blitzdb Documentation

Release 0.1

Andreas Dewes

December 05, 2016

Contents

BlitzDB, or just **Blitz** , [1] is a document-based, object-oriented, transactional database written purely in Python. Among other things, it provides a **powerful querying language**, **deep indexing of documents**, **compressed data storage** and **automatic referencing of embedded documents**. It is reasonably fast, can be easily embedded in any Python application and does not have any external dependencies (except when using a third-party backend). In addition, you can use it as a **frontend** to other database engines such as MongoDB in case you should need more power.

[1] **Blitz** is the German word for *ligthning*. "**blitzschnell**" means "*really fast*".

CHAPTER 1

Key Features

- Document-based, object-oriented interface.
- Powerful and rich querying language.
- Deep document indexes on arbitrary fields.
- Compressed storage of documents.
- Support for multiple backends (e.g. file-based storage, MongoDB).
- Support for database transactions (currently only for the file-based backend).

Warning: Please be aware that this is an early development version of BlitzDB, so there are probably still some bugs in the code and the documentation is not very extensive for the moment.
We are currently looking for contributors and people that are passionate about building BlitzDB with us, so if you're interested in joining our quest or if you have any suggestions, remarks or complaints, feel free to get in touch, e.g. via email or Github. Thanks!

CHAPTER 2

Installation

For detailed installation instructions, have a look at the documentation. The easiest way to install checkmate is via **pip** or **easy_install**:

```
pip install blitzdb
#or
easy_install blitzdb
```

Alternatively, you can just download the source from Github and install it manually by running (in the project directory):

```
git clone git@github.com:adewes/blitzdb.git
cd blitzdb
sudo python setup.py install
```

CHAPTER 3

Getting Started

To get started, have a look at the basic tutorial or check out the API documentation.

CHAPTER 4

Use Cases

Blitz has been designed as an embeddable, easy-to-use and *reasonably* fast database. In general, it performs well even with moderately large (>100.000 entries) collections of documents, provided you make good use of its indexing capabilities.

It is **NOT** a fully-fledged database system like MySQL or MongoDB:

- In the current version it does not provide any support for concurrent writes/reads to the database.
- It uses a relatively simple indexing mechanism based on hash tables and stores documents in flat files on the disk, hence querying performance will usually not be comparable to state-of-the art database systems.

However, for more demanding use cases, Blitz can be used as a frontend to a third-party backends, most notably *MongoDB*.

CHAPTER 5

Motivation

Sometimes you just want to store some structured data (think *dictionaries*) in a local database and get it out again when needed. For this, you could use e.g. the *shelve* module, or an embedded database like *sqlite*, or resort to an external database like *MongoDB*. The problem with these is that they either don't support **rich querying** of the data (*shelve*), require you to specify the data format beforehand (*sqlite*) or require additional software running on your machine (*MongoDB*).

Blitz provides a new approach to this problem by giving you an embeddable database with a **flexible data format** and **rich querying** capabilities.

5.1 Tutorials

Currently there is only a basic tutorial available, for further details have a look at the API documentation or the documentation of the backends. If you encounter any problems, feel free to get in touch, e.g. via email or Github. Thanks!

5.1.1 Basics

Welcome! This tutorial will help you to get up & running with Blitz. For a more comprehensive overview of Blitz, please consult the API documentation or the documentation of specific backends. Let's get started!

Working with Documents

Just like in Python, in Blitz all documents are objects. To create a new type of document, you just define a class that derives from *`blitzdb.document.Document`*:

```
from blitzdb import Document

class Actor(Document):
    pass

class Movie(Document):
    pass
```

That's it! We can now create and work with instances of *Actor* and *Movie* documents:

```
charlie_chaplin = Actor({
                        'first_name' : 'Charlie',
                        'last_name' : 'Chaplin',
                        'is_funny' : True,
                        'birth_year' : 1889,
                        'filmography' : [
                            ('The Kid',1921),
                            ('A Woman of Paris',1923),
                            #...
                            ('Modern Times', 1936)
                        ]
                      })
```

We can access the document attributes of the given instances as class attributes:

```
print "%s %s was born in %d" % (charlie_chaplin.first_name,
                                 charlie_chaplin.last_name,
                                 charlie_chaplin.birth_year)
```

Alternatively, we can use the *attributes* attribute to access them:

```
print "%(first_name)s %(last_name)s was born in %(birth_year)d" % charlie_chaplin.attributes
```

This is also pretty useful if you define attributes that have names which get *shadowed* by methods of the *Document* class (e.g. *save* or *filter*).

Connecting to a database

To store documents in a database, you need to create a backend first. Blitz supports multiple backends (currently a file-based one and one that wraps MongoDB). In this tutorial we will use a file-based backend, which you create like this:

```
from blitzdb import FileBackend

backend = FileBackend("./my-db")
```

This connects Blitz to a file-based database within the "./my-db" directory, or creates a new database there if none should be present. The backend provides various functions such as `save`, `get`, `filter` and `delete`, which can be used to store, retrieve, update and delete objects. Let's have a look at these operations.

Note: You can choose between different formats to store your documents when using the file-based backend, using e.g. the *json*, *pickle* or *marshal* Python libraries. Choose the document format when creating the backend by passing a configuration dictionary, e.g. `backend = FileBackend("./my-db", {'serializer_class': 'pickle'})` (`'serializer_class'` can also be `'json'` or `'marshal'`). By default, all documents will be stored as gzipped JSON files.

Warning: The default serializer class is `'json'`, but this does not allow a perfect roundtrip from python to JSON and back. Python supports many more datatypes than JSON, see the python JSON documentation.

Inserting Documents

We can store the *Author* object that we created before in our new database like this:

```
backend.save(charlie_chaplin)
```

Alternatively, we can also directly call the *save* function of the *Actor* instance with the backend as an argument:

```
charlie_chaplin.save(backend)
```

In addition, since Blitz is a **transactional database**, we have to call the `commit` function of the backend to write the new document to disk:

```
#Will commit changes to disk
backend.commit()
```

Note: Use the `Backend.begin` function to start a new database transaction and the `Backend.rollback` function to roll back the state of the database to the beginning of a transaction, if needed. By default, Blitz uses a **local isolation level** for transactions, so changes you make to the state of the database will be visible to parts of your program using the same backend, but will only be written to disk when `Backend.commit` is invoked.

Retrieving Documents

Retrieving objects from the database is just as easy. If we want to get a single object, we can use the `get()` method, specifying the Document class and any combination of attributes that uniquely identifies the document:

```
actor = backend.get(Actor,{'first_name' : 'Charlie','last_name' : 'Chaplin'})
```

Alternatively, if we know the *primary key* of the object, we can just specify this:

```
the_kid = Movie({'title' : 'The Kid'})
actor = backend.get(Actor,{'pk' : charlie_chaplin.pk})
```

Note: Pro-Tip

If Blitz can't find a document matching your query, it will raise a `Document.DoesNotExist` exception. Likewise, if it finds more than one document matching your query it will raise `Document.MultipleDocumentsReturned`. These exceptions are specific to the document class to which they belong and can be accessed as attributes of it, e.g. like this:

```
try:
    actor = backend.get(Actor,{'first_name' : 'Charlie'})
except Actor.DoesNotExist:
    #no 'Charlie' in the database
    pass
except Actor.MultipleDocumentsReturned:
    #more than one 'Charlie' in the database
    pass
```

If we want to retrieve all objects matching a given query, we can use the `filter()` method instead:

```
#Retrieve all actors that were born in 1889
actors = backend.filter(Actor,{'birth_year' : 1889})
```

This will return a `QuerySet`, which contains a list of keys of all objects that match our query. Query sets are iterables, so we can use them just like lists:

```
print "Found %d actors" % len(actors)
for actor in actors:
    print actor.first_name+" "+actor.last_name
```

Deleting Documents

We can delete documents from the database by calling the `delete()` method of the backend with an instance of the object that we wish to delete:

```
backend.delete(charlie_chaplin)
```

This will remove the document from the given collection and set its primary key to *None*. We can delete a whole query set in the same way by calling its `delete()` method:

```
#Retrieve all actors from the database
actors = backend.filter(Actor,{})
actors.delete()
```

Defining Relationships

Databases are pretty useless if there's no way to define **relationships** between objects. Like MongoDB, Blitz supports defining references to other documents inside of documents. An example:

```
modern_times = Movie({
                      'title' : 'Modern Times',
                      'year' : 1936,
                      'budget' : 1500000,
                      'run_time_minutes' : 87,
                     })

charlie_chaplin.movies = [modern_times]
modern_times.actors = [charlie_chaplin]

#this will automatically save the movie object as well
backend.save(charlie_chaplin)
```

Internally, BlitzDB converts any *Document* instance that it encounters inside a document to a database reference that contains the primary key of the embedded document and the the name of the collection in which it is stored. Like this, if we reload the actor from the database, the embedded movie objects will get automatically (lazy-)loaded as well:

```
actor = backend.filter(Actor,{'first_name' : 'Charlie','last_name' : 'Chaplin'})

#check that the movies in the retrieved Actor document are instances of Movie
assert isinstance(actor.movies[0],Movie)

#will print 'Modern Times'
print actor.movies[0].title
```

Note: When an object gets loaded from the database, references to other objects that it contains will get loaded **lazily**, i.e. they will get initialized with only their primary key and the name of the collection they can be found in. Their attributes will get automatically loaded if (and only if) you should request them.

Like this, Blitz avoids performing multiple reads from the database unless they are really needed. As a bonus, lazy loading also solves the problem of cyclic document references (like in the example above).

Advanced Querying

Like MongoDB, Blitz supports advanced query operators, which you can include in your query be prefixing them with a *$*. Currently, the following operator expressions are supported:

- **$and** : Performs a boolean **AND** on two or more expressions
- **$or** : Performs a boolean **OR** on two or more expressions
- **$gt** : Performs a **>** comparision between an attribute and a specified value
- **$gte** : Performs a **>=** comparision between an attribute and a specified value
- **$lt** : Performs a **<** comparision between an attribute and a specified value
- **$lte** : Performs a **<=** comparision between an attribute and a specified value
- **$all** : Returns documents containing all values in the argument list.
- **$in** : Returns documents matching at least one of the values in the argument list.
- **$ne** : Performs a **not equal** operation on the given expression
- **$not** : Checks for non-equality between an attribute and the given value.

The syntax and semantics of these operators is identical to MongoDB, so for further information have a look at their documentation.

Example: Boolean AND

By default, if you specify more than one attribute in a query, an implicit *$and* query will be performed, returning only the documents that match **all** attribute/value pairs given in your query. You can also specify this behavior explicitly by using then *$and* operator, so the following two queries are identical:

```
backend.filter(Actor,{'first_name' : 'Charlie','last_name' : 'Chaplin'})
#is equivalent to...
backend.filter(Actor,{'$and' : [{'first_name' : 'Charlie'},{'last_name' : 'Chaplin'}]})
```

Using *$and* can be necessary if you want to reference the same document attribute more than once in your query, e.g. like this:

```
#Get all actors born beteen 1900 and 1940
backend.filter(Actor,{'$and' : [{'birth_year' : {'$gte' : 1900}},{'birth_year' : {'$lte' : 1940}}]})
```

Where to Go from Here

Currently there are no other tutorials available (this will change soon), so if you have further questions, feel free to send us an e-mail or post an issue on Github. The test suite also contains a large number of examples on how to use the API to work with documents.

5.2 Installation

5.2.1 Using pip or easy_install

The easiest way to install Blitz is via *pip* or *easy_install*:

```
pip install blitzdb
#or...
easy_install install blitzdb
```

This will fetch the latest version from PyPi and install it on your machine.

5.2.2 Using Github

Since BlitzDB is still in heavy development, installing directly from the Github source will guarantee that your version will contain the latest features and bugfixes. To clone and install the project, just do the following

```
git clone git@github.com:adewes/blitzdb.git
cd blitzdb
sudo python setup.py install
```

5.2.3 Requirements

The *vanilla* version of Blitz does not require any non-standard Python modules to run. However, you might want to install the following Python libraries to be able to use all features of Blitz:

- pymongo: Required for the MongoDB backend
- cjson: Required for the CJsonEncoder (improved JSON serialization speed)
- pytest: Required for running the test suite
- fake-factory: Required for generating fake test data

You can install these requirements using pip and the *requirements.txt* file:

```
#in BlitzDB main folder
pip install -R requirements.txt
```

5.3 API

The architecture of BlitzDB is defined by the following three classes:

- Document : The *Document* class is the base class for all documents stored in a database.
- Backend : The *Backend* class is responsible for storing and retrieving documents from a database.
- QuerySet : The *QuerySet* class manages a list of documents as returned e.g. by the `filter` function.

In addition, specific backends might internally define further classes, with which you will normally not interact as an end user, though.

5.3.1 The Backend

The backend provides the main interface to the database and is responsible for retrieving, storing and deleting objects. The class documented here is an abstract base class that gets implemented by the specific backends. Functionality can vary between different backends, e.g. **database transactions** will not be supported by all backends.

class `blitzdb.backends.base.`**`Backend`**(*autodiscover_classes=True*, *autoload_embedded=True*, *allow_documents_in_query=True*)

Abstract base class for all backend implementations. Provides operations for querying the database, as well as for storing, updating and deleting documenta.

Parameters **`autodiscover_classes`** – If set to *True*, document classes will be discovered automatically, using a global list of all classes generated by the Document metaclass.

The ‘Meta‘ attribute

As with *blitzdb.document.Document*, the *Meta* attribute can be used to define certain class-wide settings and properties. Redefine it in your backend implementation to change the default values.

`autodiscover_classes`()

Registers all document classes that have been defined in the code so far. The discovery mechanism works by reading the value of *blitzdb.document.document_classes*, which is updated by the meta-class of the `blitzdb.document.Document` class upon creation of a new subclass.

`autoregister`(*cls*)

Autoregister a class that is encountered for the first time.

Parameters **`cls`** – The class that should be registered.

`delete`(*obj*)

Deletes an object from the database.

Parameters **`obj`** – The object to be deleted.

`deserialize`(*obj*, *decoders=None*)

Deserializes a given object, i.e. converts references to other (known) *Document* objects by lazy instances of the corresponding class. This allows the automatic fetching of related documents from the database as required.

Parameters **`obj`** – The object to be deserialized.

Returns The deserialized object.

`filter`(*cls*, ***kwargs*)

Filter objects from the database that correspond to a given set of properties.

Parameters

- **`cls`** – The class for which to filter objects from the database.
- **`properties`** – The properties used to filter objects.
- **`sorty_by`** – A field or list of fields according to which to sort the returned objects.
- **`limit`** – The maximal number of objects to return in a single query.
- **`offset`** – The offset in respect to the beginning of the result list (to be used in conjunction with *limit*).

Returns A *blitzdb.queryset.QuerySet* instance containing the keys of the objects matching the query.

Functionality might differ between backends

Please be aware that the functionality of the *filter* function might differ from backend to backend. Consult the documentation of the given backend that you use to find out which queries are supported.

`get`(*cls*, *properties*)

Abstract method to retrieve a single object from the database according to a list of properties.

Parameters

- **`cls`** – The class for which to return an object.
- **`properties`** – The properties of the object to be returned

Returns An instance of the requested object.

Exception Behavior

Raises a `blitzdb.document.Document.DoesNotExist` exception if no object with the given properties exists in the database, and a `blitzdb.document.Document.MultipleObjectsReturned` exception if more than one object in the database corresponds to the given properties.

register(*cls*, *parameters=None*)

Explicitly register a new document class for use in the backend.

Parameters

- **cls** – A reference to the class to be defined
- **parameters** – A dictionary of parameters. Currently, only the *collection* parameter is used to specify the collection in which to store the documents of the given class.

Registering classes

If possible, always use *autodiscover_classes = True* or register your document classes beforehand using the *register* function, since this ensures that related documents can be initialized appropriately. For example, suppose you have a document class *Author* that contains a list of references to documents of class *Book*. If you retrieve an instance of an *Author* object from the database without having registered the *Book* class, the references to that class will not get parsed properly and will just show up as dictionaries containing a primary key and a collection name.

Also, when *blitzdb.backends.base.Backend.autoregister()* is used to register a class, you can't pass in any parameters to customize e.g. the collection name for that class (you can of course do this throught the *Meta* attribute of the class)

save(*obj*, *cache=None*)

Abstract method to save a *Document* instance to the database.

Parameters

- **obj** – The object to be stored in the database.
- **cache** – Whether to performed a cached save operation (not supported by all backends).

serialize(*obj*, *convert_keys_to_str=False*, *embed_level=0*, *encoders=None*, *autosave=True*, *for_query=False*)

Serializes a given object, i.e. converts it to a representation that can be stored in the database. This usually involves replacing all *Document* instances by database references to them.

Parameters

- **obj** – The object to serialize.
- **convert_keys_to_str** – If *True*, converts all dictionary keys to string (this is e.g. required for the MongoDB backend)
- **embed_level** – If *embed_level > 0*, instances of *Document* classes will be embedded instead of referenced. The value of the parameter will get decremented by 1 when calling *serialize* on child objects.
- **autosave** – Whether to automatically save embedded objects without a primary key to the database.
- **for_query** – If true, only the *pk* and *__collection__* attributes will be included in document references.

Returns The serialized object.

5.3.2 The Document

The *Document* class is the base for all documents stored in the database. To create a new document type, just create a class that inherits from this base class:

```
from blitzdb import Document

class Author(Document):
    pass
```

From document classes to collections

Internally, Blitz stores document attributes in *collections*, which it distinguishes based by name. By default, the name of a collection will be the lowercased name of the corresponding class (e.g. *author* for the *Author* class above). You can override this behavior by setting the *collection* attribute in the document class' *Meta* class attribute:

```
class Author(Document):

    class Meta(Document.Meta):
        collection = 'fancy_authors'
```

Likewise, you can also change the name of the attribute to be used as primary key for this document class, which defaults to *pk*:

```
class Author(Document):

    class Meta(Document.Meta):
        pk = 'name' #use the name of the author as the primary key
```

class `blitzdb.document.`**`Document`**(*attributes=None*, *lazy=False*, *default_backend=None*, *autoload=True*)

`__eq__`(*other*)
Compares the document instance to another object. The comparison rules are as follows:

- If the Python *id* of the objects are identical, return *True*
- If the types of the objects differ, return *False*
- If the types match and the primary keys are identical, return *True*
- If the types and attributes of the objects match, return *True*
- Otherwise, return *False*

`attributes`
Returns a reference to the attributes of the document. The attributes are the *"unique source of truth"* about the state of a document.

`autogenerate_pk`()
Autogenerates a primary key for this document. This function gets called by the backend if you save a document without a primary key field. By default, it uses *uuid.uuid1().hex* to generate a (statistically) unique primary key for the object (more about UUIDs). If you want to define your own primary key generation mechanism, just redefine this function in your document class.

`delete`(*backend=None*)
Deletes a document from the database. If the *backend* argument is not specified, the function resorts to the *default backend* as defined during object instantiation. If no such backend is defined, an *AttributeError* exception will be thrown.

Parameters **`backend`** – the backend from which to delete the document.

`initialize`()

Gets called when **after** the object attributes get loaded from the database. Redefine it in your document class to perform object initialization tasks.

Keep in Mind

The function also get called after invoking the *revert* function, which resets the object attributes to those in the database, so do not assume that the function will get called only once during the lifetime of the object.

Likewise, you should **not** perform any initialization in the *__init__* function to initialize your object, since this can possibly break lazy loading and *revert* operations.

`pk`

Returns (or sets) the primary key of the document, which is stored in the *attributes* dict along with all other attributes. The name of the primary key defaults to *pk* and can be redefine in the *Meta* class. This function provides a standardized way to retrieve and set the primary key of a document and is used by the backend and a few other classes. If possible, always use this function to access the primary key of a document.

Automatic primary key generation

If you save a document to the database that has an empty primary key field, Blitz will create a default primary-key by calling the *autogenerate_pk* function of the document. To generate your own primary keys, just redefine this function in your derived document class.

`revert`(*backend=None*)

Reverts the state of the document to that contained in the database. If the *backend* argument is not specified, the function resorts to the *default backend* as defined during object instantiation. If no such backend is defined, an *AttributeError* exception will be thrown.

Parameters **`backend`** – the backend from which to delete the document.

Keep in Mind

This function will call the *initialize* function after loading the object, which allows you to perform document-specific initialization tasks if needed.

`save`(*backend=None*)

Saves a document to the database. If the *backend* argument is not specified, the function resorts to the *default backend* as defined during object instantiation. If no such backend is defined, an *AttributeError* exception will be thrown.

Parameters **`backend`** – the backend in which to store the document.

5.3.3 Query Sets

Query sets are used to work with sets of objects that have been retrieved from the database. For example, whenever you call the *filter* function of the backend, you will receive a *QuerySet* object in return. This object stores references to all documents matching your query and can retrieve these objects for you if necessary.

This class is an abstract base class that gets implemented by the specific backends.

class `blitzdb.queryset.`**`QuerySet`**(*backend*, *cls*)

Stores information about objects returned by a database query and retrieves instances of these objects if necessary.

Parameters

- **`backend`** – The backend to use for :py:meth:filter‘filtering‘ etc.
- **`cls`** – The class of the documents stored in the query set.

`__eq__`(*other*)

Checks if two query sets are equal. Implement this in your derived query set class.

Parameters **`other`** – The object this query set is compared to.

`__getitem__`(*i*)

Returns a specific element from a query set.

`__len__`()

Return the number of documents contained in this query set.

`__ne__`(*other*)

Checks if two query sets are unequal.

Parameters **`other`** – The object this query set is compared to.

`delete`()

Deletes all objects contained in this query set from the database.

`filter`(**args*, ***kwargs*)

Performs a *filter* operation on all documents contained in the query set. See `blitzdb.backends.base.Backend.filter()` for more details.

5.3.4 Exceptions

When things go awry, Blitz will throw a number of specific exceptions to indicate the type of error that has happened. For most database operations, the exception model is modeled after the one used in the **Django** web development framework.

Querying Errors

class `blitzdb.document.Document.`**`DoesNotExist`**

Gets raised when you try to retrieve an object from the database (typically using the *get* function of the backend) that does not exist.

Keep in Mind

Like the *MultipleDocumentsReturned* exception, this exception is specific to the document class for which it is raised.

example:

```
class Author(Document):
    pass

class Book(Document):
    pass
    try:
        raise Book.DoesNotExist
    except Author.DoesNotExist:
        #this will NOT catch the Book.DoesNotExist exception!
        print "got Author.DoesNotExist exception!"
    except Book.DoesNotExist:
        print "got Book.DoesNotExist exception!"
```

class `blitzdb.document.Document.`**`MultipleDocumentsReturned`**

Gets raised when a query that should return only a single object (e.g. the *get* function of the backend) finds more than one matching document in the database. Like the *DoesNotExist* exception, it is specific to the document class for which it is raised.

Transaction Errors

Transaction errors get raised if functions that are supposed to run only inside a transaction get called outside of a transaction, or vice-versa. Please note that not all backends currently support database transactions.

class `blitzdb.backends.base.`**`NotInTransaction`**

Gets raised if a function that must only be used inside a database transaction gets called outside a transaction.

class `blitzdb.backends.base.`**`InTransaction`**

Gets raised if a function that must only be used outside a database transaction gets called inside a transaction.

5.4 Backends

Under the hood, BlitzDB is not just a database engine but more a **database wrapper** like **SQLAlchemy**. Since it provides its own file-based backend, it can be used as a standalone solution though. In some cases it might be useful to use it with a third-party backend such as MongoDB though, e.g. if you need more "*horse power*" or want the additional query efficiency that real databases usually offer.

Currently, Blitz comes with two preinstalled backends:

5.4.1 The Native (file-based) Backend

Stores documents and indexes in flat files on your disk. Can be used without any external software, so it's a great fit for projects that need a document-oriented database but do not want (or are unable) to use a third-party solution for this.

Note: This backend is **transactional**, which means that changes on the database will be written to disk only when you call the *`Backend.commit()`* function explicitly (there is an *autocommit* option, though).

The performance of this backend is reasonable for moderately sized datasets (< 100.000 entries).Future version of the backend might support in-memory caching of objects to speed up the performance even more.

class `blitzdb.backends.file.`**`Backend`**(*path*, *config=None*, *overwrite_config=False*, ***kwargs*)

Bases: *`blitzdb.backends.base.Backend`*

A file-based database backend. Uses flat files to store objects on the hard disk and file-based indexes to optimize querying.

Parameters

- **`path`** – The path to the database. If non-existant it will be created

- **`config`** – The configuration dictionary. If not specified, Blitz will try to load it from disk. If this fails, the default configuration will be used instead.

Warning: It might seem tempting to use the *autocommit* config and not having to worry about calling *commit* by hand. Please be advised that this can incur a significant overhead in write time since a *commit* will trigger a complete rewrite of all indexes to disk.

`begin``()`
Starts a new transaction

`commit``()`
Commits all pending transactions to the database.

Warning

This operation can be **expensive** in runtime if a large number of documents (>100.000) is contained in the database, since it will cause all database indexes to be written to disk.

`create_index``(`*cls_or_collection*, *params=None*, *fields=None*, *ephemeral=False*`)`
Creates a new index on the given collection or class with the given parameters.

Parameters

- **`cls_or_collection`** – The name of the collection or the class for which to create an index
- **`params`** – The parameters of the index
- **`ephemeral`** – Whether to create a persistent or an ephemeral index

params expects either a dictionary of parameters or a string value. In the latter case, it will interpret the string as the name of the key for which an index is to be created.

If *ephemeral* = *True*, the index will be created only in memory and will not be written to disk when `commit()` is called. This is useful for optimizing query performance.

..notice:

```
By default, BlitzDB will create ephemeral indexes for all keys over which you perform querie
so after you've run a query on a given key for the first time, the second run will usually b
much faster.
```

Specifying keys

Keys can be specified just like in MongoDB, using a dot (‘.’) to specify nested keys.

```
actor = Actor({'name' : 'Charlie Chaplin',
 'foo' : {'value' : 'bar'}})
```

If you want to create an index on *actor['foo']['value']* , you can just say

```
backend.create_index(Actor,'foo.value')
```

Warning: Transcendental indexes (i.e. indexes transcending the boundaries of referenced objects) are currently not supported by Blitz, which means you can’t create an index on an attribute value of a document that is embedded in another document.

`rebuild_index``(`*collection*, *key*`)`
Rebuild a given index using the objects stored in the database.

Parameters

- **`collection`** – The name of the collection for which to rebuild the index
- **`key`** – The key of the index to be rebuilt

`rollback``()`
Rolls back a transaction

5.4.2 The MongoDB Backend

This backend provides a thin wrapper around MongoDB. It uses pymongo for the communication with MongoDB. Use this backend if you need high performance or expect to have a large number of documents (> 100.000) in your database.

Warning: Currently this backend does not support **database transactions**. Calls to *commit* and *begin* will be silently discarded (to maintain compatibility to transactional codes), whereas a call to *rollback* will raise a *`NotInTransaction`* exception.

- Native Backend The **native backend**, which we sometimes refer to as the **file-based backend** uses a file-based index and flat files to store objects in a local directory. It has not any external dependencies and is usually sufficent for most low- to mid-end applications.
- MongoDB Backend The **MongoDB backend** uses PyMongo to store and retrieve documents from a MongoDB database. It can be used in high-end applications, where use of a professional database engine is advocated.