ruruki Documentation

Release 0

Optiver

May 18, 2017

Contents

Contents:

CHAPTER 1

Introduction

Introduction to Ruruki - In-Memory Directed Property Graph

What is Ruruki? Well the technical meaning is "it is any **tool** used to extract snails from rocks".

So **ruruki** is a in-memory directed property graph database **tool** used for building complicated graphs of anything.

Ruruki is super useful for

- Temporary lightweight graph database. Sometimes you do not want to depend on a heavy backend that requires complicated software like Java. Or you do not have root or admin access on the server you want to run the database on. With **ruruki**, you can install it in a python virtualenv and be up and running in no time.
- Proof of concept. **Ruruki** is super great for demonstrating a proof of concept with little resources, effort, and hassle.

My idea behind using a graph database is because everything is connected in some shape or form, no matter what it is. You can apply it to things like

- Linking actors -> movies -> directors.
- Linking networks, social or computer.
- Linking people to business structures, hierarchy, or responsibilities.
- Navigation.
- Mapping which snails climb over which rocks, or tools used for extraction, and so on.
- And the list goes on, and on, and on.

You just need to change your mindset on how data is linked together, represented, and related. Like Newton's third law "*For every action there is an equal and opposite re-action*", in terms of a graph with relationships, if one vertex/node is affected, there will be an impact on another node somewhere in the graph. For example, if the CEO is hit by a asteroid, who in the business are affected.

There are many similar projects/libraries out there that do the exact same as **ruruki**, but I decided to do my own graph library for the following reasons

- Other libraries lacked documentation.
  - GrapheekDB
  - NetworkX
  - graph-tool
  - python-graph
- Code was hard and complicated to read and follow.
- Others are too big and complex for the job that I needed to do.
- And lastly, I wanted to learn more about graph databases and decided writing a graph database library was the best way to wrap my head around it, and why not?

Contributing

If you would like to contribute, below are some guidelines.

- PEP8 (pylint)
- Documentation should be done on the interfaces if possible to keep it consistent.
- Unit-tests covering 100% of the code.

Versioning

Ruruki uses the Semantic Versioning scheme.

Summary

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Functionality still being worked on

- Traversing algorithms.
- Query language.
- Extensions, for example interacting with Neo4j.
- Persistence.
- Channels for publishing and subscribing.

Demo

To see an online demo of ruruki-eye follow the following link http://www.ruruki.com.

Build and Testing Status

CHAPTER 2

Tutorial

Before we start the tutorial, let first address the single most important thing - If you are reading this, *You are awesome*

Let's begin

Note: Each step in the tutorial will continue and add from the last step.

Installing ruruki

Lets first create an environment where we can install *ruruki* and use it.

- We will do this using a python virtual environment.

```
$ virtualenv-2.7 ruruki-ve
New python executable in ruruki-ve/bin/python2.7
Also creating executable in ruruki-ve/bin/python
Installing setuptools, pip...done.
```

- Install the graph database library into the newly created virtual environment.

```
$ ruruki-ve/bin/pip install ruruki
Collecting ruruki
  Downloading http://internal-index.com/prod/+f/2e6/c4263fb2b546a/ruruki.tar.gz
Installing collected packages: ruruki
  Running setup.py install for ruruki
Successfully installed ruruki
```

Creating a database

Note: Please keep in mind that the library is only installed into the virtual environment you created above, not your system-wide Python installation, so to use it you'll need to run the virtual environment's Python interpreter:

```
$ ruruki-ve/bin/python
```

- Let's start with first creating the graph.

```
>>> from ruruki.graphs import Graph
>>> graph = Graph()
```

- In order to use the *IGraph.get_or_create_vertex()* and *IGraph.get_or_create_edge()* effectively we should create some constraints to ensure uniqueness.

```
# Ensure that vertices/nodes person, book, author, and category have a
# unique name property.
>>> graph.add_vertex_constraint("person", "name")
>>> graph.add_vertex_constraint("book", "name")
>>> graph.add_vertex_constraint("author", "name")
>>> graph.add_vertex_constraint("category", "name")
```

Adding in some data

Now that we have a empty graph database, lets start adding in some data.

- Create some nodes. Because we added uniqueness constraints above, we can use the `IGrapph.get_or_create_vertex()` method to ensure we don't create duplicate vertices with the same details.

```
# add the categories
>>> programming = graph.get_or_create_vertex("category", name="Programming")
>>> operating_systems = graph.get_or_create_vertex("category", name="Operating Systems
↪")

# add some books
>>> python_crash_course = graph.get_or_create_vertex("book", title="Python Crash␣
↪Course")
>>> python_pocket_ref = graph.get_or_create_vertex("book", title="Python Pocket␣
↪Reference")
>>> how_linux_works = graph.get_or_create_vertex("book", title="How Linux Works: What␣
↪Every Superuser Should Know", edition="second")
>>> linux_command_line = graph.get_or_create_vertex("book", title="The Linux Command␣
↪Line: A Complete Introduction", edition="first")

# add a couple authors of the books above
>>> eric_matthes = graph.get_or_create_vertex("author", fullname="Eric Matthes", name=
↪"Eric", surname="Matthes")
>>> mark_lutz = graph.get_or_create_vertex("author", fullname="Mark Lutz", name="Mark
↪", surname="Lutz")
>>> brian_ward = graph.get_or_create_vertex("author", fullname="Brian Ward", name=
↪"Brian", surname="Ward")
>>> william = graph.get_or_create_vertex("author", fullname="William E. Shotts Jr.",␣
↪name="William", surname="Shotts")

# add some random people
>>> john = graph.get_or_create_vertex("person", name="John", surname="Doe")
>>> jane = graph.get_or_create_vertex("person", name="Jane", surname="Doe")
```

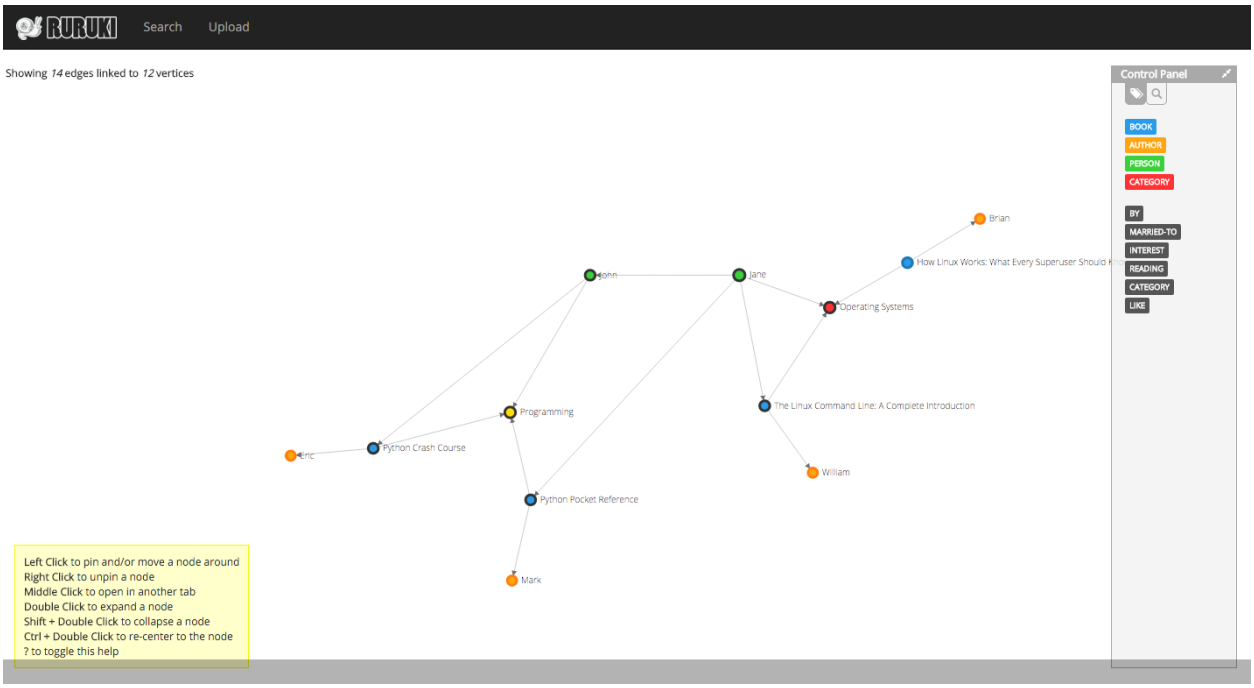
- Create a relationships between vertices created above. Again notice the use of *IGraph.get_or_create_edge()* to ensure uniqueness between the head and tails for the particular edge labels being created.

```
# link the books to a category
>>> graph.get_or_create_edge(python_crash_course, "CATEGORY", programming)
>>> graph.get_or_create_edge(python_pocket_ref, "CATEGORY", programming)
>>> graph.get_or_create_edge(linux_command_line, "CATEGORY", operating_systems)
>>> graph.get_or_create_edge(how_linux_works, "CATEGORY", operating_systems)

# link the books to their authors
>>> graph.get_or_create_edge(python_crash_course, "BY", eric_matthes)
>>> graph.get_or_create_edge(python_pocket_ref, "BY", mark_lutz)
>>> graph.get_or_create_edge(how_linux_works, "BY", brian_ward)
>>> graph.get_or_create_edge(linux_command_line, "BY", william)

# Create some arbitrary data between John and Jane Doe.
>>> graph.get_or_create_edge(john, "READING", python_crash_course)
>>> graph.get_or_create_edge(john, "INTEREST", programming)
>>> graph.get_or_create_edge(jane, "LIKE", operating_systems)
>>> graph.get_or_create_edge(jane, "MARRIED-TO", john)
>>> graph.get_or_create_edge(jane, "READING", linux_command_line)
>>> graph.get_or_create_edge(jane, "READING", python_pocket_ref)
```

Below is a visualization of the graph so far



Searching for information

Let's start searching and looking for data.

Note: The examples below only demonstrate filtering and searching on vertices, but the same operations can be

applied to edges too.

- Find all people.

```
>>> print graph.get_vertices("person").all()
[<Vertex> ident: 10, label: person, properties: {'surname': 'Doe', 'name': 'John'},
 <Vertex> ident: 11, label: person, properties: {'surname': 'Doe', 'name': 'Jane'}]
```

- Finding all help and reference books.

```
>>> result = graph.get_vertices("book", name__contains="Reference") | graph.get_
↪vertices("book", title__contains="Crash Course")
>>>> print result.all()
[<Vertex> ident: 4, label: book, properties: {'name': 'Python Pocket Reference',
↪'title': 'Python Pocket Reference'},
 <Vertex> ident: 2, label: book, properties: {'name': 'Python Crash Course', 'title':
↪'Python Crash Course'}]
```

- Finding all python books excluding crash course books.

```
>>> result = graph.get_vertices("book", name__contains="Python") - graph.get_vertices(
↪"book", title__contains="Crash Course")
>>>> print result.all()
[<Vertex> ident: 4, label: book, properties: {'name': 'Python Pocket Reference',
↪'title': 'Python Pocket Reference'}]
```

- If you already know that identity number

```
>>> print repr(graph.get_vertex(0))
<Vertex> ident: 0, label: category, properties: {'name': 'Programming'}
```

Dumping and loading data

Ruruki is an in-memory database, so all the data goes away when your program exits. However, Ruruki provides *dump()* and *load()* methods that will let you record a graph to disk and load it again later.

- Dumping your graph so that you can use it later.

```
>>> graph.dump(open("/tmp/graph.dump", "w"))
```

- Loading a dump file.

```
>>> graph.load(open("/tmp/graph.dump"))
```

Tutorial demo script

The above demo script can be found under ruruki/test_utils/tutorial_books_demo.py

CHAPTER 3

Interfaces

Graph

class `ruruki.interfaces.`**`IGraph`**

Interface for a property graph database.

`add_edge`(*head*, *label*, *tail*, ***kwargs*)

Add an directed edge to the graph.

> **Note:** If you wish to add in a undirected edge, you should add a directed edge in each direction.

Parameters

- **`head`** (*IVertex*) – Head vertex.
- **`label`** (`str`) – Edge label.
- **`tail`** (*IVertex*) – Tail vertex.
- **`kwargs`** (`str`, value.) – Property key and values to set on the new created edge.

Raises **`ConstraintViolation`** – Raised if you are trying to create a duplicate edge between head and tail.

Returns Added edge.

Return type *IEdge*

`add_vertex`(*label=None*, ***kwargs*)

Create a new vertex, add it to the graph, and return the newly created vertex.

Parameters

- **`label`** (`str` or `None`) – Vertex label.
- **`kwargs`** (`str`, value.) – Property key and values to set on the new created vertex.

Raises **ConstraintViolation** – Raised if you are adding a new vertex that violates a constraint.

Returns Added vertex.

Return type *IVertex*

add_vertex_constraint(*label*, *key*)
Add a constraint to ensure uniqueness for a particular label and property key.

Parameters

- **label** (str) – Vertex label which the constraint is meant for.
- **key** (str) – Vertex property key used to ensure uniqueness.

append_edge(*edge*)
Append the edge to the graph.

Note: The edge that you are appending to the graph should have `ident` set to `None`, so that the *IGraph* can manage what the identity number should be.

Parameters **edge** (*IEdge*) – Edge that should be appended to the graph.

Raises

- **ConstraintViolation** – Raised if you are trying to create a duplicate edge between head and tail.
- **EntityIDError** – If the edge already has a identity number set.
- **DatabaseError** – If the edge already is already bound to anther *IGraph*.

Returns The edge after it has been appended to the graph.

Return type *IEdge*

append_vertex(*vertex*)
Append the vertex to the graph.

Note: The vertex that you are appending to the graph should have `ident` set to `None`, so that the *IGraph* can manage what the identity number should be.

Parameters **vertex** (*IVertex*) – Vertex that should be appended to the graph.

Raises

- **ConstraintViolation** – Raised if you are appending a new vertex that violates a constraint.
- **EntityIDError** – If the vertex already has a identity number set.
- **DatabaseError** – If the vertex already is already bound to anther *IGraph*.

Returns The vertex after it has been appended to the graph.

Return type *IVertex*

bind_to_graph(*entity*)
Bind an entity to the graph and generate and set a unique id on the entity.

Parameters **entity** (*IEntity*) – Entity that you are binding to the graph.

Raises **UnknownEntityError** – Is raised if the entity is not a instance if a *IVertex* or *IEdge*.

close()
Close the instance.

dump(*file_handler*)
Export the database to a file handler.

Parameters

- **file_handler** – A writable file-like object; a description of this graph will be written to this file encoded as JSON data that can be read back later with *load()*.
- **file_handler** – file

get_edge(*id_num*)
Return the edge referenced by the provided object identifier.

Parameters **id_num** (int) – Edge identity number.

Returns Added edge.

Return type *IEdge*

get_edges(*head=None*, *label=None*, *tail=None*, ***kwargs*)
Return an iterable of all the edges in the graph that have a particular key/value property.

Note: See *IEntitySet.filter()* for filtering options.

Parameters

- **head** (*IVertex*) – Head vertex of the edge. If None then heads will be ignored.
- **label** (str or None) – Edge label. If None then all edges will be checked for key and value.
- **tail** (*IVertex*) – Tail vertex of the edge. If None then tails will be ignored.
- **kwargs** (str and value.) – Property key and value.

Returns *IEdge* that matched the filter criteria.

Return type *IEntitySet*

get_or_create_edge(*head*, *label*, *tail*, ***kwargs*)
Get or create a unique directed edge.

Note: If you wish to add in a unique undirected edge, you should add a directed edge in each direction.

If head or tail is a tuple, then *get_or_create_vertex()* will always be called to create the vertex.

Parameters

- **head** (*IVertex* or tuple of label str and properties dict) – Head vertex.
- **label** (str) – Edge label.

- **tail** (*IVertex* or tuple of label str and properties dict) – Tail vertex.
- **kwargs** (str, value.) – Property key and values to set on the new created edge.

Returns Added edge.

Return type *IEdge*

get_or_create_vertex(*label=None*, ***kwargs*)
Get or create a unique vertex.

Note: Constraints will always be applied first when searching for vertices.

Parameters

- **label** (str or None) – Vertex label.
- **kwargs** (str, value.) – Property key and values to set on the new created vertex.

Returns Added vertex.

Return type *IVertex*

get_vertex(*id_num*)
Return the vertex referenced by the provided object identifier.

Parameters **id_num** (int) – Vertex identity number.

Returns Vertex that has the identity number.

Return type *IVertex*

get_vertex_constraints()
Return all the known vertex constraints.

Returns Distinct label and key pairs to *add_vertex_constraint()*.

Return type Iterable of tuple of label str, key str

get_vertices(*label=None*, ***kwargs*)
Return all the vertices in the graph that have a particular key/value property.

Note: See *IEntitySet.filter()* for filtering options.

Parameters

- **label** – Vertice label. If None then all vertices will be checked for key and value.
- **label** – str or None
- **kwargs** (str and value.) – Property key and value.

Returns *IVertex* that matched the filter criteria.

Return type *IEntitySet*

load(*file_handler*)
Load and import data into the database. Data should be in a JSON format.

Note: Id's are not retained and are regenerated. This allows you to load multiple dumps into the same graph.

Parameters

- **file_handler** – A file-like object that, when read, produces JSON data describing a graph. The JSON data should be compatible with that produced by *dump()*.
- **file_handler** – file

remove_edge(*edge*)
Remove the provided edge from the graph.

Note: Removing a edge does **not** remove the head or tail vertices, but only the edge between them.

Parameters **edge** (*IEdge*) – Remove a edge/relationship.

remove_vertex(*vertex*)
Remove the provided vertex from the graph.

Parameters **vertex** (*IVertex*) – Remove a vertex/node.

Raises **VertexBoundByEdges** – Raised if you are trying to remove a vertex that is still bound or attached to another vertex via edge.

set_property(*entity*, ***kwargs*)
Set or update the entities property key and values.

Parameters **kwargs** (str, value.) – Property key and values to set on the new created vertex.

Raises

- **ConstraintViolation** – A constraint violation is raised when you are updating the properties of a entity and you already have a entity with the constrained property value.
- **UnknownEntityError** – If you are trying to update a property on a *IEntity* that is not known in the database.
- **TypeError** – If the entity that you are trying to update is not supported by the database. Property updates only support Ivertex and *IEdge*.

Base Entity

class ruruki.interfaces.**IEntity**
Base interface for a vertex/node and edge/relationship.

Note: Identity numbers are None by default. They are set by the *bind_to_graph()* when they are bound to the a graph. If using *IEntity* and *IEntitySet* without a bound graph, you will need to manually set the *ident* yourself.

IDGenerator can help you with assigning id's to vertices and edges.

as_dict(*include_privates=False*)
Return the entity as a dictionary representation.

```
>>> from pprint import pprint
>>> from ruruki.entities import Entity
>>> e = Entity("Person")
>>> e.set_property(name="Bob")
>>> e.set_property(_private_name="Sasquatch")
>>> pprint(e.as_dict()["properties"])
{'name': 'Bob'}

>>> pprint(e.as_dict(include_privates=True)["properties"])
{'_private_name': 'Sasquatch', 'name': 'Bob'}
```

Parameters **include_privates** (bool) – True to include private property keys in the dump. Private property keys are those that begin with "_".

Returns The entity as a dictionary representation.

Return type dict

is_bound()
Return True if the entity is bound to a graph.

Returns True is the entity is bound to a *IGraph*

Return type bool

remove_property(*key*)
Un-assigns a property key with its value.

Parameters **key** (str) – Key that you are removing.

set_property(***kwargs*)
Assign or update a property.

Parameters **kwargs** (key str and value.) – Key and value pairs.

Vertex

class ruruki.interfaces.**IVertex**
Interface for a vertex/node.

add_in_edge(*vertex*, *label=None*, ***kwargs*)
Add and create an incoming edge between the two vertices.

Parameters

- **vertex** (*IVertex*) – Edge the vertex is attached to.
- **label** (str) – Label for the edge being created.
- **kwargs** (str and value) – Key and values for the edges properties.

add_out_edge(*vertex*, *label=None*, ***kwargs*)
Add and create an outgoing edge between the two vertices.

Parameters

- **vertex** (*IVertex*) – Edge the vertex is attached to.

- **label** (str) – Label for the edge being created.
- **kwargs** (key str and value.) – Edges property key and value pairs.

as_dict(*include_privates=False*)

Return the entity as a dictionary representation.

```
>>> from pprint import pprint
>>> from ruruki.entities import Entity
>>> e = Entity("Person")
>>> e.set_property(name="Bob")
>>> e.set_property(_private_name="Sasquatch")
>>> pprint(e.as_dict()["properties"])
{'name': 'Bob'}

>>> pprint(e.as_dict(include_privates=True)["properties"])
{'_private_name': 'Sasquatch', 'name': 'Bob'}
```

Parameters **include_privates** (bool) – True to include private property keys in the dump. Private property keys are those that begin with "_".

Returns The entity as a dictionary representation.

Return type dict

get_both_edges(*label=None*, ***kwargs*)

Return both in and out edges to the vertex.

Parameters

- **label** (str) – Edge label. If None, all edges will be returned.
- **kwargs** (key str and value.) – Edge property key and value pairs.

Returns New *IEntitySet* with filtered entities.

Return type *IEntitySet*

get_both_vertices(*label=None*, ***kwargs*)

Return the in and out vertices adjacent to the vertex according to the edges.

Parameters

- **label** (str) – Vertices label. If None, all edges will be returned.
- **kwargs** (key str and value.) – Vertices property key and value pair.

Returns New *IEntitySet* with filtered entities.

Return type *IEntitySet*

get_in_edges(*label=None*, ***kwargs*)

Return all the in edges to the vertex.

Parameters

- **label** (str) – Edge label. If None, all edges will be returned.
- **kwargs** (key str and value.) – Edges property key and value pairs.

Returns New *IEntitySet* with filtered entities.

Return type *IEntitySet*

get_in_vertices(*label=None*, ***kwargs*)
Return the `in` vertices adjacent to the vertex according to the edge.

Parameters

- **label** (`str`) – Vertices label. If `None`, all edges will be returned.
- **kwargs** (key `str` and value.) – Vertices property key and value pairs.

Returns New *IEntitySet* with filtered entities.

Return type *IEntitySet*

get_out_edges(*label=None*, ***kwargs*)
Return all the `out` edges to the vertex.

Parameters

- **label** (`str`) – Edge label. If `None`, all edges will be returned.
- **kwargs** (key `str` and value.) – Edge property key and value pairs.

Returns New *IEntitySet* with filtered entities.

Return type *IEntitySet*

get_out_vertices(*label=None*, ***kwargs*)
Return the `out` vertices adjacent to the vertex according to the edge.

Parameters

- **label** (`str`) – Vertices label. If `None`, all edges will be returned.
- **kwargs** (key `str` and value.) – Vertices property key and value pairs.

Returns New *IEntitySet* with filtered entities.

Return type *IEntitySet*

in_edge_count()
Return the total number of in edges.

Returns Total number of `in` edges.

Return type `int`

is_bound()
Return True if the entity is bound to a graph.

Returns True is the entity is bound to a *IGraph*

Return type `bool`

out_edge_count()
Return the total number of out edges.

Returns Total number of `out` edges.

Return type `int`

remove_edge(*edge*)
Remove a *IEdge* from the vertex if it exists.

Parameters **edge** (*IEdge*) – Edge that you are removing from the vertex.

Raises **KeyError** – KeyError is raised if you are trying to remove an edge that is not found or does not exist.

remove_property(*key*)
Un-assigns a property key with its value.

Parameters **key** (str) – Key that you are removing.

set_property(***kwargs*)
Assign or update a property.

Parameters **kwargs** (key str and value.) – Key and value pairs.

Edge

class ruruki.interfaces.**IEdge**
Interface for a edge/relationship.

as_dict(*include_privates=False*)
Return the entity as a dictionary representation.

```
>>> from pprint import pprint
>>> from ruruki.entities import Entity
>>> e = Entity("Person")
>>> e.set_property(name="Bob")
>>> e.set_property(_private_name="Sasquatch")
>>> pprint(e.as_dict()["properties"])
{'name': 'Bob'}

>>> pprint(e.as_dict(include_privates=True)["properties"])
{'_private_name': 'Sasquatch', 'name': 'Bob'}
```

Parameters **include_privates** (bool) – True to include private property keys in the dump. Private property keys are those that begin with "_".

Returns The entity as a dictionary representation.

Return type dict

get_in_vertex()
Return the in/head vertex.

Returns In vertex.

Return type *IVertex*

get_out_vertex()
Return the out/tail vertex.

Returns Out vertex.

Return type *IVertex*

is_bound()
Return True if the entity is bound to a graph.

Returns True is the entity is bound to a *IGraph*

Return type bool

remove_property(*key*)
Un-assigns a property key with its value.

Parameters **key** (str) – Key that you are removing.

set_property(***kwargs*)
Assign or update a property.

Parameters **kwargs** (key `str` and value.) – Key and value pairs.

Entity Set

class `ruruki.interfaces.`**IEntitySet**
Interface for a entity containers.

add(*entity*)
Add a unique entity to the set.

Parameters **entity** (*IEntity*) – Unique entity being added to the set.

Raises **KeyError** – KeyError is raised if the entity being added to the set has a `ident` conflict with an existing *IEntity*

all(*label=None*, ***kwargs*)
Return all the items in the container as a list.

Parameters

- **label** (`str`) – Filter for entities that have a particular label. If `None`, all entities are returned.
- **kwargs** (*key=value*) – Property key and value.

Returns All the items in the container.

Return type `list` containing *IEntity*

clear()
This is slow (creates N new iterators!) but effective.

discard(*entity*)
Remove a entity from the current set.

Parameters **entity** (*IEntity*) – Entity to be removed from the set.

Raises **KeyError** – KeyError is raised if the entity being discared does not exists in the set.

filter(*label=None*, ***kwargs*)
Filter for all entities that match the given label and properties returning a new *IEntitySet*

Note: Keywords should be made of a property name (as passed to the *add_vertex()* or *add_edge()* methods) followed by one of these suffixes, to control how the given value is matched against the *IEntity*'s values for that property.

- __contains
- __icontains
- __startswith
- __istartswith
- __endswith
- __iendswith
- __le

- __lt
- __ge
- __gt
- __eq
- __ieq
- __ne
- __ine

Parameters

- **label** (str) – Filter for entities that have a particular label. If None, all entities are returned.
- **kwargs** (*key=value*) – Property key and value.

Returns New *IEntitySet* with the entities that matched the filter criteria.

Return type *IEntitySet*

get(*ident*)
Return the *IEntity* that has the identification number supplied by parameter *ident*

Parameters **ident** (int) – Identification number.

Raises **KeyError** – Raised if there are no *IEntity* that has the given identification number supplied by parameter *ident*.

Returns The *IEntity* that has the identification number supplied by parameter *indent*

Return type Iterable of str

get_indexes()
Return all the index labels and properties.

Returns All the index label and property keys.

Return type Iterable of tuple of str, str

get_labels()
Return labels known to the entity set.

Returns All the the labels known to the entity set.

Return type Iterable of str

isdisjoint(*other*)
Return True if two sets have a null intersection.

pop()
Return the popped value. Raise KeyError if empty.

remove(*entity*)
Like *discard()*, remove a entity from the current set.

Parameters **entity** (*IEntity*) – Entity to be removed from the set.

Raises **KeyError** – KeyError is raised if the entity being removed does not exists in the set.

sorted(*key=None*, *reverse=False*)
Sort and return all items in the container.

Parameters

- **key** (*callable*) – Key specifies a function of one argument that is used to extract a comparison key from each list element. The default is to compare the elements directly.
- **reverse** (bool) – If set to True, then the list elements are sorted as if each comparison were reverted.

Returns All the items in the container.

Return type list containing *IEntity*

update_index(*entity*, ***kwargs*)
Update the index with the new property keys.

Parameters

- **entity** (*IEntity*) – Entity with a set of properties that need to be indexed.
- **kwargs** (str, value.) – Property key and values to set on the new created vertex.

Locks

class ruruki.interfaces.**ILock**
Interface for locking.

acquire()
Acquire a lock.

Raises **AcquireError** – If a lock failed to be acquired.

release()
Release the lock.

Raises **ReleaseError** – If the lock was unable to be released.

CHAPTER 4

Implementations

Graphs

class ruruki.graphs.**Graph**
In-memory graph database.

See *IGraph* for doco.

class ruruki.graphs.**PersistentGraph**(*path*, *auto_create=True*)
Persistent Graph database storing data to a file system.

See *IGraph* for doco.

Note: Verices and Edges ID's are retained when the path is loaded.

Warning: Use this persistent graph if performance is not important. There is a performance hit due to the extra disk I/O overhead when doing many writing/updating operations.

```
path
   |_ vertices
   |      |_ constraints.json (file)
   |      |_ label
   |      |      |_ 0
   |      |         |_ properties.json (file)
   |      |         |_ in-edges
   |      |         |      |_ 0 -> ../../../../edges/label/0 (symlink)
   |      |         |_ out-edges
   |      |                |_
   |      |
   |      |_ label
   |      |    |_ 1
```

```
|     |          |_ properties.json (file)
|     |           |_ in-edges
|     |           |     |_
|     |           |_ out-edges
|     |                 |_ 0 -> ../../../../edges/label/0 (symlink)
|
|_ edges
      |_ label
            |
            |_0
              |_ properties.json (file)
              |_ head
              |   |_ 0 -> ../../../vertices/0 (symlink)
              |_ tail
                  |_ 1 -> ../../../vertices/1 (symlink)
```

Parameters

- **path** (str) – Path to ruruki graph data on disk.
- **auto_create** (bool) – If True, then missing vertices or edges directories will be created.

Raises **DatabasePathLocked** – If the path is already locked by another persistence graph instance.

Entities

class ruruki.entities.**EntitySet**(*entities=None*)

EntitySet used for storing, filtering, and iterating over *IEntity* objects.

Note: See *IEntitySet* for documenation.

Parameters **entities** (Iterable of *IEntity*) – Entities being added to the set.

clear()

This is slow (creates N new iterators!) but effective.

discard(*entity*)

Remove a entity from the current set.

Parameters **entity** (*IEntity*) – Entity to be removed from the set.

Raises **KeyError** – KeyError is raised if the entity being discared does not exists in the set.

isdisjoint(*other*)

Return True if two sets have a null intersection.

pop()

Return the popped value. Raise KeyError if empty.

class ruruki.entities.**Entity**(*label=None*, ***kwargs*)

Base class for containing the common methods used for the other entities like vertices and edges.

Note: See *IEntity* for doco.

Note: The properties can be accessed as if they are attributes directly by prepending `prop__` to the key.

```
>>> e = Entity("Entity", name="Example")
>>> e.prop__name
'Example'
```

Parameters

- **label** – *IEntity* label.
- **kwargs** (str`=value or :class:`dict) – Additional properties for the *IEntity*.

class ruruki.entities.**Vertex**(*label=None*, ***kwargs*)

Vertex/Node is the representation of a entity. It can be anything and contains properties for additional information.

Note: See *IVertex* for doco.

Note: The properties can be accessed as if they are attributes directly by prepending `prop__` to the key.

```
>>> v = Vertex("Person", name="Foo")
>>> v.prop__name
'Foo'
```

Parameters

- **label** – *IEntity* label.
- **kwargs** (str`=value or :class:`dict) – Additional properties for the *IEntity*.

class ruruki.entities.**PersistentVertex**(**args*, ***kwargs*)

Persistent Vertex behaves exactly the same as a *Vertex* but has an additional path attribute which is the disk location.

class ruruki.entities.**Edge**(*head*, *label*, *tail*, ***kwargs*)

Edge/Relationship is the representation of a relationship between two entities. A edge has properties for additional information.

Note: See *IEdge* for doco.

Note: The properties can be accessed as if they are attributes directly by prepending `prop__` to the key.

```
>>> v1 = Vertex("Person", name="Foo")
>>> v2 = Vertex("Person", name="Bar")
>>> e = Edge(v1, "knows", v2, since="school")
>>> e.prop__since
'school'
```

Parameters

- **head** (*IVertex*) – Head *IVertex* of the edge.
- **label** – *IEntity* label.
- **tail** (*IVertex*) – Tail *IVertex* of the edge.
- **kwargs** (str`=value or :class:`dict) – Additional properties for the *IEntity*.

class ruruki.entities.**PersistentEdge**(**args*, ***kwargs*)

Persistent Edge behaves exactly the same as a *Edge* but has an additional path attribute which is the disk location.

Locks

class ruruki.locks.**Lock**

Base locking class.

See *ILock* for doco.

locked

Return the status of the lock.

Returns True if the lock is acquired.

Return type bool

class ruruki.locks.**FileLock**(*filename*)

File based locking.

Parameters **filename** (str) – Filename to create a lock on.

locked

Return the status of the lock.

Returns True if the lock is acquired.

Return type bool

class ruruki.locks.**DirectoryLock**(*path*)

Directory based locking.

Parameters **path** (str) – Path that you are locking.

locked

Return the status of the lock.

Returns True if the lock is acquired.

Return type bool

Parsers

Cypher parser

ruruki.parsers.cypher_parser.**parse**(*query_string*)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

L

O

P

R

S

U

V