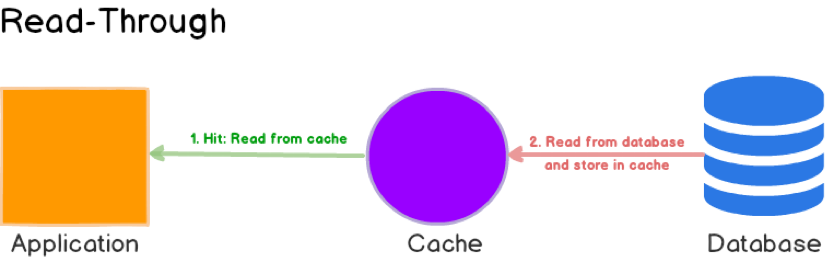


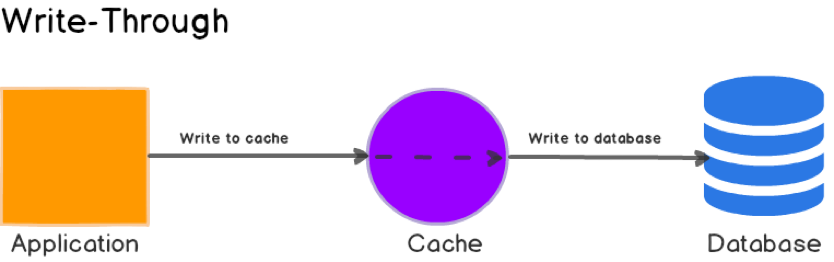
1. The **application first checks the cache**.
2. If the **data is found in the cache, we’ve cache hit**. The data is read and returned to the client.
3. If the **data is not found in the cache, we’ve cache miss**. The **application queries the database to read the data, returns it to the client, and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

**Cache-aside** caches **work best for read-heavy workloads**. Systems using cache-aside are **resilient to cache failures**. When cache-aside is used, the most **common write strategy is to write data to the database directly**. When this happens, the **cache may become inconsistent with the database**. To deal with this, developers generally use the **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.



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

**Read-through** caches **work best for read-heavy workloads** when the **same data is requested many times**. The disadvantage is that when the **data is requested the first time, it always results in a cache miss** and incurs the extra penalty of loading data to the cache. Just like cache-aside, it is also possible for data to become inconsistent between cache and the database, and the solution lies in the write strategy.



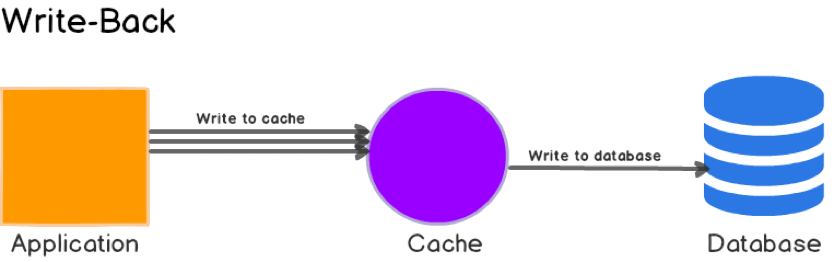
In this writing strategy, **data is first written to the cache and then to the database**. The **cache sits in line with the database** and **writes always go through the cache to the main database**.

On their 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 a data consistency guarantee**, freeing us from using cache invalidation techniques.

**Write-Around**

In this writing strategy, **data is written directly to the database and only the data that is read makes its way into the cache**.

**Write-around** can be **combined with a 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.



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

**Write back** caches **improve the write performance** and are **good for write-heavy workloads**. 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.

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**.