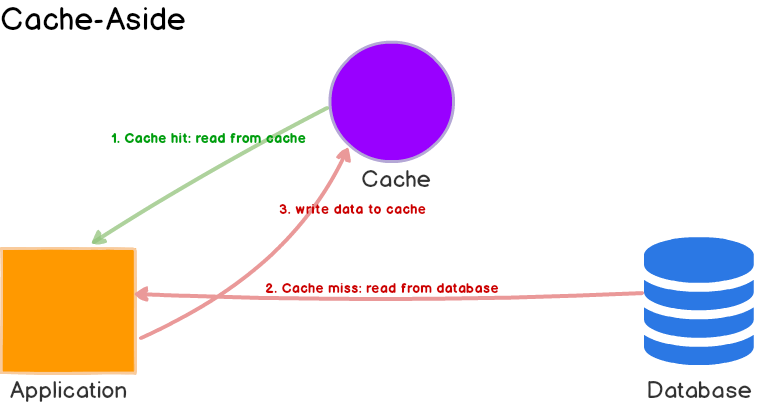
# **Caching Strategies**

**Cache-Aside**

This is perhaps the most commonly used caching approach, at least in the projects that I worked on. The cache sits on the *side* and the application directly talks to both the cache and the database.



Here’s what’s happening:

1. The application first checks the cache.
2. If the data is found in cache, we’ve *cache hit*. The data is read and returned to the client.
3. If the data is **not found** in cache, we’ve *cache miss*. The application has to do some **extra work**. It queries the database to read the data, returns it to the client and stores the data in cache so the subsequent reads for the same data results in a cache hit.

**Use Cases, Pros and Cons**

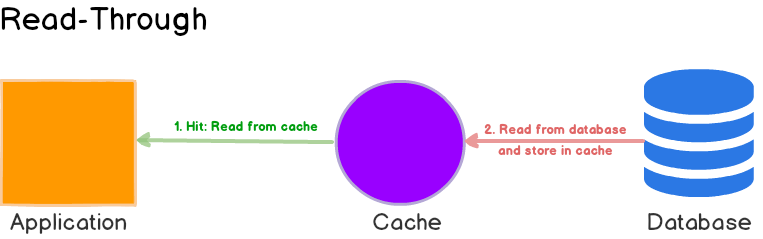
Cache-aside caches are usually general purpose and work best for **read-heavy workloads**. *Memcached* and *Redis* are widely used. Systems using cache-aside are **resilient to cache failures**. If the cache cluster goes down, the system can still operate by going directly to the database. (Although, it doesn’t help much if cache goes down during peak load. Response times can become terrible and in worst case, the database can stop working.)

Another benefit is that the data model in cache can be different than the data model in database. E.g. the response generated as a result of multiple queries can be stored against some request id.

When cache-aside is used, the most common write strategy is to write data to the database directly. When this happens, cache may become inconsistent with the database. To deal with this, developers generally use time to live (TTL) and continue serving stale data until TTL expires. If data freshness must be guaranteed, developers either **invalidate the cache entry** or use an appropriate write strategy, as we’ll explore later.

**Read-Through Cache**

Read-through cache sits in-line with the database. When there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.



Both cache-aside and read-through strategies load data **lazily**, that is, only when it is first read.

**Use Cases, Pros and Cons**

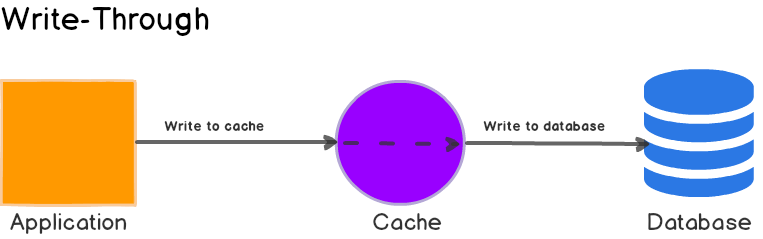
While read-through and cache-aside are very similar, there are at least two key differences:

1. In cache-aside, the application is responsible for fetching data from the database and populating the cache. In read-through, this logic is usually supported by the library or stand-alone cache provider.
2. Unlike cache-aside, the data model in read-through cache cannot be different than that of the database.

Read-through caches work best for **read-heavy** workloads when the same data is requested many times. For example, a news story. The disadvantage is that when the data is requested the first time, it always results in cache miss and incurs the extra penalty of loading data to the cache. Developers deal with this by ‘*warming*’ or ‘pre-heating’ the cache by issuing queries manually. Just like cache-aside, it is also possible for data to become inconsistent between cache and the database, and solution lies in the write strategy, as we’ll see next.

**Write-Through Cache**

In this write strategy, data is first written to the cache and then to the database. The cache sits in-line with the database and writes always go *through* the cache to the main database.



**Use Cases, Pros and Cons**

On its own, write-through caches don’t seem to do much, in fact, they introduce extra write latency because data is written to the cache first and then to the main database. But when paired with read-through caches, we get all the benefits of read-through and we also get data consistency guarantee, freeing us from using cache invalidation techniques.

[DynamoDB Accelerator (DAX)](https://aws.amazon.com/dynamodb/dax/) is a good example of read-through / write-through cache. It sits inline with DynamoDB and your application. Reads and writes to DynamoDB can be done through DAX. (Side note: If you are planning to use DAX, please make sure you familiarize yourself with [its data consistency model](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAX.consistency.html) and how it interplays with DynamoDB.)

**Write-Around**

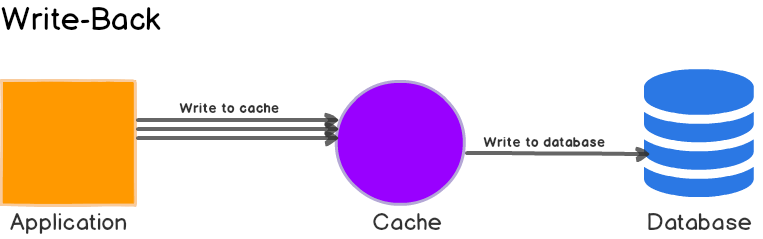
Here, data is written directly to the database and only the data that is read makes it way into the cache.

**Use Cases, Pros and Cons**

Write-around can be combine with read-through and provides good performance in situations where data is written once and read less frequently or never. For example, real-time logs or chatroom messages. Likewise, this pattern can be combined with cache-aside as well.

**Write-Back**

Here, the application writes data to the cache which acknowledges immediately and after some *delay*, it writes the data *back* to the database.



This is sometimes called write-behind as well.

**Use Cases, Pros and Cons**

Write back caches improve the write performance and are good for **write-heavy** workloads. When combined with read-through, it works good for mixed workloads, where the most recently updated and accessed data is always available in cache.

It’s resilient to database failures and can tolerate some database downtime. If batching or coalescing is supported, it can reduce overall writes to the database, which decreases the load and **reduces costs**, if the database provider charges by number of requests e.g. DynamoDB. Keep in mind that **DAX is write-through** so you won’t see any reductions in costs if your application is write heavy. (When I first heard of DAX, this was my first question - DynamoDB can be very expensive, but damn you Amazon.)

Some developers use Redis for both cache-aside and write-back to better absorb spikes during peak load. The main disadvantage is that if there’s a cache failure, the data may be permanently lost.

Most relational databases storage engines (i.e. InnoDB) have write-back cache enabled by default in their internals. Queries are first written to memory and eventually flushed to the disk.

## Workflow of Pub-Sub Messaging

In Apache Kafka, the stepwise workflow of the Pub-Sub Messaging is:

* At regular intervals, **Kafka Producers** send the message to a topic.
* **Kafka Brokers** stores all messages in the partitions configured for that particular topic, ensuring equal distribution of messages between partitions. For example, Kafka will store one message in the first partition and the second message in the second partition if the producer sends two messages and there are two partitions.
* Moreover, **Kafka Consumer** subscribes to a specific topic.
* Once the consumer subscribes to a topic, Kafka offers the current offset of the topic to the consumer and save the offset in the Zookeeper ensemble.
* Also, the consumer will request the Kafka in a regular interval, for new messages (like 100 Ms).
* Kafka will forward the messages to the consumers as soon as received from producers.
* The consumer will receive the message and process it.
* Then Kafka broker receives an acknowledgment of the message processed.
* Further, the offset is changed and updated to the new value as soon as Kafka receives an acknowledgment. Even during server outrages, the consumer can read the next message correctly, because ZooKeeper maintains the offsets.
* However, until the consumer stops the request, the flow repeats.
* As a benefit, the consumer can rewind/skip any offset of a topic at any time and also can read all the subsequent messages, as a par desire.

## Workflow of Kafka Queue Messaging/Consumer Group

A group of Kafka consumers having the same Group ID can subscribe to a topic, instead of a single consumer, in a queue messaging system.

However, with the same Group ID all consumers, those are subscribing to a topic are considered as a single group and share the messages. This system’s workflow is:

* In regular intervals, Kafka Producers send the message to a Kafka topic.
* As similar to the earlier scenario, here also Kafka stores all messages in the partitions configured for that particular topic.
* Moreover, a single consumer in Kafka subscribes to a specific topic.
* In the same way as Pub-Sub Messaging, Kafka interacts with the consumer until new consumer subscribes to the same topic.
* As the new customers arrive, share mode starts in the operations and shares the data between two Kafka consumers. Moreover, until the number of Kafka consumers equals the number of partitions configured for that particular topic, the sharing repeats.
* Although, the new consumer in Kafka will not receive any further message, once the number of Kafka consumers exceeds the number of partitions. It happens until any one of the existing consumer unsubscribes. This scenario arises because in Kafka there is a condition that each Kafka consumer will have a minimum of one partition and if no partition remains blank, then new consumers will have to wait.
* In addition, we also call it Kafka Consumer Group. Hence, Apache Kafka will offer the best of both the systems in a very simple and efficient manner

## RabbitMQ

If you do start to consider a queue-based solution, CloudAMQP offers to host the message queue with [RabbitMQ.](http://www.rabbitmq.com/) RabbitMQ is open source message-oriented middleware that implements the [Advanced Message Queuing Protocol (AMQP).](http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol) AMQP has features like queuing, routing, reliability, and security

# UBER SYSTEM DESIGN FOR TAXI LOCATION

Taxi/Driver Real Time Location

Load Balancer

Shard 1

DB

Shard 4

DB

Shard 2

DB

Shard 3

DB

Load Balancer

UBER Taxi

**10s, Websocket**

**Location (x,y) Co-ordinates**

**Taxi Drivers**

**600K Active Drivers**

**Time Shift - 8 hrs**

**Total Shifts Per Day-3**

**600K / 3 = 200K Drivers per Shift**

**200K / 10 s = 20K requests per second**

**At Peak Hours: 20K \* 2 = 40K**