Database query result caching

Introduction

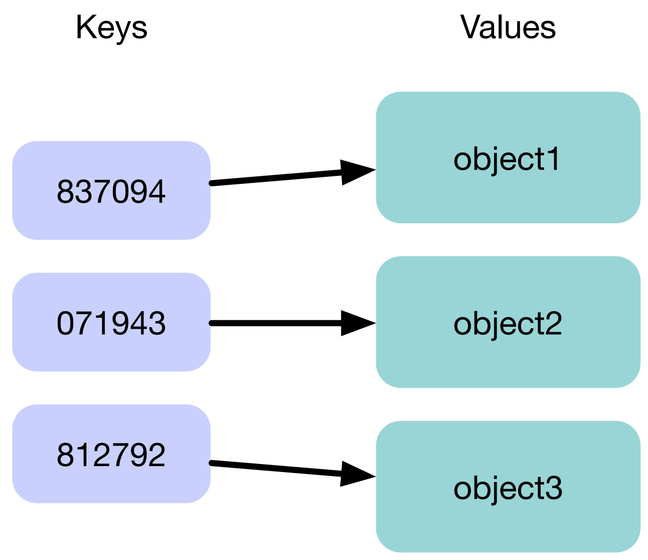
- Databases already use caching out-of-the box on different layers (datafile caching, log caching, table caching, ...) to increase performance.
- In some circumstances it can still be beneficial for a web application to cache the results of database queries.
  - For example:
    - Lots of read requests
    - Read requests require complex queries with lots of joins

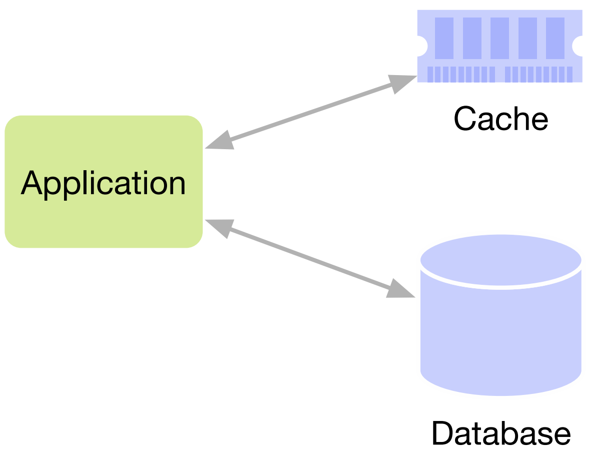
```
select users.user_id,
       users.email,
       count(*) as how_many,
       max(postings.posted) as how_recent
from users, postings
where users.user_id = postings.user_id
group by users.user_id, users.email
order by how_recent desc, how_many desc;
```

In-memory object caches

Introduction

- An **in-memory object cache** provides a hash table for storing objects in memory.
  - "Object" means arbitrary data.
  - Objects are accessed with a **key**, which can be arbitrary data as well.
  - Usual cache behavior:
    - When the table is full, subsequent writes cause older data to be purged in Least Recently Used (LRU) order.
    - The data is not persisted to disk.
- Object caches are often deployed on a separate server and accessed over the network.
- To increase capacity some object caches can be sharded, i.e. distributed over several servers.
- Examples:
  - memcached
  - Redis





Database query result caching

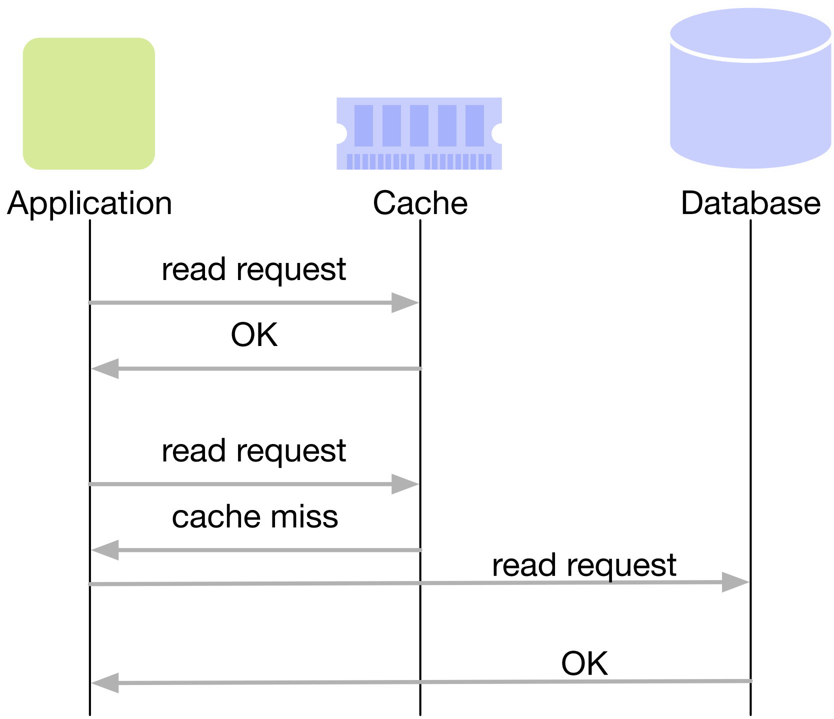
Constructing cache entries

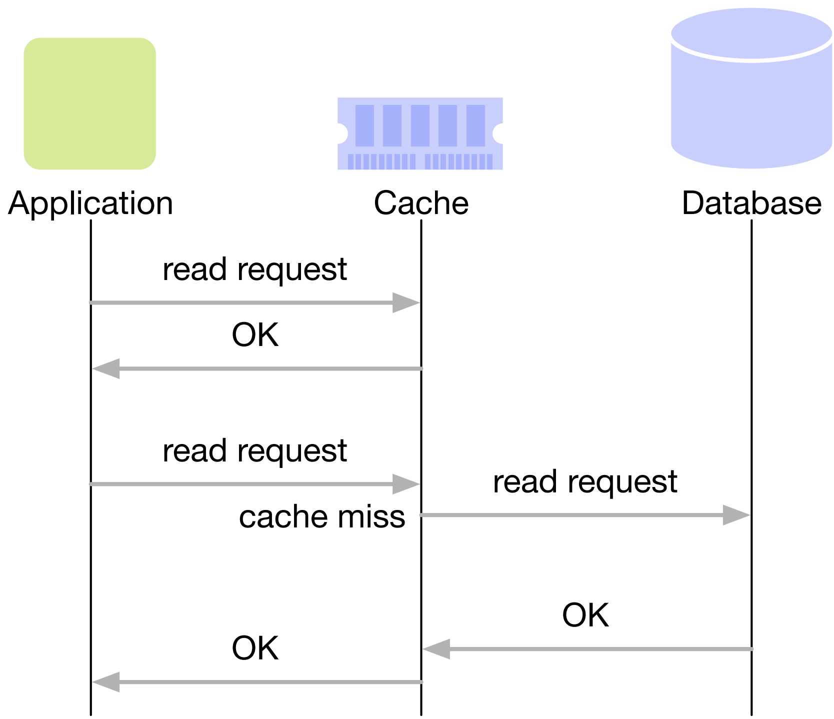
- The application has to put the database query results into the cache in the form of a key-value pair.
  - The value is the result of the query. But what is the key?
  - Developer identifies the SQL templates used to perform the queries. For example
    - Q1: SELECT qty FROM inv WHERE name = ?
    - Q2: SELECT name FROM inv WHERE entry.date > ?
    - Q3: SELECT * FROM inv WHERE qty < ?
  - The key is a composite formed by combining
    - a template identifier
    - the template parameters used for the query
  - Example keys:
    - `"Q1|chair"`, `"Q1|cabinet"`
    - `"Q2|2014-11-19"`
    - `"Q3|5"`

Caching

Read architectures

- A cache can be deployed in two different **read architectures**
  - **Look-aside**: Application interacts with both cache and database
  - **Look-through**: Application interacts only with cache. Cache interacts with database.

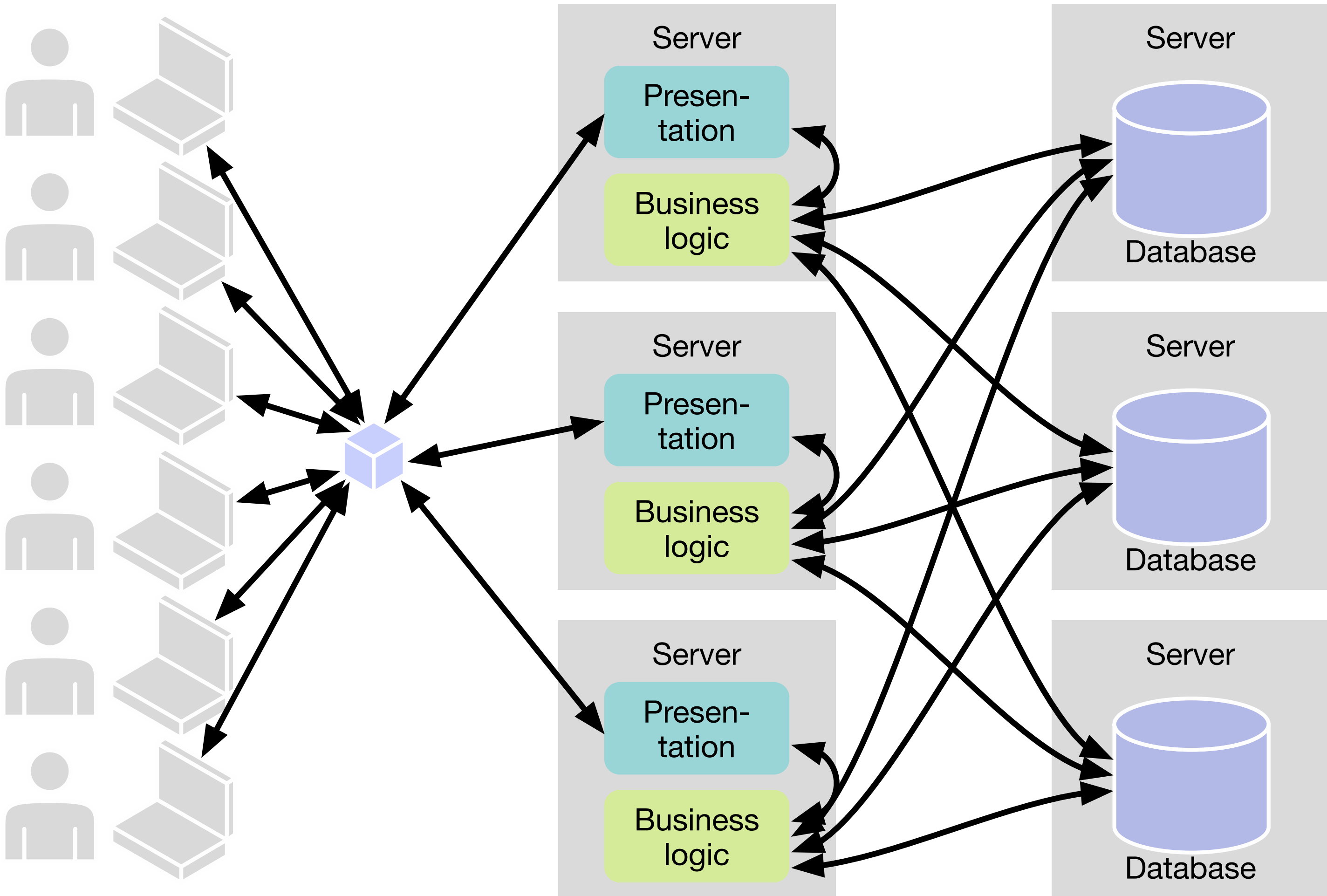




Database query result caching
Case study: Facebook

- Facebook requirements:
  - Near real-time communication
  - Scale to process millions of user requests per second
  - Two orders of magnitude more reads than writes
- Solution: use memcached for database query result caching

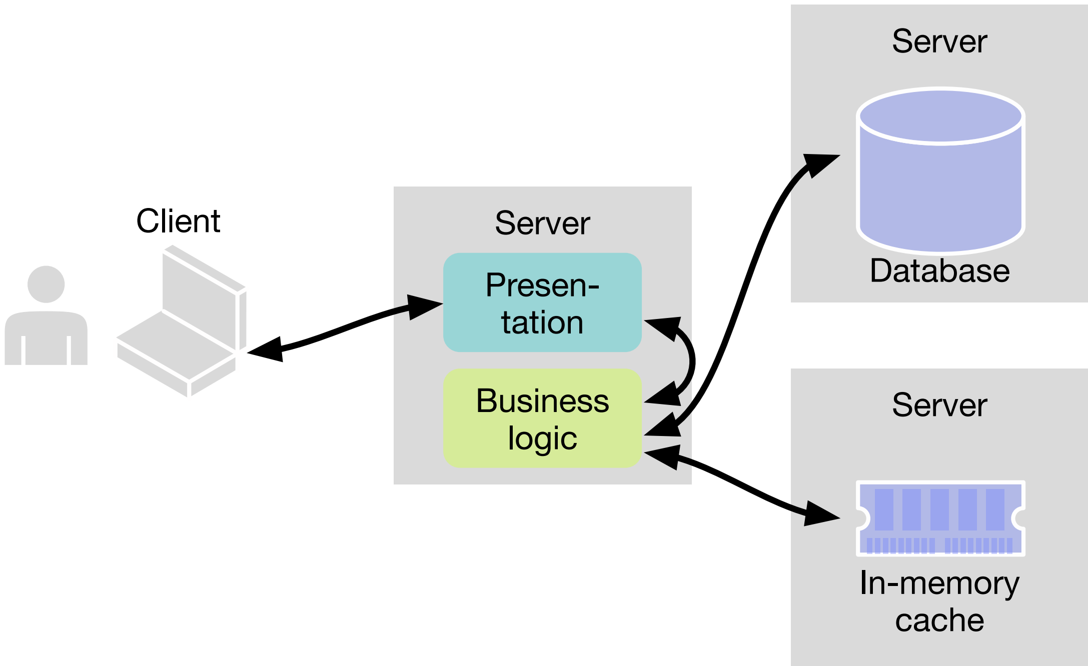
Architecture before memcached



Source: R. Nishtala et al. — Scaling Memcache at Facebook — Proc. NSDI 2013

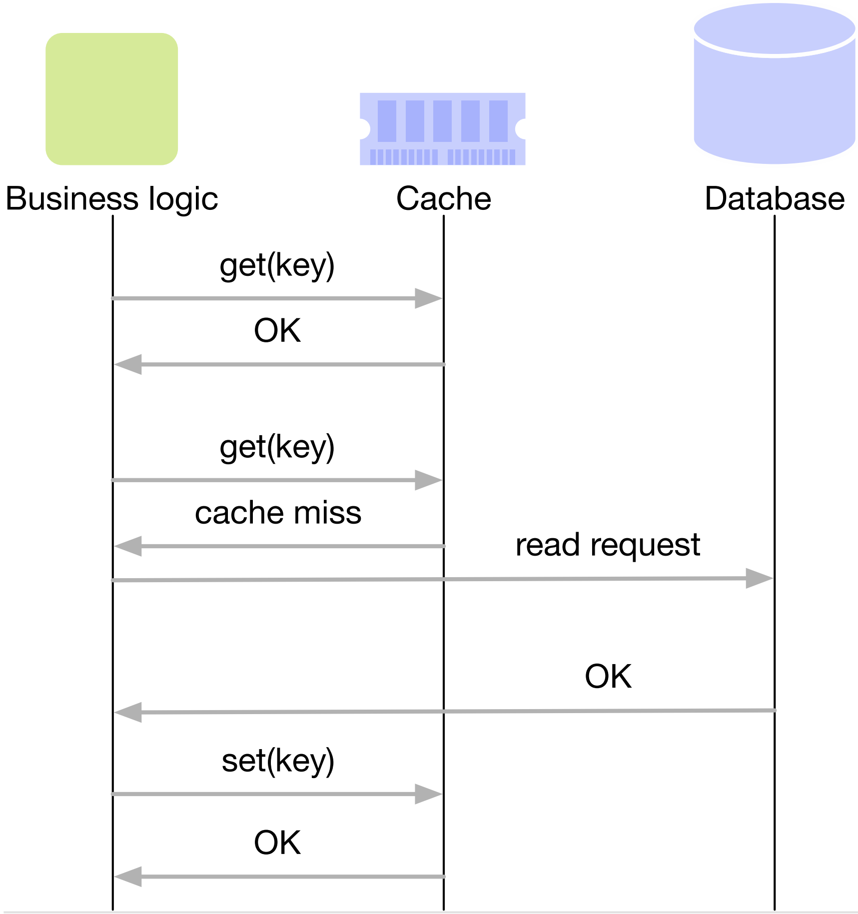
Database query result caching
Case study: Facebook

- Use memcached for database query result caching
  - Deployed as **demand-filled look-aside** cache

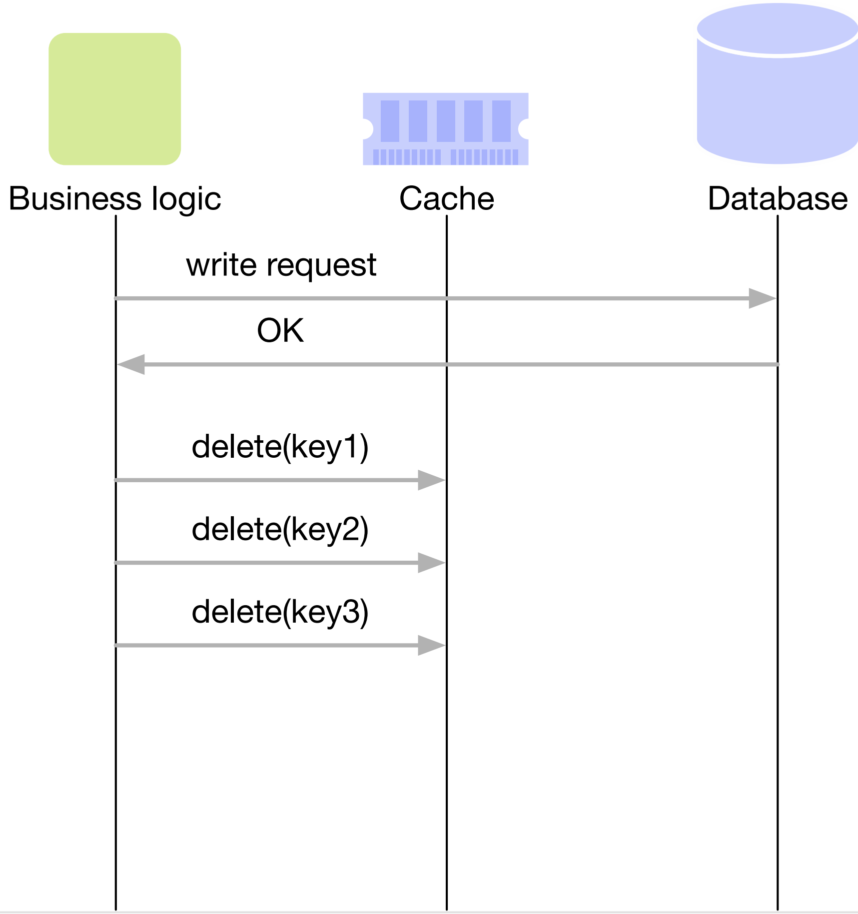


Database query result caching
Case study: Facebook

- The cache is demand-filled: The application creates a cache entry when it wants to make a read but encounters a cache miss.



- Cache invalidation: The application invalidates a cache entry after a write to the database.



Database query result caching

Case study: Facebook

- "Thundering herd" problem: An item that is needed in many user requests is not in the cache
  - Many business logic instances request the data at nearly the same time.
  - They don't find it in the cache, so they all make a request to the database.
  - The database gets overloaded by many identical requests. It would have been sufficient that one instance makes the request and puts the result in the cache.
- Solution: Cache hands out leases to the instances.
  - On a cache miss the cache gives a lease to the instance for that key. The instance makes a read request to the database and uses the lease to create the cache entry.
  - Other instances don't get a lease. They wait a bit and then try to read the cache again.