PQ Documentation

Release 1.6.1

Nov 14, 2018

Contents

A transactional queue system for PostgreSQL written in Python.

It allows you to push and pop items in and out of a queue in various ways and also provides two scheduling options: delayed processing and prioritization.

The system uses a single table that holds all jobs across queues; the specifics are easy to customize.

The system currently supports only the psycopg2 database driver - or psycopg2cffi for PyPy.

The basic queue implementation is similar to Ryan Smith's queue_classic library written in Ruby, but uses advisory locks for concurrency control.

In terms of performance, the implementation clock in at about 1,000 operations per second. Using the PyPy interpreter, this scales linearly with the number of cores available.

CHAPTER 1

Getting started

All functionality is encapsulated in a single class `PQ`.

```
class PQ(conn=None, pool=None, table='queue', debug=False)
```

Example usage:

```
from psycopg2 import connect
from pq import PQ

conn = connect('dbname=example user=postgres')
pq = PQ(conn)
```

For multi-threaded operation, use a connection pool such as `psycopg2.pool.ThreadedConnectionPool`.

You probably want to make sure your database is created with the `utf-8` encoding.

To create and configure the queue table, call the `create()` method.

```
pq.create()
```

The table name defaults to `'queue'`. To use a different name, pass it as a string value as the `table` argument for the `PQ` class (illustrated above).

CHAPTER 2

Queues

The `pq` object exposes queues through Python's dictionary interface:

```
queue = pq['apples']
```

The `queue` object provides `get` and `put` methods as explained below, and in addition, it also works as a context manager where it manages a transaction:

```
with queue as cursor:
    ...
```

The statements inside the context manager are either committed as a transaction or rejected, atomically. This is useful when a queue is used to manage jobs because it allows you to retrieve a job from the queue, perform a job and write a result, with transactional semantics.

CHAPTER 3

Methods

Use the `put(data)` method to insert an item into the queue. It takes a JSON-compatible object such as a Python dictionary:

```
queue.put({'kind': 'Cox'})
queue.put({'kind': 'Arthur Turner'})
queue.put({'kind': 'Golden Delicious'})
```

Items are pulled out of the queue using `get(block=True)`. The default behavior is to block until an item is available with a default timeout of one second after which a value of `None` is returned.

```
def eat(kind):
    print 'umm, %s apples taste good.' % kind

job = queue.get()
eat(**job.data)
```

The `job` object provides additional metadata in addition to the `data` attribute as illustrated by the string representation:

```
>>> job
<pq.Job id=77709 size=1 enqueued_at="2014-02-21T16:22:06Z" schedule_at=None>
```

The `get` operation is also available through iteration:

```
for job in queue:
    if job is None:
        break

    eat(**job.data)
```

The iterator blocks if no item is available. Again, there is a default timeout of one second, after which the iterator yields a value of `None`.

An application can then choose to break out of the loop, or wait again for an item to be ready.

```
for job in queue:
    if job is not None:
        eat(**job.data)

    # This is an infinite loop!
```

CHAPTER 4

Scheduling

Items can be scheduled such that they're not pulled until a later time:

```
queue.put({'kind': 'Cox'}, '5m')
```

In this example, the item is ready for work five minutes later. The method also accepts `datetime` and `timedelta` objects.

CHAPTER 5

Priority

If some items are more important than others, a time expectation can be expressed:

```
queue.put({'kind': 'Cox'}, expected_at='5m')
```

This tells the queue processor to give priority to this item over an item expected at a later time, and conversely, to prefer an item with an earlier expected time.

The scheduling and priority options can be combined:

```
queue.put({'kind': 'Cox'}, '1h', '2h')
```

This item won't be pulled out until after one hour, and even then, it's only processed subject to it's priority of two hours.

CHAPTER 6

Pickles

If a queue name is provided as `<name>/pickle` (e.g. `'jobs/pickle'`), items are automatically pickled and unpickled using Python's built-in `cPickle` module:

```
queue = pq['apples/pickle']

class Apple(object):
    def __init__(self, kind):
        self.kind = kind

queue.put(Apple('Cox'))
```

The old pickle protocol `0` is used to ensure the pickled data is encoded as `ascii` which should be compatible with any database encoding.

CHAPTER 7

Tasks

pq comes with a higher level API that helps to manage tasks.

```
from pq.tasks import PQ

pq = PQ(...)

queue = pq['default']

@queue.task(schedule_at='1h')
def eat(kind):
    print 'umm, %s apples taste good.' % kind

eat('Cox')

queue.work()
```

tasks's jobs can optionally be re-scheduled on failure:

```
@queue.task(schedule_at='1h', max_retries=2, retry_in='10s')
def eat(kind):
    # ...
```

Time expectations can be overriden at task call:

```
eat('Cox', _expected_at='2m', _schedule_at='1m')
```

CHAPTER 8

Thread-safety

All objects are thread-safe as long as a connection pool is provided where each thread receives its own database connection.