graphANNIS Documentation

Thomas Krause

Aug 09, 2021

Contents

CHAPTER 1

graphannis.cs module

class graphannis.cs.**CorpusStorageManager**(*db_dir='data/'*, *use_parallel=True*)

Bases: object

apply_update(*corpus_name: str*, *update*)

Apply a sequence of updates (*update* parameter) to this graph for a corpus given by the *corpus_name* parameter.

It is ensured that the update process is atomic and that the changes are persisted to disk if the no exceptions are thrown.

Parameters

- **corpus_name** – The name of the corpus to apply the update on.
- **update** – List with elements of the type *graphannis.graph.GraphUpdate*.

```
>>> from graphannis.cs import CorpusStorageManager
>>> from graphannis.graph import GraphUpdate
>>> with CorpusStorageManager() as cs:
...     with GraphUpdate() as g:
...         g.add_node('n1')
...         cs.apply_update('test', g)
```

count(*corpus_name*, *query: str*, *query_language=<QueryLanguage.AQL: 0>*) → int

Count the number of results for a query.

Parameters

- **corpus_name** – The name of the corpus to execute the query on. This can be a string or a list of strings.
- **query** – The query as string.
- **query_language** – The query language of the query (e.g. AQL).

Returns The count of matches as number.

count_extra(*corpus_name*, *query: str*, *query_language=<QueryLanguage.AQL: 0>*) → graphannis.cs.CountExtra

Count the number of results for a *query* and return both the total number of matches and also the number of documents in the result set.

Parameters

- **corpus_name** – The name of the corpus to execute the query on. This can be a string or a list of strings
- **query** – The query as string.
- **query_language** – The query language of the query (e.g. AQL).

Returns The count of matches and documents.

delete_corpus(*corpus_name: str*)

Delete a corpus from the database

```
>>> from graphannis.cs import CorpusStorageManager
>>> from graphannis.graph import GraphUpdate
>>> with CorpusStorageManager() as cs:
...     # create a corpus named "test"
...     with GraphUpdate() as g:
...         g.add_node('anynode')
...         cs.apply_update('test', g)
...     # delete it
...     cs.delete_corpus('test')
True
```

export_to_fs(*corpus_name*, *path*, *fmt: graphannis.cs.ExportFormat = <ExportFormat.GraphMLZip: 1>*)

Export a corpus to an external location on the file system using the given format.

param corpus_name The name of the corpus to export. This can be a string or a list of strings.

param path The location on the file system where the corpus data should be written to.

param fmt The format in which this corpus data will be stored stored.

```
>>> from graphannis.cs import CorpusStorageManager
>>> from graphannis.graph import GraphUpdate
>>> with CorpusStorageManager() as cs:
...     # import the same corpus under different names
...     c1 = cs.import_from_fs("examples/tutorial.graphml", ImportFormat.
↪GraphML, corpus_name="example1")
...     c2 = cs.import_from_fs("examples/tutorial.graphml", ImportFormat.
↪GraphML, corpus_name="example2")
...
...     # export them to different formats
...     cs.export_to_fs(["example1", "example2"], "all.zip", ExportFormat.
↪GraphMLZip)
...     cs.export_to_fs("example2", "example2.graphml", ExportFormat.GraphML)
```

find(*corpus_name*, *query: str*, *query_language=<QueryLanguage.AQL: 0>*, *offset=0*, *limit=10*, *order: graphannis.cs.ResultOrder = <ResultOrder.Normal: 0>*)

Find all results for a *query* and return the match ID for each result.

Parameters

- **corpus_name** – The name of the corpus to execute the query on. This can be a string or a list of strings.
- **query** – The query as string.
- **query_language** – The query language of the query (e.g. AQL).
- **offset** – Skip the *n* first results, where *n* is the offset.
- **limit** – Return at most *n* matches, where *n* is the limit.
- **order** – Specify the order of the matches.

Returns A list of match IDs, where each match ID consists of the matched node annotation identifiers separated by spaces. You can use the *subgraph()* method to get the subgraph for a single match described by the node annnotation identifiers.

frequency(*corpus_name*, *query*, *definition*, *query_language=<QueryLanguage.AQL: 0>*)

Execute a frequency query.

Parameters

- **corpus_name** – The name of the corpus. This can be a string or a list of strings.
- **query** – Query in the specified query language (per default AQL)
- **definition** – A comma seperated list of single frequency definition items as string. Each frequency definition must consist of two parts: the name of referenced node and the (possible qualified) annotation name or "tok" separated by ":". E.g. a frequency definition like:

```
1:tok,3:pos,4:tiger::pos
```

  #1, the pos annotation for node #3 and the would extract the token value for the node pos annotation in the tiger namespace for node #4.

- **query_language** – Optional query language (AQL per default)

Returns A frequency table which is a list of named tuples. The named tuples have the field **values** which is a list with the actual values for this entry and **count** with the number of occurences for these value combination.

import_from_fs(*path*, *fmt: graphannis.cs.ImportFormat = <ImportFormat.RelANNIS: 0>*, *corpus_name: str = None*, *disk_based: bool = False*, *overwrite_existing: bool = False*)

Import corpus from the file system into the database

```
>>> from graphannis.cs import CorpusStorageManager
>>> from graphannis.graph import GraphUpdate
>>> with CorpusStorageManager() as cs:
...     # import relANNIS corpus with automatic name
...     corpus_name = cs.import_from_fs("relannis/GUM")
...     print(corpus_name)
...     # import a GraphML corpus with a different name
...     corpus_name = cs.import_from_fs("examples/tutorial.graphml",␣
↪ImportFormat.GraphML, "example")
...     print(corpus_name)
GUM
example
```

list()

List all available corpora in the corpus storage.

subcorpus_graph(*corpus_name: str*, *document_ids*) → networkx.classes.multidigraph.MultiDiGraph

Return the copy of a subgraph which includes all nodes that belong to any of the given list of sub-corpus/document identifiers. :param corpus_name: The name of the corpus for which the subgraph should be generated from. :param document_ids: A list of sub-corpus/document identifiers describing the subgraph.

subgraph(*corpus_name: str*, *node_ids*, *ctx_left=0*, *ctx_right=0*, *segmentation=None*) → networkx.classes.multidigraph.MultiDiGraph

Return the copy of a subgraph which includes the given list of node annotation identifiers, the nodes that cover the same token as the given nodes and all nodes that cover the token which are part of the defined context.

Parameters

- **corpus_name** – The name of the corpus for which the subgraph should be generated from.
- **node_ids** – A list of node annotation identifiers describing the subgraph.
- **ctx_left** – Left context in token distance to be included in the subgraph.
- **ctx_right** – Right context in token distance to be included in the subgraph.
- **segmentation** – The name of the segmentation which should be used to as base for the context. * Use *None* to define the context in the default token layer.

class graphannis.cs.**CountExtra**(*match_count*, *document_count*)

Bases: tuple

document_count

Alias for field number 1

match_count

Alias for field number 0

class graphannis.cs.**ExportFormat**

Bases: enum.IntEnum

An enum of all supported output formats of graphANNIS.

GraphML = 0

GraphML (http://graphml.graphdrawing.org/) based export-format, suitable to be imported into other graph databases. This format follows the extensions/conventions of the Neo4j GraphML module (https://neo4j.com/docs/labs/apoc/current/import/graphml/).

GraphMLDirectory = 2

Like **GraphML**, but using a directory with multiple GraphML files, each for one corpus.

GraphMLZip = 1

Like **GraphML**, but compressed as ZIP file. Linked files are also copied into the ZIP file.

class graphannis.cs.**FrequencyTableEntry**(*values*, *count*)

Bases: tuple

count

Alias for field number 1

values

Alias for field number 0

class graphannis.cs.**ImportFormat**

Bases: enum.IntEnum

An enum of all supported input formats of graphANNIS.

GraphML = 1
GraphML (http://graphml.graphdrawing.org/) based export-format, suitable to be imported from other graph databases. This format follows the extensions/conventions of the Neo4j GraphML module (https://neo4j.com/docs/labs/apoc/current/import/graphml/).

RelANNIS = 0
Legacy relANNIS import file format: http://korpling.github.io/ANNIS/4.0/developer-guide/annisimportformat.html

class graphannis.cs.**QueryLanguage**
Bases: enum.IntEnum

Defines which query language is used

AQL = 0
Default ANNIS Query Language (AQL)

AQLQuirksV3 = 1
AQL in quirks mode that emulates some of the behavior of ANNIS3

class graphannis.cs.**ResultOrder**
Bases: enum.IntEnum

Defines the ordering of results

Inverted = 1

Normal = 0

NotSorted = 3

Randomized = 2

CHAPTER 2

graphannis.graph module

class graphannis.graph.**GraphUpdate**

Bases: object

add_edge(*source_node*, *target_node*, *layer*, *component_type*, *component_name*)

Add an edge between two existing nodes.

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...     g.add_node('n2')
...     g.add_edge('n1', 'n2', 'mylayer', 'Pointing', 'dep')
...
```

add_edge_label(*source_node*, *target_node*, *layer*, *component_type*, *component_name*, *anno_ns*, *anno_name*, *anno_value*)

Add a label to an existing edge

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...     g.add_node('n2')
...     g.add_edge('n1', 'n2', 'mylayer', 'Pointing', 'dep')
...     g.add_edge_label('n1', 'n2', 'mylayer', 'Pointing', 'dep',
...     'myns', 'myanno', 'annoval')
...
```

add_node(*node_name*, *node_type='node'*)

Add a named node to the graph

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...
```

add_node_label(*node_name*, *anno_ns*, *anno_name*, *anno_value*)
Add a label to an existing node to the graph

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...     g.add_node_label('n1', 'mynamespace', 'myname', 'myvalue')
...
```

delete_edge(*source_node*, *target_node*, *layer*, *component_type*, *component_name*)
Delete an existingedge between two nodes.

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...     g.add_node('n2')
...     g.add_edge('n1', 'n2', 'mylayer', 'Pointing', 'dep')
...     g.delete_edge('n1', 'n2', 'mylayer', 'Pointing', 'dep')
...
```

delete_edge_label(*source_node*, *target_node*, *layer*, *component_type*, *component_name*, *anno_ns*, *anno_name*)
Delete a label from an edge

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...     g.add_node('n2')
...     g.add_edge('n1', 'n2', 'mylayer', 'Pointing', 'dep')
...     g.add_edge_label('n1', 'n2', 'mylayer', 'Pointing', 'dep',
...     'myns', 'myanno', 'annoval')
...     g.delete_edge_label('n1', 'n2', 'mylayer', 'Pointing', 'dep',
...     'myns', 'myanno')
...
```

delete_node_label(*node_name*, *anno_ns*, *anno_name*)
Delete an existing label from an existing node

```
>>> from graphannis.graph import GraphUpdate
>>> with GraphUpdate() as g:
...     g.add_node('n1')
...     g.add_node_label('n1', 'mynamespace', 'myname', 'myvalue')
...     g.delete_node_label('n1', 'mynamespace', 'myname')
...
```

CHAPTER 3

graphannis.util module

graphannis.util.**node_name_from_match**(*match*)

Takes a match identifier (which includes the matched annotation name) and returns the node name. This can take a single string or a list of strings as argument.

```
>>> m = node_name_from_match("tiger::cat::topcorpus/subcorpus/doc1#n2")
>>> m == "topcorpus/subcorpus/doc1#n2"
True
```

CHAPTER 4

graphannis.errors module

exception graphannis.errors.**AQLSemanticError**(*msg: str*, *cause: graphannis.errors.GraphANNISException = None*)

Bases: *graphannis.errors.GraphANNISException*

exception graphannis.errors.**AQLSyntaxError**(*msg: str*, *cause: graphannis.errors.GraphANNISException = None*)

Bases: *graphannis.errors.GraphANNISException*

exception graphannis.errors.**GraphANNISException**(*msg: str*, *cause: Exception = None*)

Bases: Exception

exception graphannis.errors.**NoSuchCorpus**(*msg: str*, *cause: graphannis.errors.GraphANNISException = None*)

Bases: *graphannis.errors.GraphANNISException*

exception graphannis.errors.**SetLoggerError**(*msg: str*, *cause: graphannis.errors.GraphANNISException = None*)

Bases: *graphannis.errors.GraphANNISException*

graphannis.errors.**consume_errors**(*err*)

Processes the error list from the C-API and raises an exception if they contain an error. It also deletes the memory if the vector has been filled.

CHAPTER 5

graphannis.common module

exception graphannis.common.**ANNISException**(*m: str*)

Bases: Exception

Python Module Index

g

Index

L

M

N

Q

R

S

V