One call – fine, many calls – boo!

The Batch Aspect.

Patrick Schemitz, solute GmbH ps@solute.de
(absent co-author: Jonathan Oberländer, solute GmbH)
Karlsruhe Python Meetup 2024-11-20

soluteD

Table of Contents

- Context: solute, billiger.de, Partner Business
- Offers from the Syndication API:
  - Setting the Stage
  - A Naive Implementation
  - An Ugly Optimization
  - A Nicer Solution
- Conclusion

Prelude: An API Endpoint.

– **solute GmbH**: price comparison, 100+ mio offers from thousands of shops in Europe.

– **billiger.de** = our most important website, but also **Syndication Partner** business.

– Syndication API to get a couple of offers:

```
@api.route.get("/offers/<offer_ids>")
def get_offers(offer_ids):
    offers_from_db = get_offers_from_db(offer_ids)
    return [
        {
            **offer,  # contains a field “shop_id”
        }
        for offer in offers_from_db
    ]
```

– Shop info normalized, and shop ID not enough! Need to join shop info.

An API Endpoint, but with Shops

– Join shop info to each of the offers:

```
@api.route.get("/offers/<offer_ids>")
def get_offers(offer_ids):
    offers_from_db = get_offers_from_db(offer_ids)
    return [
        {
            "shop": get_shop_from_redis(offer["shop_id"]),
            **offer,
        }
        for offer in offers_from_db
    ]
```

– Code is self-explanatory, but **slow af!**

– Too many roundtrips to Redis! (Even with caching.)

An API Endpoint, Shops Batched

– Batch join shop info to each of the offers:

```
@api.route.get("/offers/<offer_ids>")
def get_offers(offer_ids):
    offers_from_db = get_offers_from_db(offer_ids)

    shop_ids = {offer["shop_id"] for offer in offers_from_db}
    shop_data = {}
    for batch in iter(lambda: itertools.islice(shop_ids, 20), []):
        shop_data.update(get_shops_from_redis(batch))

    return [
        {
            "shop": shop_data[offer[“shop_id”]],
            **offer,
        }
        for offer in offers_from_db
    ]
```

– Code is faster, but **ugly af!** Mixes batching and business logic.

An Ideal API Endpoint, envisioned

– We want fast and beautiful:

```
@api.route.get("/offers/<offer_ids>")
def get_offers(offer_ids):
    offers_from_db = get_offers_from_db(offer_ids)
    with batched():
        return [
            {
                "shop": get_shops_from_redis(offer["shop_id"]),
                **offer,
            }
            for offer in offers_from_db
        ]
```

– Code is self-explanatory *and* batched (and thus, fast)

– **But HOW?!**

Making ID Resolution Batchable

– We need to bridge the 1 vs. many resolver gap
– We need to postpone ID resolution

```
def get_shops_from_redis(ids):
    return {
        row["id"]: row
        for row in redis.mget(ids)
    }
```

Making ID Resolution Batchable

– We need to bridge the 1 vs. many resolver gap
– We need to postpone ID resolution

```
@batchable
def get_shops_from_redis(ids):
    return {
        row["id"]: row
        for row in redis.mget(ids)
    }

class batchable:
    def __init__(self, fn):
        self.fn = fn
        self.unresolved = []

    def __call__(self, id):
        if len(self.unresolved) >= 20:
            self.resolve()  # TODO: resolve placeholders
        placeholder = Placeholder(id)
        self.unresolved.append(placeholder)
        return placeholder
```

The Placeholder

– gc module can figure out who's using an object (the “referrers”)
→ We can replace the placeholders in lists and dicts (i.e. in list & dict comprehensions)

```
class Placeholder:
    def __init__(self, id):
        self.id = id

    def replace(self, replacement):
        gc.collect()  # so we can never get half-dead objects
        for referrer in gc.get_referrers(self):
            if isinstance(referrer, dict):
                for key in referrer:
                    if referrer[key] is self:  # object identity
                        referrer[key] = replacement
            elif isinstance(referrer, list):
                for i, element in enumerate(referrer):
                    if element is self:
                        referrer[i] = replacement
```

Batchable and Resolvable

– We can now resolve the placeholders in a `@batchable` function call:

```
@batchable
def get_shops_from_redis(ids):
    ...

class batchable:
    def __call__(self, id):
        if len(self.unresolved) >= 20:
            self.resolve()
        placeholder = Placeholder(id)
        self.unresolved.append(placeholder)
        return placeholder

    def resolve(self):
        ids = {placeholder.id for placeholder in self.unresolved}
        result = self.fn(ids)
        for placeholder in self.unresolved:
            placeholder.replace(result[placeholder.id])
        self.unresolved[:] = []  # clear unresolved queue
```

That Last Batch

– We must make sure the last placeholders are resolved as well
– Perfect for context manager!

```
from contextlib import contextmanager
BATCHABLES = []

class batchable:
    def __init__(self, fn):
        BATCHABLES.append(self)
        ...

@contextmanager
def batched():
    try:
        yield
    finally:
        For batchable in BATCHABLES:
            batchable.resolve()
```

An Ideal API Endpoint, explained

– How does the fast and beautiful code work:

```
@batchable
def get_shops_from_redis(ids):
    return {
        row["id"]: row
        for row in redis.mget(ids)
    }

@api.route.get("/offers/<offer_ids>")
def get_offers(offer_ids):
    offers_from_db = get_offers_from_db(offer_ids)
    with batched():
        return [
            {
                "shop": get_shops_from_redis(offer["shop_id"]),
                **offer,
            }
            for offer in offers_from_db
        ]
```

Summary

- Straightforward approach, has limitations:
  - Cannot be nested
  - Only one lookup
  - Works only for dicts and lists
  - gc implies CPython
  - Thread safety
  - We don’t actually use this!

- Better implementation (but we don’t use this, either): https://github.com/L3viathan/batchable

- Python supports aspect stuff surprisingly well
- Learning Days are fun!