Park Documentation

Release 1.1.0

Peter Teichman

December 24, 2015

Contents

Park is a persistent key-value API for Python with ordered traversal of keys. Both keys and values are binary safe. It's similar in use to LevelDB, but has no dependencies outside the Python standard library.

It is meant to be extremely easy to use and can scale to a few gigabytes of data. It allows you to be lazy until it doesn't meet your needs. Use it until then.

It supports simple getting and setting of byte data:

```
>>> kv = park.SQLiteStore("numbers.park")
>>> kv.put("1", "one")
>>> kv.put("2", "two")
>>> kv.put("3", "three")
>>> kv.put("4", "four")

>>> kv.get("2")
'two'
```

Batched setting of data from an iterable:

```
>>> kv.put_many([("1", "one"), ("2", "two"), ("3", "three")])

>>> kv.get("3")
'three'
```

Lexically ordered traversal of keys and items, with start and end sentinels (inclusive):

```
>>> kv.put("1", "one")
>>> kv.put("2", "two")
>>> kv.put("3", "three")
>>> kv.put("11", "eleven")
>>> kv.put("12", "twelve")

>>> list(kv.keys())
['1', '11', '12', '2', '3']

>>> list(kv.keys(key_from="12"))
['12', '2', '3']

>>> list(kv.keys(key_from="12", key_to="2"))
['12', '2']

>>> list(kv.items(key_from="12"))
[('12', 'twelve'), ('2', 'two'), ('3', 'three')]
```

Iteration over all keys or items with a given prefix:

```
>>> kv.put("pet/dog", "Canis lupus familiaris")
>>> kv.put("pet/cat", "Felis catus")
>>> kv.put("pet/wolf", "Canis lupus")

>>> list(kv.prefix_keys("pet/"))
['pet/cat', 'pet/dog', 'pet/wolf']

>>> list(kv.prefix_keys("pet/", strip_prefix=True))
['cat', 'dog', 'wolf']

>>> list(kv.prefix_items("pet/", strip_prefix=True))
[('cat', 'Felis catus'),
 ('dog', 'Canis lupus familiaris'),
 ('wolf', 'Canis lupus')]
```

It plays well with generators, so you can e.g. park all the counting numbers (this will take a while):

```
def numbers():
    for num in itertools.count(1):
        key = value = str(num)
        yield key, value

kv.put_many(numbers())
```

Or recursively park a directory's contents (keyed by relative paths) from the local filesystem:

```
def file_item(filename):
    with open(filename, "r") as fd:
        return filename, fd.read()

kv.put_many(file_item(os.path.join(root, name))
            for root, dirs, files in os.walk(directory)
            for name in files)
```

CHAPTER 1

Details, details.

class `park.`**`SQLiteStore`**`(`*path*`)`

A KVStore in an SQLite database. This is what you want to use.

Parameters **`path`** (*str*) – The filesystem path for the database, which will be created if it doesn't exist.

See *`park.KVStore`* for what you can do with it.

SQLiteStore uses an embarrassingly simple SQL schema:

```
CREATE TABLE kv (
    key BLOB NOT NULL PRIMARY KEY,
    value BLOB NOT NULL)
```

There are a few implications of this schema you might need to be aware of.

1. Declaring `key` as PRIMARY KEY automatically indexes it, which gives constant time ordered traversal of keys and O(log n) lookup. However, SQLite 3 indexes the keys separately from the table data, which means your keys are effectively stored twice in the database. A primary key also means the index can't be dropped during bulk inserts.
2. Using BLOBs for both columns keeps them binary safe, but it means everything going in must be type `bytes`. Python `str` strings are converted automatically, but if you're dealing with Unicode data you'll need to encode it to bytes first. UTF-8 is a fine option:

```
>>> kv.put("key", value.encode("utf-8"))
>>> kv.get("key").decode("utf-8")
```

class `park.`**`KVStore`**

An abstract key-value interface with support for range iteration.

`close``()`

Release any resources associated with a KVStore.

This is used to support the Python context manager protocol with semantics similar to `contextlib.closing()`. That means you can use any concrete implementation of KVStore like:

```
with park.SQLiteStore("/path/to/db") as kv:
    kv.put("my_key", "my_value")
```

`contains``(`*key*`)`

True if the store contains key.

`delete``(`*key*`)`

Remove a key from the store.

Parameters **`key`** (*bytes*) – The key to remove.

`delete_many`(*keys*)
Remove many keys from the store.

Parameters **`keys`** – An iterable producing keys to remove.

`get`(*key*, *default=None*)
Get the value associated with a key.

Parameters

- **`key`** (*bytes*) – The key to retrieve.
- **`default`** – A default value to return if the key is not present in the store.

Returns The value associated with `key`.

`items`(*key_from=None*, *key_to=None*)
Get a lexically sorted range of (key, value) tuples.

Parameters

- **`key_from`** (*bytes*) – Lower bound (inclusive), or None for unbounded.
- **`key_to`** (*bytes*) – Upper bound (inclusive), or None for unbounded.

Yields All (key, value) pairs from the store where `key_from <= key <= key_to`.

`keys`(*key_from=None*, *key_to=None*)
Get a lexically sorted range of keys.

Parameters

- **`key_from`** (*bytes*) – Lower bound (inclusive), or None for unbounded.
- **`key_to`** (*bytes*) – Upper bound (inclusive), or None for unbounded.

Yields All keys from the store where `key_from <= key <= key_to`.

`prefix_items`(*prefix*, *strip_prefix=False*)
Get all (key, value) pairs with keys that begin with `prefix`.

Parameters

- **`prefix`** (*bytes*) – Lexical prefix for keys to search.
- **`strip_prefix`** (*bool*) – True to strip the prefix from yielded items.

Yields All (key, value) pairs in the store where the keys begin with the `prefix`.

`prefix_keys`(*prefix*, *strip_prefix=False*)
Get all keys that begin with `prefix`.

Parameters

- **`prefix`** (*bytes*) – Lexical prefix for keys to search.
- **`strip_prefix`** (*bool*) – True to strip the prefix from yielded items.

Yields All keys in the store that begin with `prefix`.

`put`(*key*, *value*)
Put a key-value pair into the store.

If the key is already present, this replaces its value. Both the key and value are binary safe.

Parameters

- **`key`** (*bytes*) – The key to set.
- **`value`** (*bytes*) – The value to set the key to.

`put_many`(*items*)

Put many key-value pairs.

This method may take advantage of performance or atomicity features of the underlying store. It does not guarantee that all items will be set in the same transaction, only that transactions may be used for performance.

Parameters **`items`** – An iterable producing (key, value) tuples.

Index