'Log' section of the 'Configuration' chapter.

The configuration of the cache statistics includes:

- The output cache statistics log file. This is a CSV file where columns are separated by semicolons so it can be easily exported to a spreadsheet to be processed.
- Some statistics are reset for each *snapshot*. The frequency of snapshots can be defined by the user.

The cache statistics log includes:

- General platform statistics.
- General database statistics.
- Group statistics.

  With regards to functionality, Sparksee internally groups data pages into different groups. For each of the available groups, there is the following set of statistics available:

- Number of requests and hits.
- Number of reads and writes.
- Number of pages and cached pages.

These statistics are duplicated: for persistent and temporary data.
Note that group 0 has the accumulated results from the rest of groups.

Third party tools

TinkerPop

TinkerPop is a software developer community providing open source software products in the area of graphs.

Software from this community includes:

- Blueprints: Blueprints is a property graph model interface. It provides implementations, test suites, and supporting extensions. Graph databases and frameworks that implement the Blueprints interfaces automatically support Blueprints-enabled applications. Likewise, Blueprints-enabled applications can plug-and-play different Blueprints-enabled graph back-ends.
- Pipes: Pipes is a dataflow framework that enables the splitting, merging, filtering, and transformation of data from input to output. Computations are evaluated in a memory-efficient, lazy fashion.
- Gremlin: Gremlin is a domain specific language for traversing property graphs. This language has application in the areas of graph query, analysis, and manipulation.
- Frames: Frames exposes the elements of a Blueprints graph as Java objects. Instead of writing software in terms of vertices and edges, with Frames, software is written in terms of domain objects and their relationships to each other.
- Furnace: Furnace is a property graph algorithms package. It provides implementations for standard graph analysis algorithms that can be applied to property graphs in meaningful ways.
- Rexster: Rexster is a multi-faceted graph server that exposes any Blueprints graph through several mechanisms with a general focus on REST.

As this is a software stack and Blueprints is in the bottom, those vendors providing an implementation for the Blueprints API are able to enable other elements from the stack for its users. Together with Sparksee, other graph vendors such as Neo4j, OrientDB, InfiniteGraph, Titan, MongoDB or Oracle NoSQL also provide an implementation for Blueprints.

The following sections describe the particularities of using some of the previously described TinkerPop software with Sparksee.

Blueprints

Blueprints is a collection of interfaces, implementations, ouplementations, and test suites for the property graph data model. Blueprints would be the same as JDBC is for relational databases but for graph databases. It provides a common

set of interfaces to allow developers to plug-and-play their graph database backend. Moreover, software written on top of Blueprints works over all Blueprints-enabled graph databases.

SparkseeGraph is the Sparksee-based implementation of the Blueprints Graph base interface. Specifically it implements the following interfaces from Blueprints:

- KeyIndexableGraph
- TransactionalGraph
- MetaGraph<com.sparsity.sparksee.gdb.Graph>

To use Sparksee and Blueprints in a Maven-based application the user only has to add the following dependency to the Maven configuration file (pom.xml):

```
<dependency>
    <groupId>com.tinkerpop.blueprints</groupId>
    <artifactId>blueprints-sparksee-graph</artifactId>
    <version>X.X.X (*) </version>
</dependency>
```

(*) You should check Blueprint's current version. Sparksee's implementation is always up to date to the latest one.

When using Sparksee's implementation some aspects regarding the type management, the sessions and the collections may be taken into account to obtain the maximum performance or to be able to use Sparksee-exclusive functionality.

Type management There are some differences between the Blueprints property graph model and the Sparksee's graph model:

- In Sparksee, node and edge are typed whereas in Blueprints, just edges have a label.
- In Sparksee, attributes are defined within the scope of a specific node or edge type, scoped to all vertex objects or edge objects or global (for all instances). More information about attributes and its scopes can be read in the API chapter. In Blueprints, attributes are defined as Vertex or Edge attributes.

SparkseeGraph solves these differences and allows any Blueprints application to work without needing to be conscious of this particularity. Moreover, it provides a way to get all Sparksee functionality from Blueprints as well. To work with types SparkseeGraph has a public ThreadLocal<String> label field which specifies the node type and updates the default behavior of some Blueprints APIs. It also includes the public ThreadLocal<Boolean> typeScope field which specifies if the attributes follows the blueprints property graph model or if otherwise you are restricting the attributes to specific Vertex/Edge type. For example it enables the possibility of:

- Setting the node or edge type when a node is added
- Setting the node or edge type when accessing an indexed key
- Setting the attribute scope when an attribute is created

Take into account that the attribute mode in Blueprints is mutually exclusive for example you can not see specific Vertex/Edge types if you are working with the pure blueprints property graph model. Thus, they can share the same name but contain different values.

Here is an example of use of the blueprints implementation for the creation of nodes that are of the type "PEOPLE":

```
KeyIndexableGraph graph = new SparkseeGraph("blueprints_test.gdb");

Vertex v1 = graph.addVertex(null);
assert v1.getProperty(StringFactory.LABEL).equals(SparkseeGraph.
     DEFAULT_SPARKSEE_VERTEX_LABEL));

((SparkseeGraph) graph).label.set("people");
Vertex v2 = graph.addVertex(null);
assert v2.getProperty(StringFactory.LABEL).equals("people");
Vertex v3 = graph.addVertex(null);
assert v3.getProperty(StringFactory.LABEL).equals("people");
// v2 and v3 are two new vertices for the 'people' node type

((SparkseeGraph) graph).label.set("thing");
Vertex v4 = graph.addVertex(null);
assert v4.getProperty(StringFactory.LABEL).equals("thing");
// v4 is a new vertex for the 'thing' node type

((SparkseeGraph) graph).label.set("people");
graph.createKeyIndex("name", Vertex.class);
// 'name' is defined for the 'people' node type
((SparkseeGraph) graph).label.set("thing");
graph.createKeyIndex("name", Vertex.class);
// 'name' is defined for the 'thing' node type

v2.setProperty("name", "foo");
v3.setProperty("name", "boo");
// v2 and v3 are 'people' node, so 'people/name' is set

v4.setProperty("name", "foo");
// v4 is a 'thing' node, so 'thing/name' is set

((SparkseeGraph) graph).label.set("people");
int i = 0;
for(Vertex v : graph.getVertices("name", "foo")) {
    assert v.equals(v2);
    i++;
}
assert i == 1;

((SparkseeGraph) graph).label.set("thing");
i = 0;
for(Vertex v : graph.getVertices("name", "foo")) {
    assert v.equals(v4);
    i++;
}
assert i == 1;

((SparkseeGraph) graph).label.set("people");
int i = 0;
for(Vertex v : graph.getVertices()) {
    assert v.equals(v2) || v.equals(v3);
    i++;
}
assert i == 2;

((SparkseeGraph) graph).label.set(null);
int i = 0;
```

```
for(Vertex v : graph.getVertices()) {
    assert v.equals(v1) || v.equals(v2) || v.equals(v3) || v.equals(v4);
    i++;
}
assert i == 4;

// Create a specefic type attribute
((SparkseeGraph) graph).typeScope.set(true);
// This creates the attribute name restricted to the type thing.
// It does not overwrite the attribute value foo of the Vertex attribute also
    called name.
v4.setProperty("name", "boo");
// Restore the normal property graph behaviour behaviour
((SparkseeGraph) graph).typeScope.set(false);
```

Sessions SparkseeGraph implements TransactionalGraph in order to manage sessions efficiently.

Any graph operation executed by a thread occurs in the context of a transaction (created automatically if there is not a transaction in progress), and each transaction manages its own session.

The TransactionalGraph enables concurrency of multiple threads for Sparksee's implementation, since each thread has its own private transaction, and therefore a different Sparksee session.

So, when a transaction begins it starts a Sparksee session for the calling thread. This Sparksee session will be exclusively used by the thread until the transaction stops.

```
Graph graph = new SparkseeGraph(...);
Vertex v = graph.addVertex(null); // <-- Automatically creates a Sparksee session
    and starts a transaction

//
// (...) More operations inside the transaction
//

graph.commit(); // <-- Closes Sparksee session and the transaction
```

Collections All Sparksee collections are wrapped in the SparkseeIterable class which implements the CloseableIterable interface from the Blueprints API.

Since Sparksee is a C++ native implementation its resources are not managed by the JVM heap. Thus, Sparksee-based blueprints applications should take into account that in order to obtain the maximum performance the collections must be explicitly closed. In the case the user does not explicitly close the collections, they will be automatically closed when the transaction is stopped (in fact, Sparksee session will be closed as well).

This would be an example of collections closed at the end of a transaction, which will have a penalty fee in performance:

```
for (final Vertex vertex : graph.getVertices()) {
    for (final Edge edge : vertex.getEdges(Direction.OUT)) {
        final Vertex vertex2 = edge.getVertex(Direction.IN);
```

```
        for (final Edge edge2 : vertex2.getEdges(Direction.OUT)) {
            ...
        }
    }
}
graph.commit();
```

To avoid this performance degradation, all retrieved collections from methods in the SparkseeGraph implementation should be closed as shown below:

```
CloseableIterable<Vertex> vv = (CloseableIterable<Vertex>)graph.getVertices();
for (final Vertex vertex : vv) {
    CloseableIterable<Edge> ee = (CloseableIterable<Edge>)vertex.getEdges(
         Direction.OUT);
    for (final Edge edge : ee) {
        final Vertex vertex2 = edge.getVertex(Direction.IN);
        CloseableIterable<Edge> ee2 = (CloseableIterable<Edge>)vertex2.getEdges(
             Direction.OUT);
        for (final Edge edge2 : ee2) {
            ...
        }
        ee2.close();
    }
    ee.close();
}
vv.close();
```

Gremlin

Gremlin is a graph traversal language which works over those graph databases/frameworks that implement the Blueprints property graph data model. It is a style of graph traversal that can be natively used in various JVM languages.

Installation of Gremlin is very easy: just download the version from Gremlin github, unzip and start the gremlin console from the script in the bin directory.

Once the console has been started, the user can instantiate a SparkseeGraph instance to use a Sparksee graph database through the Gremlin DSL like this:

```
$ wget http://tinkerpop.com/downloads/gremlin/gremlin-groovy-2.2.0.zip
$ unzip gremlin-groovy-2.2.0.zip
$ ./gremlin-groovy-2.2.0/bin/gremlin.sh

         \,,,/
         (o o)
-----oOOo-(_)-oOOo-----
gremlin> g = new SparkseeGraph("./graph.gdb")
==>sparkseegraph[./graph.gdb]
gremlin> g.loadGraphML('gremlin-groovy-2.2.0/data/graph-example-2.xml')
==>null
gremlin> g.V.count()
==>809
gremlin> g.E.count()
==>8049
gremlin> g.V('name', 'HERE COMES SUNSHINE').map
==>{name=HERE COMES SUNSHINE, song_type=original, performances=65, type=song}
gremlin> g.V('name', 'HERE COMES SUNSHINE').outE('written_by', 'sung_by')
==>e[3079][1053-written_by->1117]
==>e[4103][1053-sung_by->1032]
gremlin> g.V('name', 'HERE COMES SUNSHINE').outE('written_by', 'sung_by').inV.each
     {println it.map()}
[name:Hunter, type:artist]
```

```
[name:Garcia, type:artist]
gremlin> g.shutdown()
==>null
gremlin> quit
```

An alternative is to clone the source project from the original repository.

Rexster

Rexster is a multi-faceted graph server that exposes any Blueprints graph through several mechanisms with a general focus on REST. This HTTP web service provides standard low-level GET, POST, PUT, and DELETE methods, a flexible extensions model which allows plug-in like development for external services (such as ad hoc graph queries through Gremlin), server-side "stored procedures" written in Gremlin, and a browser-based interface called The Dog House. Rexster Console makes it possible to do remote script evaluation against configured graphs inside a Rexster Server.

Configuration is done in an XML file in the server side. Specifically for Sparksee it may be important to specify the location of the Sparksee configuration file (the sparksee.cfg file, here shown as sparksee.properties) to set some configuration parameters such as the license code. Use the <config−file> property as shown in the example below:

```
<graph>
  <graph-name>sparkseesample</graph-name>
  <graph-type>sparkseegraph</graph-type>
  <graph-location>/tmp/mygraph.gdb</graph-location>
  <properties>
    <config-file>sparksee-config/sparksee.properties</config-file>
  </properties>
  <extensions>...</extensions>
</graph>
```

Pacer

Pacer is a JRuby library that enables expressive graph traversals. It currently supports two major graph databases, Sparksee and Neo4j, using the Tinkerpop graphdb stack. Plus there is also a convenient in-memory graph called TinkerGraph which is part of Blueprints.

Pacer allows the user to create, modify and traverse graphs using very fast and memory-efficient stream processing.

The following example shows how to get Pacer with Sparksee installed using Homebrew on a Mac OS X platform. Please take into account the fact that the user should adapt the example in the case of using a different platform. Also note that Pacer requires having JRuby v.1.7.0 previously installed.

```
$ brew update
...
$ brew  install jruby
==> Downloading http://jruby.org.s3.amazonaws.com/downloads/1.7.1/jruby-bin-1.7.1.
    tar.gz
######################################################################## 100.0%
/usr/local/Cellar/jruby/1.7.1: 1619 files, 29M, built in 16 seconds
$ jruby -v
jruby 1.7.1 (1.9.3p327) 2012-12-03 30a153b on Java HotSpot(TM) 64-Bit Server VM
    1.6.0_37-b06-434-10M3909 [darwin-x86_64]
$ jruby -S gem install pacer-sparksee
Fetching: pacer-1.1.1-java.gem (100%)
Fetching: pacer-sparksee-2.0.0-java.gem (100%)
Successfully installed pacer-1.1.1-java
Successfully installed pacer-sparksee-2.0.0-java
2 gems installed
$ jruby -S gem list --local

*** LOCAL GEMS ***

pacer (1.1.1 java)
pacer-sparksee (2.0.0 java)
rake (0.9.2.2)
```

Once it is installed we can work directly with the JRuby interpreter as follows:

```
$ jirb
irb(main):001:0> require 'pacer-sparksee'
=> true
irb(main):002:0> sparksee = Pacer.sparksee '/tmp/sparksee_demo'
=> #<PacerGraph sparkseegraph[/tmp/sparksee_demo]
irb(main):003:0> pangloss = sparksee.create_vertex :name => 'pangloss', :type => '
    user'
=> #<V[1024]>
irb(main):004:0> okram = sparksee.create_vertex :name => 'okram', :type => 'user'
=> #<V[1025]>
irb(main):005:0> group = sparksee.create_vertex :name => 'Tinkerpop', :type => '
    group'
=> #<V[1026]>
irb(main):006:0> sparksee.v
#<V[1024]> #<V[1025]> #<V[1026]>
Total: 3
=> #<GraphV>
irb(main):007:0> sparksee.v.properties
{"name"=>"pangloss", "type"=>"user"}   {"name"=>"okram", "type"=>"user"}      {"
    name"=>"Tinkerpop", "type"=>"group"}
Total: 3
=> #<GraphV -> Obj-Map>
irb(main):008:0> sparksee.create_edge nil, okram, pangloss, :inspired
=> #<E[2048]:1025-inspired-1024>
irb(main):009:0> sparksee.e
#<E[2048]:1025-inspired-1024>
Total: 1
=> #<GraphE>
irb(main):010:0> group.add_edges_to :member, sparksee.v(:type => 'user')
#<E[3072]:1026-member-1024> #<E[3073]:1026-member-1025>
Total: 2
=> #<Obj 2 ids -> lookup -> is_not(nil)>
irb(main):011:0> sparksee.e
#<E[2048]:1025-inspired-1024> #<E[3072]:1026-member-1024>   #<E[3073]:1026-member
    -1025>
Total: 3
=> #<GraphE>
irb(main):012:0> quit
```

Check the documentation for a detailed explanation of the use of Pacer.