**SHAHRIAR TAJBAKHSH**

AUGUST 20, 2017

# Understanding write-through, write-around and write-back caching (with Python)

This post explains the three basic cache writing policies: write-through, write-around and write-back. Although caching is not a language-dependent thing, we'll use basic Python code as a way of making the explanation and logic clearer and hopefully easier to understand. Running the Python code could also be helpful for simulating and playing around with different these caching policies.

A cache is a hardware or software component that stores data so future requests for that data can be served faster. Of course, this data is stored in a place where retrieval is faster than your actual backing store. For example, if you store a value in memory, it'll usually be quicker to access than hitting the database and trying to read it from disk.

So, to start, let's have two Python classes to represent our backing store and our cache. You can read from and write to each of them.

```python
import time


class BackingStore:
    def __init__(self):
        self.data = []

    def write(self, datum):
        print('Started writing to backing store.')
        time.sleep(2)  # Writing to disk is slow
        self.data.append(datum)
        print('Finished writing to backing store.')

    def read(self, index):
        print('Started reading from backing store.')
        time.sleep(2)  # Reading from disk is slow
        print('Finished reading from backing store.')
        return self.data[index]


class Cache:
    def __init__(self):
        self.data = []

    def write(self, datum):
        print('Started writing to cache.')
        self.data.append(datum)
        print('Finished writing to cache.')
    def read(self, index):
        print('Started reading from backing store.')
        print('Finished reading from backing store.')
        return self.data[index]
```

## Write-through

Using the write-through policy, data is written to the cache and the backing store location at the same time. The significance here is not the order in which it happens or whether it happens in parallel. The significance is that I/O completion is only confirmed once the data has been written to both places.

```
def write_through(cache, backing_store, datum):
    cache.write(datum)
    backing_store.write(datum)
```

**Advantage:** Ensures fast retrieval while making sure the data is in the backing store and is not lost in case the cache is disrupted.

**Disadvantage:** Writing data will experience latency as you have to write to two places every time.

### What is it good for?

The write-through policy is good for applications that write and then re-read data frequently. This will result in slightly higher write latency but low read latency. So, it’s ok to spend a bit longer writing once, but then benefit from reading frequently with low latency.

## Write-around

Using the write-around policy, data is written only to the backing store without writing to the cache. So, I/O completion is confirmed as soon as the data is written to the backing store.

```
def write_around(backing_store, datum):
    backing_store.write(datum)
```

**Advantage:** Good for not flooding the cache with data that may not subsequently be re-read.

**Disadvsntage:** Reading recently written data will result in a cache miss (and so a higher latency) because the data can only be read from the slower backing store.

### What is it good for?

The write-around policy is good for applications that don’t frequently re-read recently written data. This will result in lower write latency but higher read latency which is a acceptable trade-off for these scenarios.

## Write-back

Using the write-back policy, data is written to the cache and Then I/O completion is confirmed. The data is then typically also written to the backing store in the background but the completion confirmation is not blocked on that.

```
def write_back(cache, datum):
    cache.write(datum)
    # Maybe kick-off writing to backing store asynchronously, but don't
wait for it.
```

**Advantage:** Low latency and high throughput for write-intensive applications.

**Disadvantage:** There is data availability risk because the cache could fail (and so suffer from data loss) before the data is persisted to the backing store. This result in the data being lost.

### What is it good for?

The write-back policy is the best performer for mixed workloads as both read and write I/O have similar response time levels. In reality, you can add resiliency (e.g. by duplicating writes) to reduce the likelihood of data loss.

## Which one should I use?

If this post is all you know about caching policies then you’ll need to do more research. The post covered three basic caching policies at a _very high level just to give you a basic understanding and to spark an interest. In practice, there are many other (often hybrid) policies and lots of subtle nuggets to consider when implementing them.

Discussion on hackernews and reddit.

677 KUDOS

Tweet

Share 10

**NOW READ THIS**

## Underscores in Python

This post discusses the use of the _ character in Python. Like with many things in Python, we'll see that different usages of _ are mostly (not always!) a matter of convention. Single Lone Underscore (_) # This is typically used in 3... **Continue →**

**SHAHRIAR TAJBAKHSH** @STajbakhsh