datastore Documentation

Release 0.3.0

Juan Batiz-Benet

June 27, 2013

CONTENTS

CHAPTER

ONE

OVERVIEW

datastore is a generic layer of abstraction for data store and database access. It is a **simple** API with the aim to enable application development in a datastore-agnostic way, allowing datastores to be swapped seamlessly without changing application code. Thus, one can leverage different datastores with different strengths without committing the application to one datastore throughout its lifetime.

In addition, grouped datastores significantly simplify complex data access patterns, such as caching and sharding.

CHAPTER

TWO

DOCUMENTATION

The *Core datastore API* contains documentation of the core library.

2.1 Core datastore API

2.1.1 Keys

All objects stored through datastore are described by a `key`. This key uniquely identifies a particular object, and provides namespacing for queries. The `datastore.Key` class below provides the required functionality. One can define another Key class with a different format that conforms to the same interface (particularly stringifying, hashes, namespacing, and ancestry).

Key

Namespace

2.1.2 Queries

In addition to the key-value store `get` and `set` semantics, datastore provides an interface to retrieve multiple records at a time through the use of queries. The datastore Query model gleans a common set of operations performed when querying. To avoid pasting here years of database research, let's summarize the operations datastore supports. Query Operations:

- namespace - scope the query, usually by object type
- filters - select a subset of values by applying constraints
- orders - sort the results by applying sort conditions
- limit - impose a numeric limit on the number of results
- offset - skip a number of results (for efficient pagination)

datastore combines these operations into a simple Query class that allows applications to define their constraints in a simple, generic, and pythonic way without introducing datastore specific calls, languages, etc.

Of course, different datastores provide relational query support across a wide spectrum, from full support in traditional databases to none at all in key-value stores. Datastore aims to provide a common, simple interface for the sake of application evolution over time and keeping large code bases free of tool-specific code. It would be ridiculous to claim to support high-performance queries on architectures that obviously do not. Instead, datastore provides the interface, ideally translating queries to their native form (e.g. into SQL for MySQL or a MongoDB query).

However, on the wrong datastore, queries can potentially incur the high cost of performing the aforementioned *query operations* on the data set directly in python. It is the client's responsibility to select the right tool for the job: pick a data storage solution that fits the application's needs now, and wrap it with a datastore implementation. Some applications, particularly in early development stages, can afford to incurr the cost of queries on non-relational databases (e.g. using a `FileSystemDatastore` and not worry about a database at all). When it comes time to switch the tool for performance, updating the application code can be as simple as swapping the datastore in one place, not all over the application code base. This gain in engineering time, both at initial development and during later iterations, can significantly offest the cost of the layer of abstraction.

tl;dr: queries are supported across datastores. They are very cheap on top of relational databases, and very expensive otherwise. Pick the right tool for the job!

Query classes

Query

Filter

Order

Cursor

Generators

Note on generators: naive datastore queries use generators to delay performing work (such as filtering). Thus, no up-front cost is paid, but rather the cost comes at iteration. This is particularly useful in that even when working on large datasets, the naive query implementation can work as generators do not require having loading the entire dataset in memory upfront. When I say they can work, I do not imply quickly, just that they can work at all.

The **crucial exception**, of course, is orders. If any order is placed on a query, the naive query implementation loses the benefit of delaying the work. That is because one cannot properly sort an entire dataset using a generator (sure, a generator could still be used to avoid paying the cost of sorting the *entire* dataset upfront, but that could still require putting the entire dataset in memory in the worst case).

Specific datastore implementations should keep this in mind, performing as much work as possible in low-level, storage engine specific ways, and do the rest using generators. In particular, always try to push ordering into the underlying layer.

Other

Examples

TODO

2.1.3 Basic Datastores

DictDatastore

Example:

```
>>> import pprint
>>> from datastore import DictDatastore, Key, Query
>>> ds = DictDatastore()
>>> for i in range(0, 3):
...   key = Key('/%d' % i)
...   ds.put(key.child('A'), '%d a value' % i)
...   ds.put(key.child('B'), '%d b value' % i)
...
>>> pprint.pprint(ds._items)
{'/0': {Key('/0/A'): '0 a value', Key('/0/B'): '0 b value'},
 '/1': {Key('/1/A'): '1 a value', Key('/1/B'): '1 b value'},
 '/2': {Key('/2/A'): '2 a value', Key('/2/B'): '2 b value'}}
>>> ds.get(Key('/1/A'))
'1 a value'
>>> for item in ds.query(Query(Key('/2'))):
...   print item
...
2 b value
2 a value
```

InterfaceMappingDatastore

Example:

```
>>> import pylibmc
>>> from datastore import InterfaceMappingDatastore, Key
>>> mc = pylibmc.Client(['127.0.0.1'])
>>> mc_ds = InterfaceMappingDatastore(mc, put='set', key=str)
>>> mc_ds.put(Key('Hello'), 'World')
>>> mc_ds.get(Key('Hello'))
'World'
>>> mc.get('/Hello')
'World'
>>> mc.set('/Hello', 'Goodbye!')
True
>>> mc_ds.get(Key('/Hello'))
'Goodbye!'
```

2.1.4 Shims

Sometimes common functionality can be compartmentalized into logic that can be plugged in or not. For example, serializing and deserializing data as it is stored or extracted is a very common operation. Likewise, applications may need to perform routine operations as data makes its way from the top-level logic to the underlying storage.

To address this use case in an elegant way, datastore uses the notion of a `shim` datastore, which implements all four main *datastore operations* in terms of an underlying child datastore. For example, a json serializer datastore could implement `get` and `put` as:

```
def get(self, key):
  value = self.child_datastore.get(key)
  return json.loads(value)

def put(self, key, value):
  value = json.dumps(value)
  self.child_datastore.put(key, value)
```

ShimDatastore

To implement a shim datastore, derive from `datastore.ShimDatastore` and override any of the operations.

KeyTransformDatastore

LowercaseKeyDatastore

NamespaceDatastore

SymlinkDatastore

2.1.5 Collections

Grouping datastores into datastore collections can significantly simplify complex access patterns. For example, caching datastores can be checked before accessing more costly datastores, or a group of equivalent datastores can act as shards containing large data sets.

As *shims*, datastore collections also derive from *datastore*, and must implement the four datastore operations (get, put, delete, query).

DatastoreCollection

Collections may derive from DatastoreCollection

TieredDatastore

Example:

```
>>> import pymongo
>>> import datastore.core
>>>
>>> from datastore.mongo import MongoDatastore
>>> from datastore.pylru import LRUCacheDatastore
>>> from datastore.filesystem import FileSystemDatastore
>>>
>>> conn = pymongo.Connection()
>>> mongo = MongoDatastore(conn.test_db)
>>>
>>> cache = LRUCacheDatastore(1000)
>>> fs = FileSystemDatastore('/tmp/.test_db')
>>>
>>> ds = datastore.TieredDatastore([cache, mongo, fs])
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

ShardedDatastore

Example:

```
>>> import datastore.core
>>>
>>> shards = [datastore.DictDatastore() for i in range(0, 10)]
>>>
>>> ds = datastore.ShardedDatastore(shards)
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

2.1.6 Serialize

Serializing shemes are often application-specific, and thus libraries often avoid imposing one. At the same time, it would be ideal if a variety of serializers were available and trivially pluggable into the data storage pattern through a simple interface.

The `pickle` protocol has established the `pickle.loads()` and `pickle.dumps()` serialization interface, which other serializers (like `json`) have adopted. This significantly simplifies things, but specific calls to serializers need to be added whenever inserting of extracting data with data storage libraries.

To flexibly solve this issue, datastore defines a `datastore.SerializerShimDatastore` which can be layered on top of any other datastore. As data is `put`, the serializer shim serializes it and `put``s it into the underlying ``child_datastore`. Correspondingly, on the way out (through `get` or `query`) the data is retrieved from the `child_datastore` and deserialized.

SerializerShimDatastore

datastore.serialize.shim

Serializers

The serializers that `datastore.SerializerShimDatastore` accepts must respond to the protocol outlined in `datastore.serialize.Serializer` (the `pickle` protocol).

serialize.Serializer

serialize.NonSerializer

serialize.prettyjson

serialize.Stack

serialize.map_serializer

Generators

Examples

TODO

2.1.7 datastore base class

2.1.8 Examples

Hello World

```
>>> import datastore.core
>>> ds = datastore.DictDatastore()
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

Package Hierarchy:

2.2 datastore Package

2.2.1 Subpackages

datastore.core

The `datastore.core` package contains the core parts of datastore, including the base Datastore classes, Key, serializers, collections, shims, etc.

The core package exists mainly because namespace packages cannot include members. datastore is a generic layer of abstraction for data store and database access. It is a **simple** API with the aim to enable application development in a datastore-agnostic way, allowing datastores to be swapped seamlessly without changing application code. Thus, one can leverage different datastores with different strengths without committing the application to one datastore throughout its lifetime.

datastore.basic

class datastore.core.basic.**CacheShimDatastore**(**args*, ***kwargs*)
Bases: datastore.core.basic.ShimDatastore

Wraps a datastore with a caching shim optimizes some calls.

contains(*key*)
Returns whether the object named by *key* exists. First checks cache_datastore.

delete(*key*)
Removes the object named by *key*. Writes to both cache_datastore and child_datastore.

get(*key*)
Return the object named by key or None if it does not exist. CacheShimDatastore first checks its cache_datastore.

put(*key*, *value*)
Stores the object *value* named by *key*‘self. Writes to both ‘‘cache_datastore‘ and child_datastore.

class datastore.core.basic.**Datastore**
Bases: object

A Datastore represents storage for any key-value pair.

Datastores are general enough to be backed by all kinds of different storage: in-memory caches, databases, a remote datastore, flat files on disk, etc.

The general idea is to wrap a more complicated storage facility in a simple, uniform interface, keeping the freedom of using the right tools for the job. In particular, a Datastore can aggregate other datastores in interesting ways, like sharded (to distribute load) or tiered access (caches before databases).

While Datastores should be written general enough to accept all sorts of values, some implementations will undoubtedly have to be specific (e.g. SQL databases where fields should be decomposed into columns), particularly to support queries efficiently.

contains(*key*)
Returns whether the object named by *key* exists.

The default implementation pays the cost of a get. Some datastore implementations may optimize this.

Args: key: Key naming the object to check.

Returns: boalean whether the object exists

delete(*key*)
Removes the object named by *key*.

Args: key: Key naming the object to remove.

get(*key*)
Return the object named by key or None if it does not exist.

None takes the role of default value, so no KeyError exception is raised.

Args: key: Key naming the object to retrieve

Returns: object or None

put(*key*, *value*)
Stores the object *value* named by *key*.

How to serialize and store objects is up to the underlying datastore. It is recommended to use simple objects (strings, numbers, lists, dicts).

Args: key: Key naming *value* value: the object to store.

query (*query*)
Returns an iterable of objects matching criteria expressed in *query*

Implementations of query will be the largest differentiating factor amongst datastores. All datastores **must** implement query, even using query's worst case scenario, see **:ref:class:'Query'** for details.

Args: query: Query object describing the objects to return.

Raturns: iterable cursor with all objects matching criteria

class datastore.core.basic.**DatastoreCollection**(*stores*=[])
Bases: datastore.core.basic.ShimDatastore

Represents a collection of datastores.

appendDatastore (*store*)
Appends datastore *store* to this collection.

datastore (*index*)
Returns the datastore at *index*.

insertDatastore (*index*, *store*)
Inserts datastore *store* into this collection at *index*.

removeDatastore (*store*)
Removes datastore *store* from this collection.

class datastore.core.basic.**DictDatastore**
Bases: datastore.core.basic.Datastore

Simple straw-man in-memory datastore backed by nested dicts.

contains (*key*)
Returns whether the object named by *key* exists.

Checks for the object in the collection corresponding to `key.path`.

Args: key: Key naming the object to check.

Returns: boalean whether the object exists

delete (*key*)
Removes the object named by *key*.

Removes the object from the collection corresponding to `key.path`.

Args: key: Key naming the object to remove.

get (*key*)
Return the object named by *key* or None.

Retrieves the object from the collection corresponding to `key.path`.

Args: key: Key naming the object to retrieve.

Returns: object or None

put (*key*, *value*)
Stores the object *value* named by *key*.

Stores the object in the collection corresponding to `key.path`.

Args: key: Key naming *value* value: the object to store.

query (*query*)
Returns an iterable of objects matching criteria expressed in *query*

Naively applies the query operations on the objects within the namespaced collection corresponding to `query.key.path`.

Args: query: Query object describing the objects to return.

Raturns: iterable cursor with all objects matching criteria

class `datastore.core.basic.`**DirectoryDatastore** (*datastore*)
Bases: `datastore.core.basic.ShimDatastore`

Datastore that tracks directory entries, like in a filesystem. All key changes cause changes in a collection-like directory.

For example:

```
>>> import datastore.core
>>>
>>> dds = datastore.DictDatastore()
>>> rds = datastore.DirectoryDatastore(dds)
>>>
>>> a = datastore.Key('/A')
>>> b = datastore.Key('/A/B')
>>> c = datastore.Key('/A/C')
>>>
>>> rds.get(a)
[]
>>> rds.put(b, 1)
>>> rds.get(b)
1
>>> rds.get(a)
['/A/B']
>>> rds.put(c, 1)
>>> rds.get(c)
1
>>> rds.get(a)
['/A/B', '/A/C']
>>> rds.delete(b)
>>> rds.get(a)
['/A/C']
>>> rds.delete(c)
>>> rds.get(a)
[]
```

delete (*key*)
Removes the object named by *key*. DirectoryDatastore removes the directory entry.

directory (*key*)
Retrieves directory entries for given key.

directory_values_generator (*key*)
Retrieve directory values for given key.

put (*key*, *value*)
Stores the object *value* named by ‘key‘self. DirectoryDatastore stores a directory entry.

query (*query*)
Returns objects matching criteria expressed in *query*. DirectoryDatastore uses directory entries.

class datastore.core.basic.**InterfaceMappingDatastore**(*service*, *get='get'*, *put='put'*, *delete='delete'*, *key=<type 'str'>*)

Bases: datastore.core.basic.Datastore

Represents simple wrapper datastore around an object that, though not a Datastore, implements data storage through a similar interface. For example, memcached and redis both implement a *get*, *set*, *delete* interface.

delete(*key*)

Removes the object named by *key* in *service*.

Args: key: Key naming the object to remove.

get(*key*)

Return the object in *service* named by *key* or None.

Args: key: Key naming the object to retrieve.

Returns: object or None

put(*key*, *value*)

Stores the object *value* named by *key* in *service*.

Args: key: Key naming *value*. value: the object to store.

class datastore.core.basic.**KeyTransformDatastore**(**args*, ***kwargs*)

Bases: datastore.core.basic.ShimDatastore

Represents a simple ShimDatastore that applies a transform on all incoming keys. For example:

```
>>> import datastore.core
>>> def transform(key):
...   return key.reverse
...
>>> ds = datastore.DictDatastore()
>>> kt = datastore.KeyTransformDatastore(ds, keytransform=transform)
None
>>> ds.put(datastore.Key('/a/b/c'), 'abc')
>>> ds.get(datastore.Key('/a/b/c'))
'abc'
>>> kt.get(datastore.Key('/a/b/c'))
None
>>> kt.get(datastore.Key('/c/b/a'))
'abc'
>>> ds.get(datastore.Key('/c/b/a'))
None
```

contains(*key*)

Returns whether the object named by key is in this datastore.

delete(*key*)

Removes the object named by keytransform(key).

get(*key*)

Return the object named by keytransform(key).

put(*key*, *value*)

Stores the object names by keytransform(key).

query(*query*)

Returns a sequence of objects matching criteria expressed in *query*

class datastore.core.basic.**LoggingDatastore**(*child_datastore*, *logger=None*)

Bases: datastore.core.basic.ShimDatastore

Wraps a datastore with a logging shim.

contains (*key*)
Returns whether the object named by *key* exists. LoggingDatastore logs the access.

delete (*key*)
Removes the object named by *key*. LoggingDatastore logs the access.

get (*key*)
Return the object named by key or None if it does not exist. LoggingDatastore logs the access.

put (*key*, *value*)
Stores the object *value* named by ‘key‘self. LoggingDatastore logs the access.

query (*query*)
Returns an iterable of objects matching criteria expressed in *query*. LoggingDatastore logs the access.

class datastore.core.basic.**LowercaseKeyDatastore** (**args*, ***kwargs*)
Bases: datastore.core.basic.KeyTransformDatastore

Represents a simple ShimDatastore that lowercases all incoming keys. For example:

```
>>> import datastore.core
>>> ds = datastore.DictDatastore()
>>> ds.put(datastore.Key('hello'), 'world')
>>> ds.put(datastore.Key('HELLO'), 'WORLD')
>>> ds.get(datastore.Key('hello'))
'world'
>>> ds.get(datastore.Key('HELLO'))
'WORLD'
>>> ds.get(datastore.Key('HeLlO'))
None
>>> lds = datastore.LowercaseKeyDatastore(ds)
>>> lds.get(datastore.Key('HeLlO'))
'world'
>>> lds.get(datastore.Key('HeLlO'))
'world'
>>> lds.get(datastore.Key('HeLlO'))
'world'
```

classmethod lowercaseKey (*key*)
Returns a lowercased *key*.

class datastore.core.basic.**NamespaceDatastore** (*namespace*, **args*, ***kwargs*)
Bases: datastore.core.basic.KeyTransformDatastore

Represents a simple ShimDatastore that namespaces all incoming keys. For example:

```
>>> import datastore.core
>>>
>>> ds = datastore.DictDatastore()
>>> ds.put(datastore.Key('/a/b'), 'ab')
>>> ds.put(datastore.Key('/c/d'), 'cd')
>>> ds.put(datastore.Key('/a/b/c/d'), 'abcd')
>>>
>>> nd = datastore.NamespaceDatastore('/a/b', ds)
>>> nd.get(datastore.Key('/a/b'))
None
>>> nd.get(datastore.Key('/c/d'))
'abcd'
>>> nd.get(datastore.Key('/a/b/c/d'))
None
```

```
>>> nd.put(datastore.Key('/c/d'), 'cd')
>>> ds.get(datastore.Key('/a/b/c/d'))
'cd'
```

namespaceKey(*key*)
Returns a namespaced *key*: namespace.child(key).

class datastore.core.basic.**NestedPathDatastore**(**args*, ***kwargs*)
Bases: datastore.core.basic.KeyTransformDatastore

Represents a simple ShimDatastore that shards/namespaces incoming keys.

Incoming keys are sharded into nested namespaces. The idea is to use the key name to separate into nested namespaces. This is akin to the directory structure that `git` uses for objects. For example:

```
>>> import datastore.core
>>>
>>> ds = datastore.DictDatastore()
>>> np = datastore.NestedPathDatastore(ds, depth=3, length=2)
>>>
>>> np.put(datastore.Key('/abcdefghijk'), 1)
>>> np.get(datastore.Key('/abcdefghijk'))
1
>>> ds.get(datastore.Key('/abcdefghijk'))
None
>>> ds.get(datastore.Key('/ab/cd/ef/abcdefghijk'))
1
>>> np.put(datastore.Key('abc'), 2)
>>> np.get(datastore.Key('abc'))
2
>>> ds.get(datastore.Key('/ab/ca/bc/abc'))
2
```

nestKey(*key*)
Returns a nested *key*.

static nestedPath(*path*, *depth*, *length*)
returns a nested version of *basename*, using the starting characters. For example:

```
>>> NestedPathDatastore.nested_path('abcdefghijk', 3, 2)
'ab/cd/ef'
>>> NestedPathDatastore.nested_path('abcdefghijk', 4, 2)
'ab/cd/ef/gh'
>>> NestedPathDatastore.nested_path('abcdefghijk', 3, 4)
'abcd/efgh/ijk'
>>> NestedPathDatastore.nested_path('abcdefghijk', 1, 4)
'abcd'
>>> NestedPathDatastore.nested_path('abcdefghijk', 3, 10)
'abcdefghij/k'
```

query(*query*)

class datastore.core.basic.**NullDatastore**
Bases: datastore.core.basic.Datastore

Stores nothing, but conforms to the API. Useful to test with.

delete(*key*)
Remove the object named by *key* (does nothing).

get(*key*)
Return the object named by key or None if it does not exist (None).

put(*key*, *value*)
Store the object *value* named by *key* (does nothing).

query(*query*)
Returns an iterable of objects matching criteria in *query* (empty).

class datastore.core.basic.**ShardedDatastore**(*stores=[]*, *shardingfn=<built-in function hash>*)
Bases: datastore.core.basic.DatastoreCollection

Represents a collection of datastore shards.

A datastore is selected based on a sharding function. Sharding functions should take a Key and return an integer.

WARNING: adding or removing datastores while mid-use may severely affect consistency. Also ensure the order is correct upon initialization. While this is not as important for caches, it is crucial for persistent datastores.

contains(*key*)
Returns whether the object is in this datastore.

delete(*key*)
Removes the object from the corresponding datastore.

get(*key*)
Return the object named by key from the corresponding datastore.

put(*key*, *value*)
Stores the object to the corresponding datastore.

query(*query*)
Returns a sequence of objects matching criteria expressed in *query*

shard(*key*)
Returns the shard index to handle *key*, according to sharding fn.

shardDatastore(*key*)
Returns the shard to handle *key*.

shard_query_generator(*query*)
A generator that queries each shard in sequence.

class datastore.core.basic.**ShimDatastore**(*datastore*)
Bases: datastore.core.basic.Datastore

Represents a non-concrete datastore that adds functionality between the client and a lower level datastore. Shim datastores do not actually store data themselves; instead, they delegate storage to an underlying child datastore. The default implementation just passes all calls to the child.

delete(*key*)
Removes the object named by *key*.

Default shim implementation simply calls `child_datastore.delete(key)` Override to provide different functionality.

Args: key: Key naming the object to remove.

get(*key*)
Return the object named by key or None if it does not exist.

Default shim implementation simply returns `child_datastore.get(key)` Override to provide different functionality, for example:

```
def get(self, key):
  value = self.child_datastore.get(key)
  return json.loads(value)
```

Args: key: Key naming the object to retrieve

Returns: object or None

put (*key*, *value*)
Stores the object *value* named by *key*.

Default shim implementation simply calls `child_datastore.put(key, value)` Override to provide different functionality, for example:

```
def put(self, key, value):
  value = json.dumps(value)
  self.child_datastore.put(key, value)
```

Args: key: Key naming *value*. value: the object to store.

query (*query*)
Returns an iterable of objects matching criteria expressed in *query*.

Default shim implementation simply returns `child_datastore.query(query)` Override to provide different functionality, for example:

```
def query(self, query):
  cursor = self.child_datastore.query(query)
  cursor._iterable = deserialized(cursor._iterable)
  return cursor
```

Args: query: Query object describing the objects to return.

Raturns: iterable cursor with all objects matching criteria

class `datastore.core.basic.`**SymlinkDatastore** (*datastore*)
Bases: `datastore.core.basic.ShimDatastore`

Datastore that creates filesystem-like symbolic link keys.

A symbolic link key is a way of naming the same value with multiple keys.

For example:

```
>>> import datastore.core
>>>
>>> dds = datastore.DictDatastore()
>>> sds = datastore.SymlinkDatastore(dds)
>>>
>>> a = datastore.Key('/A')
>>> b = datastore.Key('/B')
>>>
>>> sds.put(a, 1)
>>> sds.get(a)
1
>>> sds.link(a, b)
>>> sds.get(b)
1
>>> sds.put(b, 2)
```

```
>>> sds.get(b)
2
>>> sds.get(a)
2
>>> sds.delete(a)
>>> sds.get(a)
None
>>> sds.get(b)
None
>>> sds.put(a, 3)
>>> sds.get(a)
3
>>> sds.get(b)
3
>>> sds.delete(b)
>>> sds.get(b)
None
>>> sds.get(a)
3
```

get (*key*)
Return the object named by ‘key. Follows links.

link (*source_key*, *target_key*)
Creates a symbolic link key pointing from *target_key* to *source_key*

put (*key*, *value*)
Stores the object named by *key*. Follows links.

query (*query*)
Returns objects matching criteria expressed in *query*. Follows links.

sentinel = ‘datastore_link’

class datastore.core.basic.**TieredDatastore**(*stores*=[ ])
Bases: datastore.core.basic.DatastoreCollection

Represents a hierarchical collection of datastores.

Each datastore is queried in order. This is helpful to organize access order in terms of speed (i.e. read caches first).

Datastores should be arranged in order of completeness, with the most complete datastore last, as it will handle query calls.

Semantics:

- get : returns first found value
- put : writes through to all
- delete : deletes through to all
- contains : returns first found value
- query : queries bottom (most complete) datastore

contains (*key*)
Returns whether the object is in this datastore.

delete (*key*)
Removes the object from all underlying datastores.

get (*key*)
Return the object named by key. Checks each datastore in order.

put (*key*, *value*)
Stores the object in all underlying datastores.

query (*query*)
Returns a sequence of objects matching criteria expressed in *query*. The last datastore will handle all query calls, as it has a (if not the only) complete record of all objects.

datastore.key

class datastore.core.key.**Key** (*key*)
Bases: object

A Key represents the unique identifier of an object.

Our Key scheme is inspired by file systems and the Google App Engine key model.

Keys are meant to be unique across a system. Keys are hierarchical, incorporating more and more specific namespaces. Thus keys can be deemed 'children' or 'ancestors' of other keys:

```
Key('/Comedy')
Key('/Comedy/MontyPython')
```

Also, every namespace can be parametrized to embed relevant object information. For example, the Key *name* (most specific namespace) could include the object type:

```
Key('/Comedy/MontyPython/Actor:JohnCleese')
Key('/Comedy/MontyPython/Sketch:CheeseShop')
Key('/Comedy/MontyPython/Sketch:CheeseShop/Character:Mousebender')
```

child (*other*)
Returns the child Key by appending namespace *other*.

```
>>> Key('/Comedy/MontyPython').child('Actor:JohnCleese')
Key('/Comedy/MontyPython/Actor:JohnCleese')
```

instance (*other*)
Returns an instance Key, by appending a name to the namespace.

isAncestorOf (*other*)
Returns whether this Key is an ancestor of *other*.

```
>>> john = Key('/Comedy/MontyPython/Actor:JohnCleese')
>>> Key('/Comedy').isAncestorOf(john)
True
```

isDescendantOf (*other*)
Returns whether this Key is a descendant of *other*.

```
>>> Key('/Comedy/MontyPython').isDescendantOf(Key('/Comedy'))
True
```

isTopLevel ()
Returns whether this Key is top-level (one namespace).

list
Returns the *list* representation of this Key.

Note that this method assumes the key is immutable.

name
Returns the name of this Key, the value of the last namespace.

namespaces
Returns the list of namespaces of this Key.

parent
Returns the parent Key (all namespaces except the last).

```
>>> Key('/Comedy/MontyPython/Actor:JohnCleese').parent
Key('/Comedy/MontyPython')
```

path
Returns the path of this Key, the parent and the type.

classmethod randomKey()
Returns a random Key

classmethod removeDuplicateSlashes(*path*)
Returns the path string *path* without duplicate slashes.

reverse
Returns the reverse of this Key.

```
>>> Key('/Comedy/MontyPython/Actor:JohnCleese').reverse
Key('/Actor:JohnCleese/MontyPython/Comedy')
```

type
Returns the type of this Key, the field of the last namespace.

class datastore.core.key.**Namespace**
Bases: str

A Key Namespace is a string identifier.

A namespace can optionally include a field (delimited by ':')

Example namespaces:

```
Namespace('Bruces')
Namespace('Song:PhilosopherSong')
```

field
returns the *field* part of this namespace, if any.

namespace_delimiter = ':'

value
returns the *value* part of this namespace.

datastore.query

class datastore.core.query.**Cursor**(*query*, *iterable*)
Bases: object

Represents a query result generator.

apply_filter()
Naively apply query filters.

apply_limit()
Naively apply query limit.

apply_offset()
Naively apply query offset.

apply_order()
Naively apply query orders.

next()
Iterator next. Build up count of returned elements during iteration.

query

returned

skipped

class datastore.core.query.**Filter**(*field*, *op*, *value*)
Bases: object

Represents a Filter for a specific field and its value.

Filters are used on queries to narrow down the set of matching objects.

Args: field: the attribute name (string) on which to apply the filter.

op: the conditional operator to apply (one of ['<', '<=', '=', '!=', '>=', '>']).

value: the attribute value to compare against.

Examples:

```
Filter('name', '=', 'John Cleese')
Filter('age', '>=', 18)
```

conditional_operators = ['<', '<=', '=', '!=', '>=', '>']
Conditional operators that Filters support.

classmethod filter(*filters*, *iterable*)
Returns the elements in *iterable* that pass given *filters*

generator(*iterable*)
Generator function that iteratively filters given *items*.

static object_getattr(*obj*, *field*)
Object attribute getter. Can be overridden to match client data model. See datastore.query._object_getattr().

valuePasses(*value*)
Returns whether this value passes this filter

class datastore.core.query.**Order**(*order*)
Bases: object

Represents an Order upon a specific field, and a direction. Orders are used on queries to define how they operate on objects

Args:

order: an order in string form. This follows the format: [+-]name where + is ascending, - is descending, and name is the name of the field to order by. Note: if no ordering operator is specified, + is default.

Examples:

```
Order('+name')    #  ascending order by name
Order('-age')     # descending order by age
Order('score')    #  ascending order by score
```

isAscending()

isDescending()

keyfn(*obj*)
A key function to be used in pythonic sort operations.

classmethod multipleOrderComparison(*orders*)
Returns a function that will compare two items according to *orders*

static object_getattr(*obj*, *field*)
Object attribute getter. Can be overridden to match client data model. See `datastore.query._object_getattr()`.

order_operators = ['-', '+']
Ordering operators: + is ascending, - is descending.

classmethod sorted(*items*, *orders*)
Returns the elements in *items* sorted according to *orders*

class `datastore.core.query.`**Query**(*key*, *limit=None*, *offset=0*, *object_getattr=None*)
Bases: `object`

A Query describes a set of objects.

Queries are used to retrieve objects and instances matching a set of criteria from Datastores. Query objects themselves are simply descriptions, the actual Query implementations are left up to the Datastores.

copy()
Returns a copy of this query.

dict()
Returns a dictionary representing this query.

filter(**args*)
Adds a Filter to this query.

Args: see `Filter` constructor

Returns self for JS-like method chaining:

```
query.filter('age', '>', 18).filter('sex', '=', 'Female')
```

classmethod from_dict(*dictionary*)
Constructs a query from a dictionary.

static object_getattr(*obj*, *field*)
Attribute getter for the objects to operate on.

This function can be overridden in classes or instances of Query, Filter, and Order. Thus, a custom function to extract values to attributes can be specified, and the system can remain agnostic to the client's data model, without loosing query power.

For example, the default implementation works with attributes and items:

```
def _object_getattr(obj, field):
  # check whether this key is an attribute
  if hasattr(obj, field):
    value = getattr(obj, field)

  # if not, perhaps it is an item (raw dicts, etc)
  elif field in obj:
```

```
    value = obj[field]

  # return whatever we've got.
  return value
```

Or consider a more complex, application-specific structure:

```
def _object_getattr(version, field):

  if field in ['key', 'committed', 'created', 'hash']:
    return getattr(version, field)

  else:
    return version.attributes[field]['value']
```

order(*order*)
Adds an Order to this query.

Args: see `Order` constructor

Returns self for JS-like method chaining:

```
query.order('+age').order('-home')
```

datastore.core.query.**chain_gen**(*iterables*)
A generator that chains *iterables*.

datastore.core.query.**is_iterable**(*obj*)

datastore.core.query.**limit_gen**(*limit*, *iterable*)
A generator that applies a count *limit*.

datastore.core.query.**offset_gen**(*offset*, *iterable*, *skip_signal=None*)
A generator that applies an *offset*, skipping *offset* elements from *iterable*. If skip_signal is a callable, it will be called with every skipped element.

datastore.serialize

class datastore.core.serialize.**NonSerializer**
Bases: datastore.core.serialize.Serializer

Implements serializing protocol but does not serialize at all. If only storing strings (or already-serialized values).

classmethod dumps(*value*)
returns *value*.

classmethod loads(*value*)
returns *value*.

class datastore.core.serialize.**Serializer**
Bases: object

Serializing protocol. Serialized data must be a string.

classmethod dumps(*value*)
returns serialized *value*.

classmethod loads(*value*)
returns deserialized *value*.

class datastore.core.serialize.**SerializerShimDatastore**(*datastore*, *serializer=None*)
Bases: datastore.core.basic.ShimDatastore

Represents a Datastore that serializes and deserializes values.

As data is put, the serializer shim serializes it and put''s it into the underlying ''child_datastore. Correspondingly, on the way out (through get or query) the data is retrieved from the child_datastore and deserialized.

Args: datastore: a child datastore for the ShimDatastore superclass.

serializer: a serializer object (responds to loads and dumps).

deserializedValue(*value*)
Returns deserialized *value* or None.

get(*key*)
Return the object named by key or None if it does not exist. Retrieves the value from the child_datastore, and de-serializes it on the way out.

Args: key: Key naming the object to retrieve

Returns: object or None

put(*key*, *value*)
Stores the object *value* named by *key*. Serializes values on the way in, and stores the serialized data into the child_datastore.

Args: key: Key naming *value* value: the object to store.

query(*query*)
Returns an iterable of objects matching criteria expressed in *query* De-serializes values on the way out, using a *deserialized_gen* to avoid incurring the cost of de-serializing all data at once, or ever, if iteration over results does not finish (subject to order generator constraint).

Args: query: Query object describing the objects to return.

Raturns: iterable cursor with all objects matching criteria

serializedValue(*value*)
Returns serialized *value* or None.

serializer = <module 'json' from '/usr/lib/python2.7/json/__init__.pyc'>

class datastore.core.serialize.**Stack**
Bases: datastore.core.serialize.Serializer, list

represents a stack of serializers, applying each serializer in sequence.

dumps(*value*)
returns serialized *value*.

loads(*value*)
Returns deserialized *value*.

datastore.core.serialize.**deserialized_gen**(*serializer*, *iterable*)
Generator that yields deserialized objects from *iterable*.

class datastore.core.serialize.**map_serializer**
Bases: datastore.core.serialize.Serializer

map serializer that ensures the serialized value is a mapping type.

classmethod dumps(*value*)
returns mapping typed serialized *value*.

classmethod loads (*value*)
Returns mapping type deserialized *value*.

sentinel = '@wrapped'

datastore.core.serialize.**monkey_patch_bson**(*bson=None*)
Patch bson in pymongo to use loads and dumps interface.

class datastore.core.serialize.**prettyjson**
Bases: datastore.core.serialize.Serializer

json wrapper serializer that pretty-prints. Useful for human readable values and versioning.

classmethod dumps (*value*)
returns json serialized *value* (pretty-printed).

classmethod loads (*value*)
returns json deserialized *value*.

datastore.core.serialize.**serialized_gen**(*serializer*, *iterable*)
Generator that yields serialized objects from *iterable*.

datastore.core.serialize.**shim**(*datastore*, *serializer=None*)
Return a SerializerShimDatastore wrapping *datastore*.

Can be used as a syntacticly-nicer eay to wrap a datastore with a serializer:

```
my_store = datastore.serialize.shim(my_store, json)
```

datastore.core.util

datastore.core.util.fasthash

datastore.filesystem

filesystem datastore implementation.

Tested with:

- Journaled HFS+ (Mac OS X 10.7.2)

class datastore.filesystem.**FileSystemDatastore**(*root*, *case_sensitive=True*)
Bases: datastore.core.basic.Datastore

Simple flat-file datastore.

FileSystemDatastore will store objects in independent files in the host's filesystem. The FileSystemDatastore is initialized with a *root* path, under which to store all objects. Each object will be stored under its own file: *root*/*key*.obj

The *key* portion also replaces namespace parameter delimiters (:) with slashes, creating several nested directories. For example, storing objects under *root* path '/data' with the following keys:

```
Key('/Comedy:MontyPython/Actor:JohnCleese')
Key('/Comedy:MontyPython/Sketch:ArgumentClinic')
Key('/Comedy:MontyPython/Sketch:CheeseShop')
Key('/Comedy:MontyPython/Sketch:CheeseShop/Character:Mousebender')
```

will yield the file structure:

```
/data/Comedy/MontyPython/Actor/JohnCleese.obj
/data/Comedy/MontyPython/Sketch/ArgumentClinic.obj
/data/Comedy/MontyPython/Sketch/CheeseShop.obj
/data/Comedy/MontyPython/Sketch/CheeseShop/Character/Mousebender.obj
```

Implementation Notes:

Separating key namespaces (and their parameters) within directories allows granular querying for under a specific key. For example, a query with key:

```
Key('/data/Comedy:MontyPython/Sketch:CheeseShop')
```

will query for all objects under *Sketch:CheeseShop* independently of queries for:

```
Key('/data/Comedy:MontyPython/Sketch')
```

Also, using the *.obj* extension gets around the ambiguity of having both a *CheeseShop* object and directory:

```
/data/Comedy/MontyPython/Sketch/CheeseShop.obj
/data/Comedy/MontyPython/Sketch/CheeseShop/
```

Hello World:

```
>>> import datastore.filesystem
>>>
>>> ds = datastore.filesystem.FileSystemDatastore('/tmp/.test_datastore')
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

contains(*key*)
Returns whether the object named by *key* exists. Optimized to only check whether the file object exists.

Args: key: Key naming the object to check.

Returns: boalean whether the object exists

delete(*key*)
Removes the object named by *key*.

Args: key: Key naming the object to remove.

get(*key*)
Return the object named by key or None if it does not exist.

Args: key: Key naming the object to retrieve

Returns: object or None

ignore_list = []

object_extension = '.obj'

object_path(*key*)
return the object path for *key*.

path(*key*)
Returns the *path* for given *key*

put(*key*, *value*)
Stores the object *value* named by *key*.

Args: key: Key naming *value* value: the object to store.

query(*query*)
Returns an iterable of objects matching criteria expressed in *query* FSDatastore.query queries all the *.obj* files within the directory specified by the query.key.

Args: query: Query object describing the objects to return.

Raturns: Cursor with all objects matching criteria

relative_object_path(*key*)
Returns the relative path for object pointed by *key*.

relative_path(*key*)
Returns the relative path for given *key*

datastore.filesystem.**ensure_directory_exists**(*directory*)
Ensures *directory* exists. May make *directory* and intermediate dirs. Raises RuntimeError if *directory* is a file.

CHAPTER

THREE

INSTALL

From pypi (using pip):

```
sudo pip install datastore
```

From pypi (using setuptools):

```
sudo easy_install datastore
```

From source:

```
git clone https://github.com/jbenet/datastore/
cd datastore
sudo python setup.py install
```

3.1 License

datastore is under the MIT Licence

3.2 Contact

datastore is written by [Juan Batiz-Benet](https://github.com/jbenet). It was originally part of **'py-dronestore<https://github.com/jbenet/py-dronestore>'_** On December 2011, it was re-written as a standalone project.

Project Homepage: https://github.com/jbenet/datastore

Feel free to contact me. But please file issues in github first. Cheers!

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

FIVE

HELLO WORLDS

To illustrate the api and how it works across different data storage systems, here is Hello World in various datastores. Note the common code.

5.1 Hello Dict

```
>>> import datastore.core
>>>
>>> ds = datastore.DictDatastore()
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

5.2 Hello filesystem

```
>>> import datastore.filesystem
>>>
>>> ds = datastore.filesystem.FileSystemDatastore('/tmp/.test_datastore')
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

5.3 Hello Serialization

```
>>> import datastore.core
>>> import json
>>>
>>> ds_child = datastore.DictDatastore()
>>> ds = datastore.serialize.shim(ds_child, json)
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

5.4 Hello Tiered Access

```
>>> import pymongo
>>> import datastore.core
>>>
>>> from datastore.mongo import MongoDatastore
>>> from datastore.pylru import LRUCacheDatastore
>>> from datastore.filesystem import FileSystemDatastore
>>>
>>> conn = pymongo.Connection()
>>> mongo = MongoDatastore(conn.test_db)
>>>
>>> cache = LRUCacheDatastore(1000)
>>> fs = FileSystemDatastore('/tmp/.test_db')
>>>
>>> ds = datastore.TieredDatastore([cache, mongo, fs])
>>>
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

5.5 Hello Sharding

```
>>> import datastore.core
>>>
>>> shards = [datastore.DictDatastore() for i in range(0, 10)]
>>>
>>> ds = datastore.ShardedDatastore(shards)
>>>
```

```
>>> hello = datastore.Key('hello')
>>> ds.put(hello, 'world')
>>> ds.contains(hello)
True
>>> ds.get(hello)
'world'
>>> ds.delete(hello)
>>> ds.get(hello)
None
```

PYTHON MODULE INDEX

d