DEPARTMENT OF
COMPUTER SCIENCE

FRANCISCO RAMOS DE DEUS MENDES

BSc in Computer Science and Engineering

CONSISTENT AND EFFICIENT APPLICATION CACHING

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
October, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CONSISTENT AND EFFICIENT APPLICATION CACHING

FRANCISCO RAMOS DE DEUS MENDES

BSc in Computer Science and Engineering

Adviser: Nuno Manuel Ribeiro Preguiça
Full Professor, NOVA School of Science and Technology

Co-adviser: João Carlos Antunes Leitão
Associate Professor, NOVA School of Science and Technology

Examination Committee

Chair: Carla Maria Gonçalves Ferreira
Associate Professor, NOVA School of Science and Technology

Rapporteur: Ricardo Vilaça
Researcher, INESC TEC

Adviser: Nuno Manuel Ribeiro Preguiça
Full Professor, NOVA School of Science and Technology

Co-adviser: João Carlos Antunes Leitão
Associate Professor, NOVA School of Science and Technology

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
October, 2023

Consistent and Efficient Application Caching

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.2) [20].

To my partner in crime, Margarida, and my family.

Acknowledgements

First and foremost, I am greatly thankful to my supervisors, Professor Nuno Preguiça and Professor João Leitão, for their guidance, patience, and expertise throughout the entirety of this dissertation. Their mentorship was instrumental in shaping the quality and direction of this work.

My sincere appreciation to NOVA School of Science and Technology and to NOVA University Lisbon for providing the necessary resources, facilities, and financial support that made this research possible. I am thankful for NOVA Lincs, which funded my participation in the INForum conference.

I am grateful to have had the support of my family and friends. Their encouragement, understanding, and patience were key throughout this academic journey.

Finally, I have been fortunate to share my academic journey with my partner in crime, Margarida. Her love, support, motivation and, especially, patience were essential throughout the ups and downs of our academic journey.

This dissertation represents the culmination of years of hard work, dedication, and collaboration, and I am grateful to each and every one of you who played a role in its completion.

Thank you.

Abstract

Application caching has become extremely popular and vital to today's Web applications and services, becoming a core component of these systems. Application caching consists of integrating a caching layer into the existing application and storing data and computation results from the database. It significantly improves performance by producing faster responses and reducing overall system load.

However, while this level of caching provides better latency and scalability for the applications and services that use it, there are some complex challenges to consider. One of the primary challenges is to guarantee that the data in the cache remains consistent with the data stored in the database when concurrent requests are issued to the application.

The integration of mechanisms in the application to ensure this consistency can be complex and is often overlooked, leading to the return of stale data or the unnecessary invalidation of still up-to-date data from the cache. If this issue remains unresolved, it leads to consistency or performance problems for the application, ultimately defeating the purpose of using an application cache.

In this dissertation, we propose a system named ClearCache, which addresses this consistency problem. It was designed as an integrated library that can be used in applications that make use of MongoDB as their database, and it includes the logic required to write to and access data from a Redis cache instance automatically. Through the use of timestamps and Redis functionalities, it transparently manages the consistency between the database and cache layers.

Lastly, we present an experimental evaluation, where the results show that ClearCache enables better performance for applications that use it, reducing latency and increasing performance. In addition to its efficiency, it is also simple to use.

Keywords: Application caching, Consistency, Concurrency, Invalidation, Transparent caching

Resumo

O caching aplicacional tornou-se extremamente popular e vital para as aplicações e serviços Web atuais, tornando-se um componente central destes sistemas. O caching aplicacional consiste na integração de uma camada de cache na aplicação existente e na sua utilização para armazenar dados e o resultado de cálculos provenientes da base de dados. Proporciona melhorias significativas de desempenho ao produzir respostas mais rápidas e ao reduzir a carga global do sistema.

No entanto, embora este nível de caching proporcione uma melhor latência e escalabilidade para as aplicações e serviços que o utilizam, existem alguns desafios complexos a considerar. Um dos principais desafios é garantir que os dados presentes na cache permanecem consistentes com os dados armazenados na base de dados quando são efetuados pedidos concorrentes à aplicação Web.

A integração de mecanismos na aplicação para garantir esta consistência pode ser complexa e é frequentemente negligenciada, levando ao retorno de dados obsoletos ou à invalidação desnecessária de dados ainda corretos da cache. Se esta questão não for resolvida, irá resultar em problemas de consistência ou de desempenho para a aplicação, acabando por anular o objetivo da utilização de uma cache de aplicação.

Nesta dissertação, propomos um sistema designado ClearCache, que aborda este problema de consistência. Foi implementado como uma biblioteca integrada que pode ser usada em aplicações que utilizam o MongoDB como base de dados, e inclui a lógica necessária para escrever e aceder automaticamente a dados de uma instância de cache Redis. Através do uso de *timestamps* e funcionalidades do Redis, o sistema gere de forma transparente a consistência entre as camadas de banco de dados e cache.

Por fim, apresentamos uma avaliação experimental, onde os resultados mostram que o ClearCache permite um melhor desempenho das aplicações que o utilizam, reduzindo a latência e aumentando a performance. Para além da sua eficiência, é também simples de utlizar.

Palavras-chave: Cache aplicacional, Consistência, Concorrência, Invalidação, Cache transparente

Contents

List of Figures

List of Tables

List of Listings

Glossary

Cumulative Distribution Function Cumulative Distribution Function is a concept used in probability theory and statistics to describe the probability distribution of a random variable. The CDF provides a way to understand the likelihood of a random variable taking on a value less than or equal to a specific value. *(pp. xiv, 64)*

RESTful Applications with interfaces that adopt and follow the REST architecture constraints, enabling them to be scalable and faster. *(p. 1)*

Acronyms

API Application Programming Interface *(pp. 1, 2, 5, 34, 37)*

CDF Cumulative Distribution Function *(pp. 64, 65)*
CDN Content Delivery Network *(p. 5)*

DNS Domain Name System *(p. 5)*

IOPS Input/Output Operations per Second *(p. 6)*

P-S-S Primary with Two Secondary Members *(pp. 28, 29, 56, 62, 65)*

RDBMS Relational Database Management System *(pp. 23, 24)*
RTT Round-Trip Time *(pp. 9–11)*

TTL Time to Live *(pp. 9, 13, 14, 19, 23, 48, 70)*

YCSB Yahoo! Cloud Serving Benchmark *(p. 53)*

1 Introduction

In this chapter, we start by providing context for where (Section 1.1) and how (Section 1.2) the problem to be addressed occurs and how we plan to solve it (Section 1.3). Finally, the remaining document structure is presented (Section 1.5).

1.1 Context

Today, most Web applications and services are based on having application servers and RESTful services, such as Application Programming Interface (API)s, that must be extremely fast to return the expected data to the user. These applications tend to have extensive data sets due to the extremely personalised data based on each user's experience. This data is kept in databases, making these applications database-driven.

To ensure that a Web application is efficient, many developers and engineers put in a lot of work to get the most performance out of their databases. However, the performance obtained with a disk-based database is limited. For instance, the mechanics of retrieving data from the disk determine a significant percentage of the latency of a database query, often causing the database to be a bottleneck for these applications. Slow performance can compromise the goals of any application or service, which is why finding smart mechanisms to improve it is critical.

Application caching consists of integrating a caching layer into the existing application to improve performance. Caching solutions, like Redis [33] and Memcached [21], have thus begun to be used in these kinds of applications and services to cache large amounts of data. Caches, as opposed to databases that write to disk, can be more efficient because they are implemented in a machine's local memory. Additionally, as durability is not an issue, data does not need to be replicated as in databases, which minimises the overhead when accessing the cache.

This way, the application server does not need to query the database every time it needs data for a user, reducing the database load while still delivering the most optimal and cost-effective response. Furthermore, the cache can store the result of complex queries or data resulting from some computation over data fetched from the database.

Frequent access to the database can be costly for applications deployed on cloud platforms because of their pay-per-operation model.

1.2 Motivation

Using a caching layer in modern high-traffic applications is essential, but caches also have some hard decisions that must be properly addressed to take full advantage of their benefits.

One of the primary challenges is to guarantee that the data in the cache remains consistent with the data stored in the database. To this end, a procedure must be established that correctly determines when to add new data or discard cached data once it no longer represents the data in the database. This is a very complex synchronisation problem to solve, as most applications either ignore the problem, expose an inconsistent state, or unnecessarily invalidate large amounts of the cache, which leads to performance issues.

The problem arises from the concurrent nature of these applications, with a large number of read and write requests being issued at the same time to the database and cache.

In addition to defining *when* the selected content will be inserted into and removed from the cache component, there are other similar important considerations when developing such systems: choosing *what* content to cache, determining *where* to store cache content efficiently, and deciding *how* to properly implement the caching logic (Mertz and Nunes [23]).

1.3 Expected Contributions

The main objective of this dissertation is to simplify the correct and efficient use of application caches in web applications and services. To this end, we present a system design that integrates the database and the cache. Applications will use a single interface to access data, with the system transparently accessing and modifying the database or the cache to guarantee that applications access a consistent view of the data.

1.4 Contributions

The prime contributions of the developed work are:

- Development of a library that can be integrated with Java applications that use MongoDB as their database of choice. By matching the API of the official driver to connect the application to the MongoDB database, it allows ClearCache to be transparent to the client application and introduces the necessary logic to use a Redis cache.

- Implementation of a solution based on the document's ID and MongoDB timestamps to order operations and the documents involved in them so that the consistency of data stored in the cache is ensured, even in the presence of concurrent requests.
- Implementation and application of Redis Functions, a recently introduced technique in Redis that allows for efficient and consistent management of cached data.
- Implementation to support replica set database deployments through the concerns specified by the clients to control consistency and availability guarantees.
- Implementation to support transactions and the simultaneous use of the cache while maintaining the isolation and consistency guarantees set by the database.
- Evaluation of the results produced by ClearCache against a set of workloads in different database deployment configurations, along with a comparison between an application with and without our system.
- Publication of research communication for the 2023 edition of INForum conference entitled "Cache Aplicacional Consistente e Eficiente". This communication presents some of the work developed in this dissertation, namely certain implementation details that maintain and manage consistency, as well as some of the results obtained in the system evaluation.

1.5 Document Structure

The remainder of the document is organised as follows:

- Chapter 2: In this chapter, we provide some background on how and where caches are most commonly used today.
- Chapter 3: In this chapter, we introduce related work. It covers the core concepts of caching, some popular caching services, and systems that address the data consistency issue of application caching.
- Chapter 4: In this chapter, we present a design for the system solution proposal, its architecture, and the tools to be used for the implementation.
- Chapter 5: In this chapter, we describe the implementation aspects of the ClearCache and the various decisions to consider throughout its development.
- Chapter 6: In this chapter, we discuss the results of the evaluation of the system.
- Chapter 7: In this chapter, we conclude the dissertation with an overview of the work done and discuss some possible future work.

2

Background

In this chapter, we present caching, including what a cache is and its role in computing systems (Section 2.1). We delve into cache hierarchies, the levels of caching (Section 2.2), and how caches work (Section 2.3). Finally, we also discuss the importance of caches in computing systems (Section 2.4).

2.1 What is a Cache?

A cache is a high-speed data storage component that stores portions of a dataset intending that future requests referring to the cached data can be served faster than accessing the underlying slower storage layer. Using a cache allows for reusing recently or frequently accessed or computed data, thus improving the system's overall performance.

2.2 Levels of Caching

The concept of caching is present at various levels, almost becoming ubiquitous in computer technology. Whether used within a local machine or by Web applications, all aim to enhance performance. Below, we present some levels where a caching mechanism is utilised.

CPU: Sitting between the registers of the CPU and the main memory (RAM) are three levels of cache memory. Each one is smaller, faster and more costly per byte than the next: L1, L2 and L3. Instructions and data are managed between the three so that the execution of the processor is more efficient.

Operating Systems: Similarly to the CPU, operating systems use a technique called page cache, or disk cache, to load files from the slower storage device, like a hard disk drive (HDD) or solid-state drive (SSD), into the random-access memory (RAM).

Browser Caching: When a user loads a web page for the first time, static resources are cached into the user's machine. These resources can include text, media files, CSS

and JavaScript files. This way, reusing them faster for a web page that uses the same resources and helps reduce network traffic and load on the origin server is possible.

Server Caching: Technologies like Content Delivery Network (CDN) services rely heavily on caching to maintain a copy of the static content from a web application or service geographically closer to its users. Similarly to browser caching, the content becomes faster to retrieve as the proxy servers are optimised for accelerated content distribution while reducing network traffic and load on the origin server.

Database Caching: Databases usually have an internal cache to avoid repeatedly querying a database. The most popular approach uses a hash table that stores key-value pairs.

Application Caching: Application caching is typically used in database-driven applications. Storing frequently accessed data or heavy computations from the database into local memory produces faster response times and less overhead on the primary database.

DNS: Domain Name System (DNS) cache servers can be used to convert domain names to the associated IP addresses.

Session Management: HTTP sessions contain data such as login information, user preferences, and their histories. All those sessions can be stored in a scalable and robust manner using cache servers to provide a richer experience to the users.

In this dissertation, and from now on, we will focus on application caching.

2.3 How does a Cache function?

Typically, a cache is implemented as a key-value store, where the `key` is a unique identifier, and the `value` is the cached data. The provided API usually consists of the methods `PUT(key, value)` and `GET(key)`, which returns the associated value. Key-value stores are popular due to their speed, efficiency, and simplicity. This is possible by using memory instead of disk and excluding complex queries and aggregation logic. Many specifications can be optimised to suit the cache best for the respective workload and type of data it will work with, as discussed in Section 3.1.

2.3.1 In-memory Cache

An in-memory cache is a type of cache where the data is stored directly in the RAM of a single machine/server, which is faster than the original disk-based database.

In turn, one can be local or remote. If local, it means that the cache is only accessible by the web application running on the same server/machine, so the server's RAM is shared between both. Consequently, the application has faster access to the cache, but the size of

the cache is also limited to the available memory of that machine. If remote, the cache has all RAM available, but extra latency is also due to the required network communications.

If we need to scale and add more application instances, each machine will have its own independent cache, which may result in the same key being associated with different versions or in the same data being duplicated in the cache of multiple machines.

This potential divergence in cached data versions across machines introduces a risk of inconsistency and data integrity. In addition, the duplication of data in the caches of multiple machines not only wastes memory resources but also poses a threat to efficient resource utilization.

2.3.2 Distributed Cache

Distributed caches became popular as a way to improve some aspects of in-memory caches, such as data loss in server restarts or failures, a more available and scalable cache, and the possibility to share the cache between distinct application server instances in the same network.

A distributed cache can grow beyond the memory limits of a single computer by linking together multiple computers, creating a distributed architecture, and using the combined memory for a larger cache capacity and increased processing power. They are especially useful in environments with high data volume and load.

This architecture allows replication or sharding of data between the cache servers, plus incremental scaling by adding more computers to the cluster, allowing the cache to grow in step with the data growth. This type of cache is not as fast as an in-memory cache because it is heavily dependent on the latency of the network communications.

2.4 Why do we need to use a Cache?

When developing an application that needs to provide low latency responses, relying only on a disk-based database (HDD or SSD) to achieve this goal can be challenging, either because queries can be slow to process or simply the high cost to scale. Employing a cache can result in better performance and highly available and scalable applications.

Through the use of caching, the performance of the application improves since memory is orders of magnitude faster than disk, resulting in sub-millisecond data accesses for in-memory caches. As a consequence, read throughput and, therefore, Input/Output Operations per Second (IOPS) increases while the application benefits from the cache's ability to serve more read requests.

High throughput also leads to a more predictable application performance under times when the volumes of internet traffic are higher. With a lower number of read requests going to the backend database, the load on it also reduces, helping with performance under times with spikes and further preserving the hardware it runs on. Furthermore,

when a particular data is accessed more frequently than others, the use of a cache also removes database hotspots.

In specific architectures, a cache can be used to cover some primary database failures by serving content to the users.

Lastly, cloud platforms commonly use business models such as pay-as-you-use and pay-per-operation, and for applications that have their backend database deployed in the cloud, utilising a cache instance can significantly decrease the total cost to pay. This results from potentially fewer database instances required and fewer operations going to the backend database.

3

Related Work

In this chapter, we present relevant related work, considering the objective of the proposed work in this dissertation. First, we introduce some of the possible implementations and rules a cache can employ (Section 3.1). Afterwards, popular caching services are presented (Section 3.2). Then, we discuss the consistency problem that arises from employing a caching layer in an application and present systems where they attempt to resolve it (Section 3.3).

3.1 Cache implementations

3.1.1 Data Access Strategies

Data access strategies specify how to execute our application while taking into account the use of the cache component, including when to store and read data from the cache. There are multiple strategies we can opt to implement - either reactive or proactive - and the chosen strategy should meet the specific requirements and objectives of the application.

Cache-aside

In the cache-aside strategy, one of the most popular caching strategies, the cache sits next to the database, and the application handles all operations to the cache and database.

The core logic of this strategy is demonstrated with Figure 3.1 and can be described as follows:

1. When a read request is received, check in the cache if the respective data is available;
2. If the data is available (a cache hit), return the data;
3. If the data is unavailable (a cache miss), query the database for the data. Before returning the data received from the database, store it in the cache.

This strategy is reactive, meaning that the cache is only updated with some data after that data is requested by a client, also known as lazy loading.

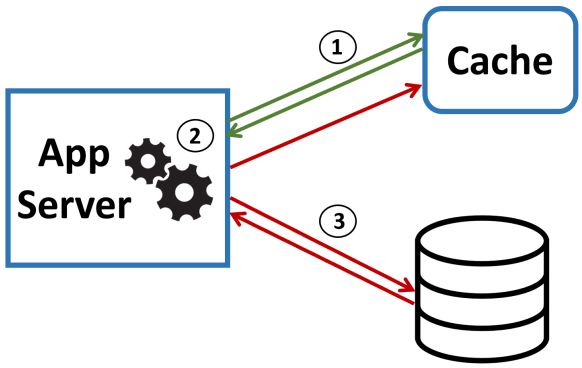


Figure 3.1: Cache-aside execution.

This strategy works best for read-heavy workloads, although it is used as an excellent general-purpose caching strategy. Furthermore, with the cache separated from the database, the system can still rely on the database to perform the clients' requests and guarantee some resiliency upon cache failures.

One drawback is that since the data is only inserted into the cache once it is requested, there is a consequence of an initial overhead in the event of a cache miss. The responses on these events can be considerably slower due to the extra Round-Trip Time (RTT)s required to complete the request.

Another downside of this strategy originates from having write requests executed directly to the database, which allows for possible inconsistencies between data in the database and cache. To deal with this, developers may use techniques such as Time to Live (TTL) to remove expired entries and make the cache refresh its data or, not to show stale data, stricter policies that invalidate cache entries depending on the executed writes.

Read-through

In a read-through strategy, the cache now sits between the application and the database. This means that the application always communicates with the cache on a read request and assigns the responsibility of communicating with the database to the cache, as shown in Figure 3.2.

1. The application sends the data read request to the cache;
2. If there is a cache hit, the data is returned;
3. If there is a cache miss, the cache reads the data from the database, stores it in the cache and then returns the data.

This strategy is also best suited for read-heavy workloads.

The main difference between read-through and cache-aside strategies is which component is responsible for fetching data from the database and inserting it into the cache. In the former case, it is the cache itself, while in the latter, it is the application.

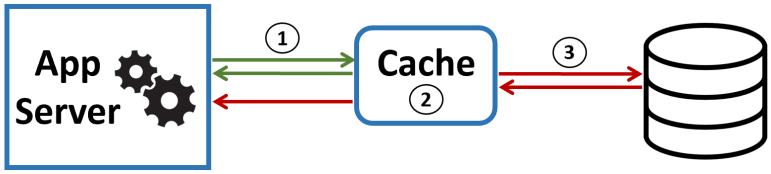


Figure 3.2: Read-through caching execution.

It is crucial to notice that when using this setup, a cache failure can produce a bottleneck in the performance of the application. In the worst of cases, this can lead to a crash of the application since it cannot access the database.

Similarly to the last strategy, read-through has the disadvantage of increased latency whenever a cache miss occurs, although it manages to be faster when compared to the same event in a cache-aside strategy.

As in cache-aside, the application executes write requests directly to the database, which produces the same potential problem of inconsistent data in the database and cache. This is a consequence of the write strategy that the application employs, as discussed in the following paragraph.

The last two strategies were read strategies that define the behaviour of the execution when the application receives a read request. Henceforth, the focus of this section will rely on discussing some of the more popular write strategies.

Write-around

Write-around is the designation for the write strategy considered when discussing both previous read strategies. The application writes the data directly to the database, and the data is only stored in the cache once read.

1. Application writes data to the database.

This strategy offers optimal latency for write operation as it only requires one RTT. Additionally, it ensures that the cache does not store data that is rarely read.

The clear downside, also previously mentioned, is that recently written data always results in a cache miss, leading to higher response latencies.

Write-through

The setup for the write-through strategy is similar to the one in the read-through, where the cache is sitting between the application and the database, and the application delegates the responsibility of writing to the cache component. The setup and execution are present in Figure 3.3

1. Application writes data to the cache;
2. Cache writes data to the database.

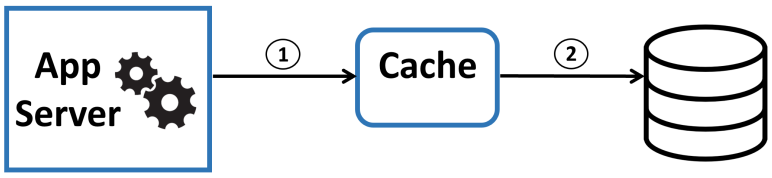


Figure 3.3: Write-through caching execution.

With this strategy, the cache is updated in the same execution as the update to the database instead of waiting for the cache miss. As a consequence, write-through is categorised as a proactive strategy, contrary to lazy loading.

This strategy introduces an extra RTT to complete the write request when compared to the previous write strategy. However, the intention behind this is that newly written data can be quickly read. On the other hand, if the most recent data is never read, this leads to the cache holding unnecessary data.

It shares with read-through the disadvantage of the cache becoming a bottleneck for the application performance in case of cache failures.

The major benefit occurs when utilising both write-through and read-through in combination. This pair ensures that the data between the database and cache always remains consistent.

Write-back or Write-behind

The behaviour of a write-back cache and a write-through cache are very much alike in the sense that the application writes to the cache, and the cache writes to the database. However, a write operation is considered complete in the write-back strategy after the data is written to the cache. The cache only writes to the database after some set interval, where it can perform a batch update on the relevant data, as seen in Figure 3.4.

1. Application writes data to the cache;
2. Asynchronously, the cache writes a batch of data to the database.

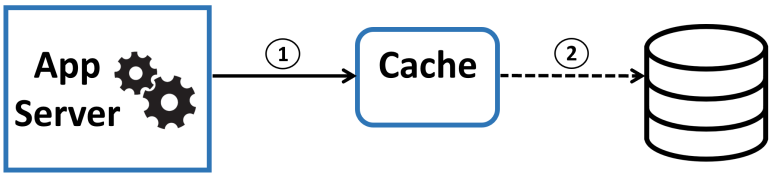


Figure 3.4: Write-back caching execution.

Since the write operation returns as soon as the application writes the data to the cache component, the time it takes the cache to write to the database is not considered anymore. As a consequence, the latency for this type of operation is significantly lower

when compared to previously discussed write strategies. With this improvement, the write-back strategy is adequate for write-heavy workloads. Moreover, when paired with read-through, it makes the combination suitable for read-heavy and write-heavy workloads.

It can also hide some failures of the database if read requests access data that is stored in the cache. In case of a cache failure, if some data exists in the cache but has not yet been written to the database, it may incur a loss of that data and thus affect the consistency of the application.

The major downside of this strategy is that it introduces problems regarding data consistency. Assuming that both reads and writes go first to the cache, between the cache and the database, there is only eventual consistency. Hence, there is a need for a mechanism that resolves any conflicting updates that may occur. Furthermore, if the database is shared among multiple applications, stale data can be served as a result of batches yet to be written.

3.1.2 Eviction Policies

As previously mentioned in Section 2.3.1, caches trade capacity for speed, and therefore, a policy that describes how to remove entries from the cache whenever it is full must be chosen.

Similarly to the data access strategies, this policy must take into account the type of data, workload and access patterns the application will have. There are several possible policies, and we present some of the more popular ones.

Random

The simplest technique to deal with this problem is randomly removing an entry from the cache.

First-In-First-Out (FIFO)

The idea behind the FIFO policy is to remove data from a cache based on the order in which it was added to the cache. The oldest entry stored in the cache is removed to make room for new data. It is simple to implement, but it may not always be the best policy for dynamic data access patterns.

Least Recently Used (LRU)

The least recently used policy requires the developer to maintain a timestamp associated with each entry. When an entry of the cache is accessed, the corresponding timestamp is updated to reflect the last time it was accessed. At a given moment, the entry to remove from the cache is the one with the lowest timestamp.

Most Recently Used (MRU)

Uses the same technique as the LRU policy but removes the entry with the highest timestamp instead.

Least Frequently Used (LFU)

Instead of using a timestamp, the least frequently used approach relies on a counter for each entry that represents the number of accesses since the entry was stored in the cache. When required, the entry that was accessed the least amount of times is removed from the cache.

Most Frequently Used (MFU)

Uses the same technique as the LFU policy but removes the entry that was most accessed instead.

Least Time To Live (LTTL)

The least time to live mechanism takes advantage of the use of TTL for each entry of the cache. TTL is a technique used to invalidate cache entries, where each entry is assigned a TTL when stored in the cache and decremented throughout the time. This policy removes the entry with the lowest TTL among all entries.

3.2 Popular Cache Services

3.2.1 Memcached

Memcached [21] is a general-purpose high-performance caching system that stores key-value pairs, regularly used to speed up dynamic web applications by alleviating database load. It is used in big companies like Facebook, Instagram, and Pinterest.

The implementation is based on a hash table and a hashing algorithm to map keys to specific memory locations. It uses an allocation mechanism called *Slab*, which segments the allocated memory into chunks of different sizes and stores entries of the corresponding size. Moreover, the stored objects are treated (mostly) as Binary Large OBjects (BLOBs), which means the binary data is unstructured, and also the value for each key is limited to a size of up to 1MB.

Memcached is easily scaled vertically as a consequence of its multithreaded nature. It is also possible to scale horizontally by adding more servers to the cache cluster, although no native support exists for that purpose.

In a distributed setting, all servers are known, and each key is assigned to a server and a hash function allows the client to use the key and communicate with the respective server, thus enabling the potential for data partitioning.

The system employs a least-recently-used (LRU) eviction policy, and an object is ranked in one of three tiers: HOT, WARM, and COLD; based on its accesses, the object is automatically moved between tiers. Additionally, Memcached has the ability to set TTL values for individual keys.

3.2.2 Redis

Redis [33] is a popular in-memory caching system and key-value store accepted in a variety of large-scale companies such as Twitter, GitHub, and StackOverflow due to its speed, multitude of functionalities, and stability. It can also be used as a database or a message broker.

It uses a hash table data structure to map keys to values, and when used as a cache, it enables much faster data retrieval by not querying the persistent storage every time a data request is made.

It supports a wide variety of data types and respective operations for each type. The data types include strings, lists, sets, sorted sets, hashes (similar to maps or dictionaries), bit arrays, and hyperloglogs (a probabilistic data structure that estimates the cardinality of a set).

Redis is mainly single-threaded and typically used as a remote in-memory cache. Additionally, it can be made distributed since it was designed to be horizontally scalable, and it provides several built-in features to make scaling out easier.

One of the most essential features that make Redis easy to scale horizontally is Redis Cluster. Redis Cluster is a built-in sharding solution that allows data to be split across multiple servers/machines using range and hash partitioning strategies, giving the cache permission to scale out as the data set grows. Redis Cluster automatically handles data partitioning, replication, and automatic failover, making adding or removing nodes from the cluster simple.

Another feature is Redis Sentinel, a high-availability solution that allows Redis to automatically failover to a slave node in the event of a master node failure.

It uses a random or approximated LRU or LFU algorithm for cache eviction, with the possibility to control it by making some keys persistent and associating a TTL to the volatile keys. Transactions are also supported to guarantee atomicity and that the commands are serialised and executed sequentially.

3.2.3 Memcached vs Redis

Memcached and Redis are both in-memory caching systems that are commonly used to improve the performance of web applications. However, there are some key differences between the two:

- Data Types: Memcached only supports simple key-value pairs of unstructured binary data, whereas Redis supports a variety of data structures. As a result, it can

provide more advanced functionalities, for example, leaderboards or a list of recent topics.

- Eviction Policy: Redis offers 8 possible alternatives, while Memcached only offers a LRU policy.
- Scaling: Redis has built-in support for scaling, while Memcached requires additional tools or libraries to achieve this.
- Data Persistence and Replication: Memcached is a volatile cache, which means that it stores data only in memory, and all data is lost when the server is rebooted or goes down. Redis, on the other hand, has the option to persist data to disk or replicate across nodes for data durability and high availability in case of failures.
- Lua Scripting: Redis supports Lua scripting, which allows to perform complex operations on the stored data using scripts, while Memcached does not have this feature.
- Speed: Redis is generally considered to be faster than Memcached. However, in some cases where the data set is large enough, Memcached can be better because of its simplicity.

Overall, Redis is more feature-rich than Memcached and is well-suited for applications that require data persistence, complex data structures, and support for Lua scripting. While Memcached is more simple and focused on caching performance, it is a good fit for systems that do not require all those features and need a simple, easy-to-use caching solution.

3.2.4 Integrated solutions

Today, cloud platforms have begun to provide a caching layer already integrated with the offered database services. With these solutions, the cache is made transparent, and the developer is not responsible for the data put in the cache as well as the implementation details of the caching layer. Some possible architecture alternatives are introduced below:

Cosmos DB integrated cache

Azure Cosmos DB [7] is a globally distributed, multi-model database service from Microsoft that supports document, key-value, graph, and column-family data models. It has a feature called "Integrated cache" that allows for fast access to data by caching frequently accessed data in memory.

The Integrated cache in Azure Cosmos DB is a read-through cache that automatically caches data when it is read from the underlying storage. The cache is also invalidated automatically when the data is updated or deleted, ensuring that the data in the cache is always up-to-date.

The integrated cache feature [8] is available for all data models supported by Azure Cosmos DB, and it can be enabled or disabled on a per-collection basis. The size of the cache can also be configured, allowing more control over the amount of memory used by the cache.

However, the integrated cache only supports read requests with a session (the most commonly used consistency level for both single-region and globally distributed Azure Cosmos DB accounts) and eventual consistency. If a read has a consistent prefix, bounded staleness, or strong consistency, it will always bypass the integrated cache and be served from the backend.

Amazon DynamoDB Accelerator

Amazon DynamoDB Accelerator [4] (DAX) is a highly scalable and fast NoSQL cache service provided by Amazon Web Services [3] (AWS) that is designed to enhance the performance of read-intensive workloads. DAX operates as an in-memory cache for DynamoDB [6] tables, providing fast and predictable low-latency access to data. This can significantly reduce the response time for read-intensive workloads from milliseconds to microseconds.

One of the key advantages of DAX is its ease of use and integration with existing applications. It is transparent, meaning that users do not need to change the code of their application to take advantage of its performance benefits. DAX is compatible with the DynamoDB API, so users connect their application to DAX instead of DynamoDB, and DAX will handle the rest. The cache is automatically updated when users write data to the underlying DynamoDB table, eliminating the need for manual tuning or management.

While DAX provides fast read performance, it is important to note that it only provides eventual consistency for read operations, meaning that it may take some time for the write operations to be reflected in the cache. The cache will eventually be updated with the most recent data, but until then, subsequent read operations may return stale data.

3.3 Systems and Concepts to Improve Cache Functionality

There are various tools that address some challenges that come from using a cache. Each tool has a different approach, but the common goal is to achieve more efficiency and performance, in addition to the benefits already gained from using a cache, while also reducing costs. In this section, we will present some systems and concepts that discuss and attempt to solve problems shown in Section 1.2.

3.3.1 Consistency and Cache Invalidation

We start by discussing consistency and cache invalidation, as it is the main aspect we tend to cover in the elaboration of this dissertation. This problem refers to maintaining the

data consistent between the cache and the underlying database while also managing the invalidation of stale or outdated data in the cache.

Consistency relates to ensuring that the data in the cache is the same as in the database. When concurrent requests are made to the cache and database, there is a risk of inconsistencies between the two, and to ensure this problem is not present, the corresponding data in the cache needs to be updated or invalidated when the database changes.

Additionally, this is a challenging problem to solve, as it requires a balance between performance and consistency. On the one hand, a frequently updated cache will be more consistent with the database, but it will also have a higher overhead and may have lower performance. On the other hand, a cache that is updated less frequently will have better performance but may be less consistent with the database.

Transactional caches are a type of cache that takes into account the execution of transactions. As such, different isolation levels can be considered, namely those supported in the context of SQL databases.

Levels of Isolation in SQL

The paper [9] was the first to define how the SQL isolation levels are defined in terms of phenomena. Phenomena are actions that may lead to behaviour with anomalies. The different isolation levels describe how distinct transactions interact with each other when executing concurrently in a database. The following are the isolation levels commonly used in SQL databases and the respective phenomena that they incur:

Read Uncommitted: This is the lowest isolation level and allows transactions to read data that has not yet been committed by other transactions. This can lead to *dirty reads*, where a transaction reads data that is later rolled back or undone by another transaction.

Read Committed: This isolation level ensures that a transaction can only read data that has been committed by other transactions. This prevents *dirty reads* but can still result in *non-repeatable reads*, where a transaction reads data that has been updated by another transaction.

Repeatable Read: This isolation level ensures that a transaction sees a consistent view of the data and that the same result set is returned from a query, even if other transactions are updating the data concurrently. This prevents *dirty reads* and *non-repeatable reads* but can still result in *phantom reads*, where a transaction reads data that has been inserted by another transaction.

Serializable: This is the highest isolation level and ensures that transactions are executed serially, as if they were executed one after the other, completely isolating them from each other. This prevents *dirty reads*, *non-repeatable reads*, and *phantom reads*.

Snapshot Isolation: Also introduced by [9], Berenson et al. propose an alternative isolation level, named Snapshot Isolation, which is based on a multiversion concurrency control mechanism. Snapshot isolation addresses the potential problems of the previous four known isolation levels and provides better consistency guarantees for concurrent transactions. Snapshot isolation is a type of isolation level that provides a transaction with a consistent view of the data as if the transaction started executing at a point in time when the data was consistent. This allows a transaction to access data without being affected by concurrent updates made by other transactions.

Snapshot isolation behaviour:

1. When a transaction starts executing, the database takes a snapshot of the data, capturing the state of the data at that point in time.
2. The transaction then operates on the snapshot without being affected by any concurrent updates made by other transactions.
3. When the transaction is committed, the database checks for any conflicts with updates made by other transactions since the snapshot was taken.
4. If there are no conflicts, the transaction is committed, and the changes made to the data are made permanent.
5. If there are conflicts, the transaction is rolled back, and the transaction can be repeated in a new snapshot.

Snapshot isolation is helpful for environments with high concurrency since it does not block other transactions from accessing and modifying data. However, it does not guarantee serializability, meaning that transactions may still interleave in unexpected ways and result in anomalies such as lost updates or write skew. Many database systems, primarily SQL but also NoSQL systems, provide it.

Until the end of Section 3.3, we will present some systems and techniques that, in some way, have been developed to address the issue of data consistency.

3.3.2 TxCache

TxCache [30] is one of the systems that most closely resembles what we are trying to achieve with this dissertation.

TxCache is a transactional, self-managing application-level cache. It provides a simple programming model while also providing transactional consistency.

Application developers designate functions as cacheable, and the system automatically caches their results. It is designed as a library that sits between the application and database layers, transparently making cache accesses whenever possible and updating and invalidating the cached data as the underlying database changes.

Consistency Model

For the consistency model, TxCache provides transactional consistency with the underlying database, meaning that all requests within a transaction see a consistent view of the system from a given timestamp. Moreover, an application can store the timestamp of a user's last transaction and use it as a staleness bound so that causal ordering is ensured between related transactions.

In addition to tagging the functions as cacheable, the application also needs to specify whether a transaction is read/write or read-only. If read-write, all the accesses go directly to the database, while read-only can benefit from the data in the cache given a limited staleness. This was done to avoid introducing new anomalies and rely on the same guarantees of the underlying database for read/write transactions, i.e., see the effects of their own updates, while others only after the commit.

To ensure consistent views, TxCache uses versioning to associate a validity interval to each database query, and the library tracks the queries that a cached value depends on and uses them to tag the cache entry with a validity interval.

The consistency within read-only transactions is ensured by only retrieving values from the cache and the database that were valid at the same time. The cached data is partitioned among multiple servers, and the library produces a hash to determine its location.

Versioning

In addition to its key, which is a hash of the name and arguments of a function call, each entry has a tag with its validity interval. The lower bound is the commit time of the corresponding transaction, and the upper bound is the commit time of the transaction that leads to a change in the result. It allows for multiple entries to exist with the same key, but their associated time intervals are disjoint as only one of them is valid at a given time. When the library queries the cache for a result, it sends both the key and range of acceptable timestamps.

For cache eviction, an LRU policy is employed, as well as the removal of data that is too stale to be useful (possibly through the use of TTL).

Invalidation Tags and Streams

Every object in the cache that matches the latest state of the database has a set of invalidation tags associated that describe which portions of the database it depends on. A tag consists of a table name and an optional index key description, e.g., a search for users with the name Alice would receive the tag `USERS:NAME=ALICE`. If multiple tables or keys are accessed for a query, that result receives multiple tags.

The invalidation stream is a stream of messages sent by the database to the cache, one for each update transaction, containing the timestamp and any invalidation tags affected

by the transaction. The cache processes the information of each message in order and knows what objects have their validity interval finished at the transaction's timestamp.

Upon cache lookups, items that are valid in the latest state of the database have an upper validity bound equal to the timestamp of the last invalidation received prior to the lookup. This ensures that there are no race conditions between an item being modified in the database and being invalidated in the cache and that multiple items modified by the same transaction are invalidated atomically.

Database support

Regarding the database used with the system, one of the major drawbacks of TxCache is that it needs to use a custom version of a PostgreSQL database. This was done to accommodate the mechanisms used by TxCache, such as explicit control of snapshots, automatic validity interval tracking and invalidation stream generation. They claim that any database that implements snapshot isolation has a way of keeping a history of recent database states as they did with this system.

TxCache uses additional mechanisms to support and enhance its performance, including the development of a lazy timestamp selection algorithm in the database that assigns a transaction to a timestamp in the recent past based on data availability to reduce the number of cache misses, as well as a daemon that manages the various snapshots in use at any given time.

With their experiments, they concluded that TxCache can substantially increase the performance of a web application. On the RUBiS benchmark, it increased throughput by up to 5.2x relative to a system without caching. More importantly, TxCache achieves performance comparable to a non-transactional cache, demonstrating that consistency does not have to come at the expense of performance.

3.3.3 SI-Cache

SI-Cache [29] is a high-performance multi-version cache for transactional multi-tier systems. It uses transactions to access data from the cache and the database and guarantees snapshot isolation caching in a multi-tier system. Furthermore, it promises a completely transparent cache and to not impose any constraints on the application.

SI-Cache uses *Enterprise Java Beans* (EJB) technology, which was once more popular than it is today, for its development. Most notably, the use of *Entity beans* (EBs) model business data are stored in some persistence storage, usually a database, and they are shared by all clients. An EB represents a tuple of the database, and we can think of a set of EBs as a cache of the database. It is also assumed that the application server takes care of all transactions, specifically, the reads and writes to the *beans*.

The access to EBs has the same behaviour as a cache-aside, described in Section 3.1.1. When an update of a data item x occurs in the transaction, a new private version of the corresponding *EBX* is created. When the transaction is committed, both the database and cache are updated with the new version of the *EBX*, the behaviour of a write-through cache, as described in Section 3.1.1.

By providing snapshot isolation across the database and cache tiers, SI-Cache addresses scenarios where inconsistencies may occur. These scenarios are the result of combining the locking mechanisms of the cache with the snapshot isolation of the database. As a consequence, these executions provide neither snapshot isolation nor serializability guarantees. This is why the system is called SI-Cache, where the cache adapts its concurrency control mechanism to the database.

Overview of the protocol

Start: A transaction starts in both the database and the application server, which represents the cache. The cache protocol makes sure that the correct version is read, either from the database or cache, according to snapshot isolation.

If the transaction is read-only, it simply commits. As previously mentioned, if the transaction is read/write, a new EB is created for the variable, and possible conflicts with concurrent transactions are detected at execution time. The updates are written to the database at commit time, and both sides commit.

In the application server, each transaction T has an associated *start timestamp ST(T)* and a *commit timestamp CT(T)*. Each data item bean $EBX_C T(T)$ has a tag with the *CT(T)* of the transaction T that updated and committed this version.

Read: For the data reads of a data item x, a transaction T reads its own version if it has been written to it before. Otherwise, it reads from the cache the last committed version as of the time it starts. If no bean for that item is cached, a database read is issued, and the bean *EBX* is tagged with -1 since the version is unknown.

The system also has a garbage collection mechanism since the cache can have multiple versions of the EB for a data item, all with different commit times, to support the reads of older concurrent transactions that have not yet been terminated.

Write: To detect conflicts between concurrent writes to the same data item, SI-Cache uses locking, where the requests follow a wait queue to access, and version checking to determine if a transaction can continue or abort.

Commit: In case of a write transaction, the private version is tagged with a new commit timestamp and added to the cache. The transactions of the application server and database commit and starting transactions can begin using the new and correct snapshot.

Abort: In this case, the private bean version is discarded, and the locks are released.

To deal with the creation of EBs, the system locks the EB to prevent other concurrent transactions from creating data items with the same primary key. For the deletion of EBs, a tombstone is created to represent the deletion of a data item, and upon committing, the tombstone version is made public to all subsequent snapshots.

In the paper, Perez-Sorrosal et al. presents the pseudocode for the operation of the system as well as a correctness proof to show that both atomicity and snapshot isolation are provided across both layers.

Elastic SI-Cache

Elastic SI-Cache is an extension of the original system that works by replicating both the application server and database tiers to provide scalability and availability.

By coupling an application server instance with a database instance to be a replica and replicating it as a unit, it avoids non-replicated layers being a single point of failures and performance bottlenecks. Each application server talks with its database and with all other application servers. Each application server has its own cache.

The difference in the execution is at commit time, where the replication protocol, which is embedded in the application server cache, multicasts all EB versions created by a transaction. The replication protocol performs a final validation at each replica to compare with existing timestamps, and if successful, the transaction commits in both layers.

For this system, proof of correctness is also presented to show that the system provides snapshot isolation at the global level.

Comparison

A comparison to TxCache (Section 3.3.2) is made in the related work section of the paper, and SI-Cache does not require modification of the database and yet is able to keep the consistency between the versions from the cache and from the database. Therefore, any database system that provides snapshot isolation can be used.

It is also mentioned that SI-Cache always provides the latest snapshot when a transaction starts, in contrast to TxCache, where an older snapshot might be given to a transaction based on the available cached results.

Lastly, TxCache does not address elasticity and scalability, whereas SI-Cache makes the ability to replicate the base system one of its priorities.

3.3.4 Gumball

Gumball Technique (GT) [16] prevents race conditions for cache-augmented SQL database management systems. During the normal execution of a cache (database update, the corresponding cache invalidation, and following cache miss on subsequent requests), race conditions might occur and result in cache states that represent stale data. Gumball tries to detect these race conditions and prevent any key-value pair from being inconsistent with the database.

More specifically, the problem that this technique addresses arises when deleting operations from the system starts racing with reads that have observed a cache miss and tries to set the cache with the old values, resulting in stale cached data. The key idea is to allow some cache put operations to be ignored to prevent data inconsistency and to redirect requests to use the database, not to return stale cache data. This technique works with all off-the-shelf Relational Database Management System (RDBMS) and existent systems that already enforce consistency on caches.

For example, an undesirable race condition occurs when a code segment that treats a cache miss by a user request, CS_{fuse}, and the code segment that treats the invalidation of a cache entry when the database is updated, CS_{mod}, are executing in parallel. And even though both CS_{mod} and CS_{fuse} employ the concept of transactions, their cache and database operations are non-transactional and may leave the cache inconsistent.

With their experiments, the paper shows that GT reduces inconsistencies dramatically while adding negligible overhead due to the extra requests redirected to the database and extra space occupied in the cache by the gumballs.

Technique

GT is implemented within the key-value store by extending the simple provided operations of put, get and delete to manage the gumballs.

For the delete method of k_i, if an entry k_i-v_i exists, then delete and generate a gumball, k_i-g_i, with T_{g_i} set to the current time. If an entry k_i-g_i exists, then update the T_{g_i} to the current time. Finally, if no entry exists, then generate a gumball, k_i-g_i, with T_{g_i} set to the current time. Important to notice that every time a gumball is generated, GT assigns it a fixed TTL, which they call Δ, to prevent it from occupying the cache memory longer than necessary.

For the get method of k_i, if an entry k_i-v_i exists, then return it. If an entry k_i-g_i exists or no entry exists, then a cache miss is reported with the current time as T_{miss}. After the cache miss, and after CS_{fuse} has calculated the value for the corresponding k_i, the T_{miss} previously provided by GT is included as an argument to the put method.

For the put method, the executing cache server compares T_{miss} with the current timestamp, T_C, and if $T_C - T_{miss} > \Delta$, then the put operation is ignored, as a gumball may have existed and it is no longer in the cache because it timed out. If the condition is not true, one of three scenarios can happen.

First, if an entry k_i-g_i exists and $T_{miss} < T_{g_i}$, then ignore the put operation. Otherwise, it succeeds, and T_{miss} is kept as metadata for this entry.

Second, if an entry k_i-v_i exists and its metadata timestamp is after T_{miss}, the put operation is ignored. Otherwise, the put succeeds overwrites the existing value and updates the timestamp on the metadata.

Third, if no entry exists for k_i, then k_i-v_i is inserted and its metadata timestamp is set to T_{miss}.

Note that GT rejects all race conditions, even those that have an execution that does not produce inconsistencies, and as a result, some requests will be redirected to the database without it being needed.

Defining Δ

In the paper, the authors also mentioned how to define the value of Δ, both statically and dynamically. In a static way, it is mentioned that Δ should be the time passed between CS_{fuse} observing a cache miss for key k_i until the moment where a put(k_i, v_i, T_{miss}) is issued. The dynamic computation approach is based on estimating the CS_{fuse} response time, RT, which is obtained by the formula: $RT = T_C - T_{miss}$. Every time Δ is updated a timestamp, T_{adjust} is kept, so that put operations that have a $T_{miss} < T_{adjust}$ are also ignored.

Comparison

In the related work section of the paper, a comparison with the TxCache system introduced in Section 3.3.2 is made. Although they both use timestamps to detect race conditions, the semantics are different and obtained at different times of the execution.

TxCache is a consistency technique that implements transactions with snapshot isolation, while GT is a building block of a consistency technique. This means that GT can choose to abandon the extra properties and overhead that come with the transactions, unlike TxCache. Finally, GT does not require changes or specific features from the RDBMS.

3.3.5 Scaling Memcache at Facebook

The scientific paper "Scaling memcache at Facebook" [27] is widely considered a reference in the field of caching systems as it tackles the problems and respective solutions of scaling such systems.

The paper describes how Facebook uses its memcache service as a cache and how it was scaled to handle the large and rapidly growing amount of data and traffic on the site.

The paper considers three scenarios and a set of problems and solutions for each: within a single cluster, within a region (that may have many clusters), and between many regions (where each has many clusters).

In a cluster

The main objectives at this level were reducing latency and reducing load. To reduce the load, the authors introduce a mechanism called leases that addresses the problem of concurrent writes, specifically stale sets and thundering herds. Leases are a way to ensure consistency in a distributed caching system when multiple clients are trying to update the same data item simultaneously.

Leases are values distributed to clients for a given key. To deal with stale sets, i.e., when a client sets a value with an old value, the backend server examines the most recent

lease for a given key and blocks writes from an outdated copy of the key. Leases are distributed at a steady rate to deal with thundering herds, i.e., a specific key that is not in the cache undergoes heavy read or write volume. If a client requests data for a key that has already been leased, the lease request fails, and the client must try again. Typically, the client that owns the lease will have successfully set the data within a few milliseconds. Therefore, when waiting clients retry the request, the data is often cached.

Another optimisation used was memcache pools. Memcached pools are used to distribute data storage across multiple memcached servers, where they work together to handle requests for data. The data is distributed according to the different churn rates, i.e., how frequently the data changes, making it more efficient to group data with similar churn rates.

In a region

In this scenario, one of the three features implemented is an invalidation daemon, *mcsqueal*, that replicates the cache invalidations across all cache servers within a region. The daemon inspects the SQL statements that its database commits (logs), extracts any deletes, and broadcasts these deletes to the memcache deployment in every frontend cluster in that region.

Across regions

The primary technical challenge, in this case, is maintaining consistency between data in memcache and the database.

Each region consists of one database and multiple frontend clusters. One of the regions is named master, meaning that the master database is in this region, while the other regions contain read-only replicas. To keep the databases up-to-date between regions, Facebook relied on MySQL's replication mechanism.

Facebook found an acceptable trade-off by providing best-effort eventual consistency but prioritising performance and availability.

More interestingly, the authors address how the system proceeds when a write is executed from a non-master region. Their approach was to employ a *remote marker* mechanism to indicate that the data in the local replica database are potentially stale and the query should be redirected to the master region.

Single Server Improvements

One appealing feature referenced in the paper is the *Transient Item Cache*. Used to cache short-lived items, it uses a circular buffer or linked list based on the expiration time of an item. Every second, all of the items at the head of the buffer are evicted, and the head advances by one. This way, most keys are still lazily evicted by checking the expiration time on a get request or when they reach the end of the LRU, and some short-lived keys are proactively evicted.

Comparison

In the related work section of this paper, TxCache and Gumball, presented in Sections 3.3.2 and 3.3.4, respectively, are briefly mentioned. Facebook states that memcached requires a more flexible caching strategy, unlike TxCache, where the only possible configurations are what functions to cache and how much staleness the application supports. While Gumball is just specified as an algorithm that uses timestamps instead of explicit version numbers.

Finally, the paper describes the performance and scalability improvements that Facebook achieved by using all the features that help to solve the challenges presented.

3.4 Other Systems

There are numerous other systems and scientific papers that describe new approaches and techniques for improving caching system performance while adhering to the strict requirements of the applications.

Other systems, such as [5, 41, 17, 42, 10, 19, 22], provide excellent insights into how they solved and addressed current caching system issues. The paper [5] focuses on extending traditional transactional caches (such as the TxCache of Section 3.3.2) with more efficient cache policies and makes the case for batch-consistent caching. In the paper [41], Tolia and Satyanarayanan mentioned two important priorities when developing their system: (1) to maintain the consistency constraints of a relational database, and (2) to keep the system transparent to the rest of the application, requiring no further modifications.

Lastly, due to the large number of proposals where newer and more efficient mechanisms have been developed and published, the paper [23] leaves us with a better understanding of the industry by providing an overview of these systems and many comparisons between them.

4

Design

In this chapter, we start by introducing the most important systems for the development of ClearCache: the MongoDB database (Section 4.1) and the Redis caching service (Section 4.2). This is followed by the proposed architecture for the developed system and a discussion about various details that need to be considered in the development phase (Section 4.3).

4.1 MongoDB

Contrary to the various systems mentioned in Section 3.3, we based our system on applications that use MongoDB, a NoSQL database. NoSQL databases are becoming increasingly popular due to their scalability, better performance, flexibility and ease of use compared to traditional SQL databases.

MongoDB [24] is a popular open-source NoSQL database management system. Unlike traditional relational databases, which use tables and rows to store data, MongoDB uses a flexible document data model to store semi-structured data in the form of BSON (Binary JSON) [11] documents. BSON is a binary serialisation format that extends the capabilities of JSON by providing a more efficient and compact representation of structured data, making it suitable for data storage, transmission, and efficient data retrieval in databases.

MongoDB is designed to be highly scalable, making it a good choice for large, complex applications that handle a lot of data. It provides features such as indexing, built-in aggregation, and horizontal scaling through sharding, making it a powerful tool for data processing and analysis.

MongoDB is also known for its simplicity and ease of use, allowing developers to get up and running quickly without spending a lot of time learning the system.

Another important aspect of MongoDB that we will introduce more in-depth is its support for replication. Consequently, we wanted to ensure that ClearCache supported this type of deployment and made the most of the functionalities offered by MongoDB to maintain consistency but make it as efficient as possible.

4.1.1 Replication

Today's database systems must be replicated to ensure redundancy and data availability. Maintaining multiple copies of the data on different replicas provides fault tolerance capabilities upon database server failures. Furthermore, efficient load balancing ensures that client requests are evenly distributed among multiple servers. This distribution can prevent individual servers from becoming bottlenecks, improving application performance. Also, strategically placing additional replicas near end users allows the system to scale more effectively as needed.

Replication has developed into a core technology for building reliable and effective database infrastructures that meet the demands of modern applications and data-intensive workloads.

4.1.2 Replica Set in MongoDB

In MongoDB, a replica set is a group of nodes that maintain the same data, also known as a cluster. It can be composed of nodes that store data and, generally, at most, one arbiter that only participates in primary elections [37].

Throughout this dissertation, we consider a three-member replica set consisting of one Primary with Two Secondary Members (P-S-S), as shown in Figure 4.1. This architecture provides enough redundancy to survive most network partitions and other system failures.

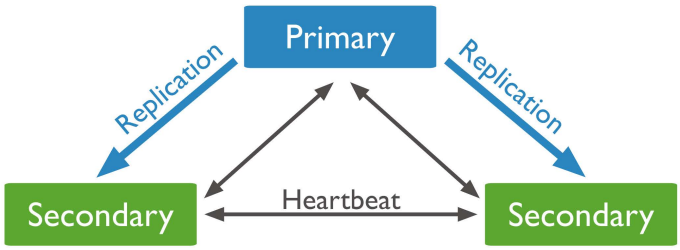


Figure 4.1: Three-member replica set architecture. (Taken from [37])

One of the servers with data is deemed primary, and the others are secondary. The primary receives all write operations, and the read operations can be distributed among the servers with data. The secondaries replicate the write operations executed in the primary through its operations log. This replication process is asynchronous, which may lead to secondaries not always reflecting the most recent data. If the primary stops communicating with the remaining members, the replica set will attempt to elect an eligible secondary as the new primary.

MongoDB offers write and read concerns that are especially applicable for replica sets to control replicated environments' consistency and availability properties. By leveraging both concerns effectively, the level of these properties can be adjusted as appropriate, for example, by relaxing consistency requirements to provide higher availability or vice-versa [38].

4.1.3 Write Concerns

Write concerns determine MongoDB's required level of acknowledgement to confirm the success and propagation of a write operation in the database cluster. These acknowledgement levels define the durability of the writes, i.e. what failure scenarios the writes must survive prior to being considered complete.

It is crucial to select appropriate write concerns for specific requirements since the behaviour of write concerns might affect the application's performance and behaviour. Writing with stricter concerns takes more time but guarantees stronger consistency in case of failures.

Unacknowledged. Unacknowledged is the fastest but least reliable acknowledgement level because the client does not wait for a response from the database server. This means that if the server encountered an error and could not complete the operation, the client would not be aware of it.

It is generally recommended to use a higher level of write concern, in which MongoDB waits for a response from the operation, and the client knows the outcome of the operation.

1. This level is also known as acknowledged since MongoDB waits for the operation to be executed in the primary node before returning to the client.

Data can be rolled back if the primary steps down before the write operation has been replicated to any of the secondaries.

N. MongoDB allows the definition of custom write concerns to specify the number of nodes that must acknowledge the write operation before it succeeds, e.g. "3" to require acknowledgement from three replica set nodes.

Majority. At this level, MongoDB acknowledges the write operation only when it has been written to a majority of replica set members. This ensures that the data is more strongly consistent and resilient against failures. This level is often used in situations where data consistency is critical. The majority is automatically calculated by MongoDB, depending on the deployment configuration. An example of a timeline of a write operation using this write concern to a P-S-S replica set can be seen in Figure 4.2.

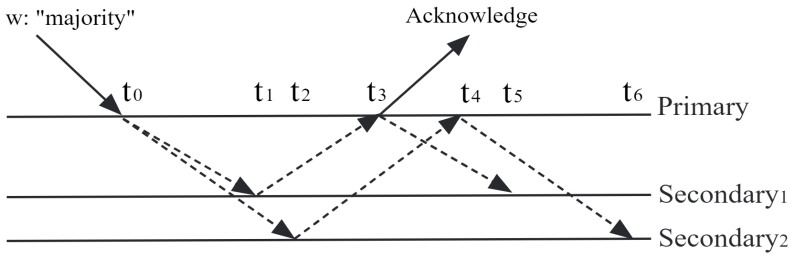


Figure 4.2: Timeline of a write operation to a three-member replica set. (Taken from [15])

The Journaling Option

This boolean journal option specifies if MongoDB acknowledges the write operation immediately after applying the write in memory to the requested number of nodes or only after writing to their journal. This option consists of write-ahead logging to on-disk journal files to provide more durability in case of power or hardware failures in exchange for slower write operations.

However, MongoDB recommends always enabling this feature when running in production, especially for applications that require stronger durability guarantees. The reason for this is that even with majority writing, a write can be rolled back in the event of a transient loss (e.g. crash and restart) of a majority of nodes if the journaling option is false.

Default Level

Since MongoDB 5.0, the default level is "majority", except in some cases involving arbiters, which are nodes that do not hold data and only participate in primary elections [14].

For standalone deployments, if the write concern is "majority", then the journal option defaults to true. If it is "1", then it defaults to false. Similarly, the journal option also defaults to true for replica sets deployments with write concern "majority". If it is any "N", then it defaults to false. This means that, by default, write success will only be acknowledged to the client once it has been committed and persisted to disk on a majority of replicas [15].

4.1.4 Read Concerns

Read concerns control the consistency and isolation level of the data read from replica sets.

Local. A read operation with this level returns the data present on the local node with no guarantee that the data has been replicated to a majority of nodes (i.e. data that can still be rolled back).

Majority. Reading with majority concern ensures that a read operation returns data acknowledged by a majority of replica set members and is durable, even in the event of failure. To make read concern "local" and "majority" performances comparable, read majority does not perform a majority read on the replicas. Instead, each replica set member maintains an in-memory view/snapshot of the data state according to the most recent majority write, called the majority-commit point, and uses it to fulfil this concern level.

This may lead to reads that return different data depending on the node where the data is read. For example, if secondary 1 receives notice from the primary to update its view of its most recent majority committed write before secondary 2, there is a brief period of time where that majority write is not yet available on that particular node.

Linearizable. This is the slowest but strongest read concern level since it returns the most recent data and reflects all previously completed writes across different nodes. Moreover, it may wait for concurrent writes to propagate to a majority of replica set members before returning any data.

The request must be issued to the primary node to read with this level. Unlike "majority", the primary confirms with other members that it is connected to a majority of nodes and, thus, can execute write operations with "majority" write concern.

This addresses the problem stated for read concern "majority", where different reads may return different results depending on the nodes contacted, given that the request is sent to the primary. Furthermore, it also helps not to return stale data in the event of network partitions.

Finally, this level can only be used if the filter specified in the query of the read operation identifies a single document.

Default Level

The default level is "local" for both reads against the primary or against secondaries [15]. As previously stated, it can return data that may be rolled back.

Applicable Levels for Sharded Cluster

MongoDB offers a couple of read concern levels that provide different additional behaviours when the database is configured as a sharded cluster. These levels are "available" and "snapshot". They were not addressed in more detail, considering we do not focus on sharded clusters in this dissertation.

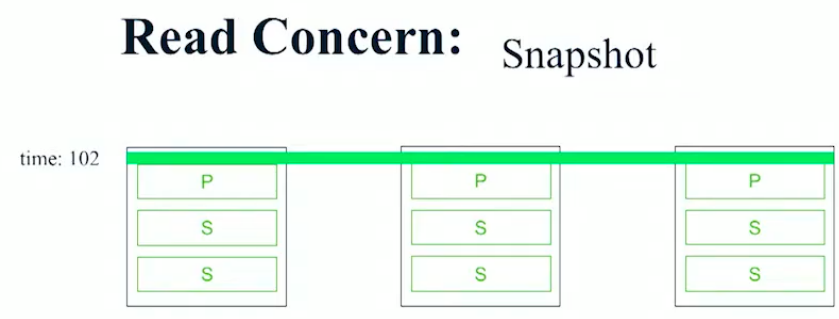
Although the "available" level is still usable for configurations that are not sharded (i.e. standalone or replica set deployments), its behaviour is identical to the "local" level.

We resort to the "snapshot" level to get a snapshot point in time consistent across all cluster shards (Figure 4.3(a)), which is used especially in the context of transactions (more on transactions in Section 5.8). When using the "majority" or "local" levels, the snapshots are only consistent per partition, as seen in Figure 4.3(b). When using level "snapshot", coordinating a snapshot search throughout all shards of the cluster may be expensive as all shards need to agree on the same consistent point in time to use.

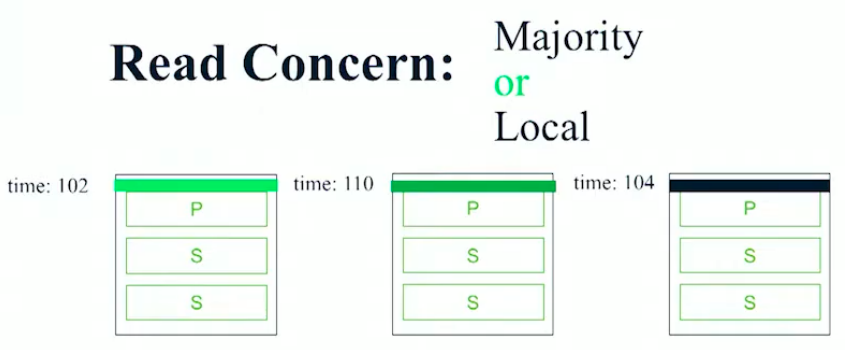
4.1.5 Read Preferences

Besides concerns, the read preference is also an essential tool to manage and adapt the behaviour of read operations in MongoDB. It defines how read operations should be distributed across replica set members, effectively giving the client control of where read operations retrieve data depending on its application's needs.

Primary. Reads from the primary and ensures the return of consistent data by reading the most recent data. This is the default mode, and its functioning is shown in Figure 4.4.



(a) Consistent point in time across all of the shards.



(b) Consistent snapshots per partition.

Figure 4.3: Contrast between different read concern levels in a sharded deployment.

A variation of this mode is named "primary preferred", which can read from any secondary member if the primary is unavailable. This offers a balance between consistency and fault tolerance.

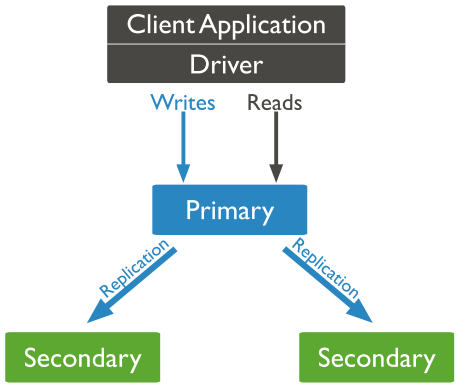


Figure 4.4: Primary Read Preference Mode (Taken from [37])

Secondary. Reads from secondary members of the replica set. It is suitable to relieve the primary from read traffic in read-heavy workloads. However, reading from the secondary may come with the disadvantage of returning slightly outdated data compared to the

primary. Its functioning is presented in Figure 4.5.

A variation of this mode is named "secondary preferred", which can read from the primary if no secondary member is available. This offers a more balanced load while maintaining fault tolerance.

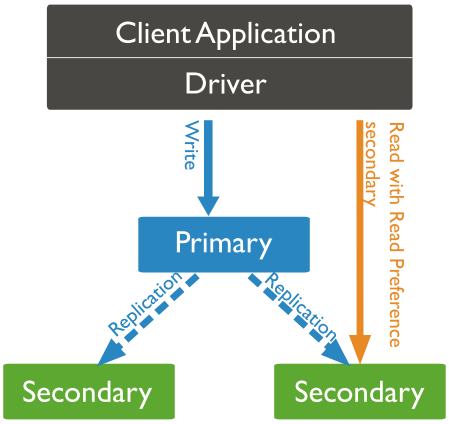


Figure 4.5: Secondary Read Preference Mode (Taken from [37])

Nearest. Reads from the replica set member with the lowest network latency, providing low-latency reads by selecting the physically closest replica member.

It is important to note that all read preference modes except "primary" may return stale data because secondaries replicate operations from the primary in an asynchronous process. If it is expected to use a mode other than primary then the application should tolerate stale data.

4.2 Redis

In our system, we decided to make use of a Redis cache over Memcached, given its numerous features and support for complex data types. A more in-depth description and comparison of both Redis and Memcached is presented in Sections 3.2.1 to 3.2.3.

4.3 Proposed Architecture

In this dissertation, we developed a caching system in the form of a library that intercepts requests from the application layer to the database and transparently manages the cached data and its accesses while maintaining consistency between the database and caching layers. The components that helped to achieve this system are introduced in the following subsections.

4.3.1 API

The goal was to maintain the same API used by a client application to communicate with MongoDB. As a consequence, the client's application logic code remains unchanged from that used to access the database directly while simultaneously benefiting from caching and not requiring the developer to worry about cache usage.

For the system to accomplish the desired behaviour of intercepting requests without explicit changes in the application's code, we developed the API to match the official MongoDB driver to connect to Java applications.

This system is eligible for application with architectures like the one displayed in Figure 4.6.

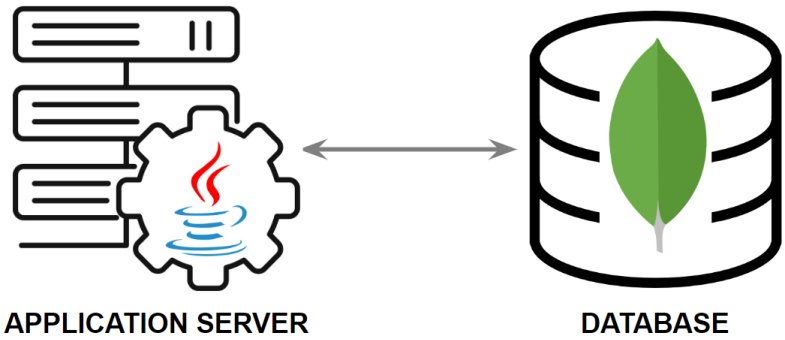


Figure 4.6: Outline of original application architecture.

That being said, a system with functionalities similar to ClearCache is easy to reproduce for applications that employ other programming languages or even other databases, as long as there is an established driver that connects both to recreate the API.

With the API matched by ClearCache, all client requests are redirected to ClearCache. For each method, the system decides if the request is routed to the database or if it applies a custom logic to write or access data from the cache.

Figure 4.7 shows an outline of the architecture of the applications using our system and how its components interact. The developed library, ClearCache, runs in each application server, and instead of always going to the database, it controls whether it can access data from the cache.

The key challenge that must be addressed to achieve the proposed design and implementation of the caching system is to identify which consistency model should be used in the system and develop the necessary algorithms to ensure that all accesses comply with the model from the database. For this purpose, it is necessary to define the protocols that will be executed during the read and write operations, with special attention to the write operations to invalidate or update the cache entries.

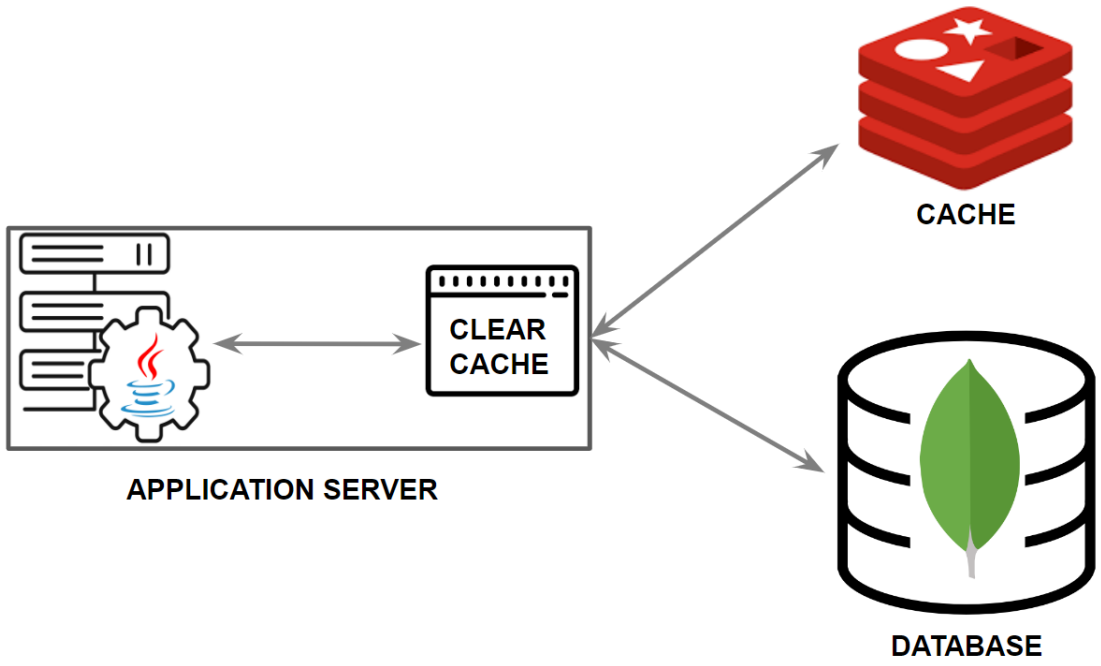


Figure 4.7: Outline of resulting application architecture.

4.3.2 Timestamp

Keeping the same data in two different layers comes with the inherent responsibility of keeping them consistent with each other. This is simple if only one application server exists, essentially by storing the data in the cache when it is inserted in the database and removing it when it is deleted. However, this scenario is not realistic because most applications and services have numerous clients, and only one application server cannot keep up with all the requests. When multiple application servers exist, this becomes a significant problem that can seriously jeopardise data consistency.

Considering the example displayed in Figure 4.8, two application servers exist, and two clients send write requests for the same variable. Both application servers will try to insert the data into the database and subsequently store it in the cache.

The issue arises from the inability to coordinate the order of the requests to the data layers. From Figure 4.8, we can observe that the first application server sends the first request to the database, but then it gets delayed for some reason, and in that time, the second application server executes both requests. Only afterwards is the first application server able to perform the write to the cache. This results in the two data layers having different versions of the data and is just one of many scenarios where the data can become inconsistent.

To resolve this problem, we integrate a timestamp into each document. The rationale behind this is that the timestamp denotes the document's creation or last updated time. This facilitates the management of the cached data, and it is a crucial strategy for keeping the most updated version readily accessible. When a document is updated, its timestamp

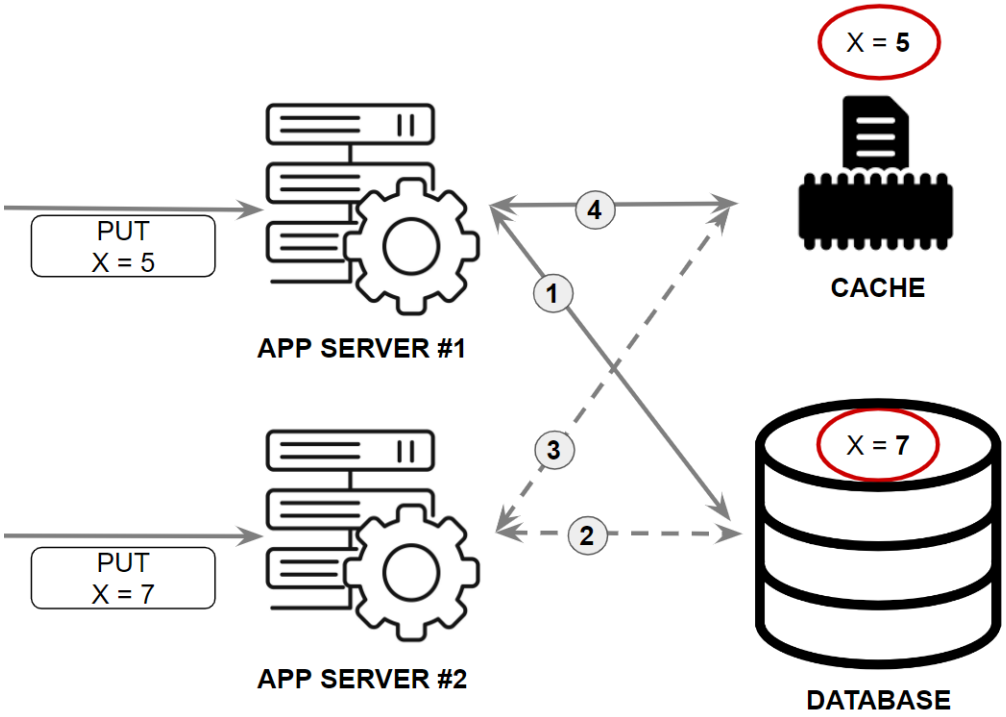


Figure 4.8: Scenario that produces data inconsistency.

changes, indicating that it is newer. By leveraging these timestamps, ClearCache only writes to the cache if it verifies that the cached document has a lower timestamp than the one it is trying to write, minimising the risk of serving stale or inaccurate data to clients.

This is the core idea that is used in ClearCache to maintain the cached data consistent, and we further explain how we implemented it in Chapter 5.

5

Implementation

This chapter is structured into several sections that focus on various aspects of the ClearCache system and its implementation. In Section 5.1, we present how the system's API was developed. In Section 5.2 and in Section 5.3, we introduce some considerations taken regarding fields of the documents inside MongoDB. Section 5.4 and Section 5.5 consist of the implementation phase for the developed methods, and Section 5.6 focuses on the system's logic associated with Redis. Section 5.7 and Section 5.8 both discuss how the system supports MongoDB features, such as replication and transactions. Finally, in Section 5.9, we present additional factors to consider when using ClearCache.

5.1 API

To accomplish the desired behaviour of the application's code being left unchanged while talking to ClearCache instead of the database, the developed API for the system was based on version 4.10.2 of the MongoDB synchronous driver for Java [25]. This ensures both APIs match, and ClearCache offers the same methods as MongoDB. Some specific methods forward the requests to the database, in essence, calling the database's corresponding method, while others carry out the extra logic mandated by the ClearCache system.

To achieve this middleware behaviour for our system, we started by modifying the class *MongoClients*. This class creates instances of the standard MongoDB client of type *MongoClient*.

This class provides multiple methods with the same name, `create`, but different parameter lists (i.e. different types or numbers of parameters). This is known as method overloading, and the Java compiler determines which method to call based on the arguments provided during the method invocation. Additionally, the class is designed to eliminate redundancy in the code by requiring all `create` overloads to use the method that effectively creates the client. This method is shown in Listing 5.1.

We modified this class to return a custom object we defined, MongoClientImpl, also of the type MongoClient, but with a different behaviour for the method that creates the client. The new implementation of the class MongoClients is depicted in Listing 5.2. The

Listing 5.1: Original implementation of MongoClients class

```
1  // MongoClients.java
2  package com.mongodb.client;
3  import com.mongodb.client.internal.MongoClientImpl;
4  import ...
5
6  public final class MongoClients {
7      ...
8      public static MongoClient create(MongoClientSettings settings, @Nullable
           ↪ MongoDriverInformation mongoDriverInformation) {
9          Builder builder = mongoDriverInformation == null ? MongoDriverInformation.builder()
               ↪  : MongoDriverInformation.builder(mongoDriverInformation);
10         return new MongoClientImpl(settings, builder.driverName("sync").build());
11     }
12     ...
13 }
```

Listing 5.2: Implementation of MongoClients class in ClearCache.

```
1  // MongoClients.java
2  package clearcache.db;
3  import clearcache.db.impl.MongoClientImpl;
4  import ...
5
6  public final class MongoClients {
7      ...
8      public static MongoClient create(MongoClientSettings settings, @Nullable
           ↪ MongoDriverInformation mongoDriverInformation) {
9          MongoClient client = com.mongodb.client.MongoClients.create(settings,
               ↪ mongoDriverInformation);
10         return new MongoClientImpl(client);
11     }
12     ...
13 }
```

new method starts by creating a MongoDB client instance in Line 9, calling the original method shown in Listing 5.1.

Instead of returning this object to the client like the original, in Line 10, an instance of the custom MongoClientImpl is created with the original MongoClient passed as an argument and then returned.

Depicted in Listing 5.3 is the code of the custom implementation of the MongoClient interface, MongoClientImpl. It wraps around an existing MongoClient instance and provides custom implementations for certain methods, like the `getDatabase` in Line 13, while delegating others to the underlying MongoClient instance, like the `listDatabaseNames` in Line 18.

Custom implementations of MongoDatabase and MongoCollection, specifically MongoDatabaseImpl (as seen in Line 15) and MongoCollectionImpl, were created to accomplish the same function as MongoClientImpl, which intercepts calls to client operations. They provide a way to intercept, customise, and extend MongoDB driver functionality to

Listing 5.3: Implementation of MongoClientImpl class in ClearCache.

```
// MongoClientImpl.java
package clearcache.db.impl;
import clearcache.db.impl.MongoDatabaseImpl;
import ...

public class MongoClientImpl implements MongoClient {
    private final MongoClient client;

    public MongoClientImpl(MongoClient client) {
        this.client = client;
    }

    public MongoDatabase getDatabase(String s) {
        MongoDatabase db = client.getDatabase(s);
        return new MongoDatabaseImpl(db);
    }
    ...
    public MongoIterable<String> listDatabaseNames() {
        return client.listDatabaseNames();
    }
    ...
}
```

database and collection operations, such as insertions, queries, updates, or deletions.

Although ClearCache was developed with MongoDB and its corresponding Java driver in mind, similar systems can be recreated with other languages and even databases as long as there is an existent driver to connect both layers and allow the reproducing of its API.

5.2 ID Field

In MongoDB, each document stored in a collection requires a unique `_id` field that acts as a primary key for the document inside the collection. This field is always the first field stored for each document in the database, and it may contain values of any BSON data type other than an array. Additionally, if it includes subfields, their names cannot begin with a $ symbol. By default, MongoDB also creates a unique index on the `_id` field while creating a collection.

During inserts or upserts, the MongoDB driver automatically generates an object of type *ObjectId* for documents that do not specify a value for the field `_id`. This *ObjectId* object consists of a 24-character hexadecimal string. It comprises 12 bytes, where the first 4 bytes represent a timestamp of the object creation in seconds since the Unix epoch; the next 5 bytes stand for a random value unique to each machine and process; and the last 3 bytes correspond to an incrementing counter, that is initialised to a random value. A valid hexadecimal string looks like "`507f191e810c19729de860ea`".

Considering that Redis requires the keys to be strings and we use the document's `_id` field as part of the key, to better map documents stored in MongoDB to values stored in

Redis, we decided to restrict the possible types of the field `_id` only to *String*. This decision ensured the compatibility between the primary key in MongoDB and Redis keys and avoided ambiguity and casting errors on lookup operations. For example, if a document to insert has a field named `_id`, and its value is not of type *String*, then an exception is raised. If it is, then the document is inserted. If the field is missing, instead of letting the MongoDB driver automatically generate an *ObjectId* and pass it to the database, we create the *ObjectId* ourselves and use its value as a *String* for the new document. We consider this not only for the `insert` method but also for the `update` methods where the option for upserting is available.

5.3 Timestamp Field

For the timestamp added to each document, an extra field, `_ts`, is added and updated automatically by the system, representing the timestamp in which the document was inserted or last updated.

To implement the field `_ts` that will hold a timestamp, there were two possible data types to represent it: type *Date* or type *Timestamp*.

To help choose which type to use, we must remember that a problem with distributed applications and systems is that clients may have differing system clocks, leading to a false order of operations when these timestamps are generated by the clients. For this reason, it was important that the value of the timestamp be generated by the database server instead of by the client.

Using an object of type *Date* is recommended by MongoDB for most use cases. However, only by converting the insert into an upsert and using the *$currentDate* update operator could it be ensured that the added *Date* type timestamp reflected the time according to the database server's clock instead of the client's. Unfortunately, implementing the insert method as an upsert would require additional database queries and extra processing logic to provide full support for the expected insert method's error cases, slowing the method's performance.

Considering we can add a *Timestamp* directly with insert, an insert is faster than an upsert, and the error handling is already correct, we chose the *Timestamp* type to represent the field over the *Date* type. An object of the type *Timestamp* has two parameters: `t` and `i`. The first parameter represents the current time since the Unix epoch, and the second is used for ordering operations executed within a given second. With the timestamps generated by the server, the operations are automatically ordered, and there is no ambiguity between two write operations because they cannot have the same *Timestamp* value.

The one potential benefit of creating the `_ts` field *a priori* on the client would be to process the write operations in parallel for both MongoDB and Redis instead of executing in MongoDB and then in Redis. However, the advantage assumes that the operations in MongoDB run without errors. Otherwise, the changes made to Redis would need to be undone before any other client reads that data to maintain consistency with the database's

actual data state. Furthermore, due to the lengthy and somewhat complex syntax for operations in MongoDB that would need to be addressed inside the system, we concluded that it was better to let the database handle the request rather than introduce the logic required to process all the operators available in MongoDB.

```
{
  _id:"1",
  name:"John",
  age: 25,
  groups: ["news", "sports"],
}
```

(a) Original document.

```
{
  _id:"1",
  name:"John",
  age: 25,
  groups: ["news", "sports"],
  _ts : Timestamp(1579728101, 1)
}
```

(b) Document stored in MongoDB.

Figure 5.1: Difference between original and stored document.

Figure 5.1(a) represents how a document to be inserted is passed from the application to the ClearCache system, and Figure 5.1(b) shows how that document is actually stored in the MongoDB database after adding the *Timestamp* field.

5.4 Write Methods

The write methods needed to be adapted in order to not only carry out their normal behaviour but also introduce the logic that maintains the data consistency between both the database and cache. In this subsection, we present the changes made in them in order to apply the consistency checks and safeguards.

5.4.1 Insert Methods

The insert methods receive one document or a list of documents to insert, respectively, and one InsertOneOptions or InsertManyOptions parameter with the options for the respective operation. The return type for each method is InsertOneResult and InsertManyResult. The first returns the `_id` of the inserted document, whereas the second returns a map of the index of the inserted document to its `_id` value.

`insertOne`

Considering that the *Timestamp* added to the inserted document is added by the server, and we need the resulting document to be stored in Redis, the method was redefined to provide this behaviour.

As a first approach, we tried using the method `findOneAndUpdate`, with the upsert option set to true to simulate an insertion and, most importantly, to take advantage of its return object. As stated in Section 5.3, the approach of upserting would require extra database requests and logic in the system, leading to higher complexity and latencies. The

reason behind this is that the method returns the document before or after the execution of the operation specified by the field ReturnDocument in the operation's options. We would be interested in the document after the insertion since it includes the *Timestamp* added.

Despite this logic being correct for inserts that do not throw errors, when it comes to handling duplicate key errors with this implementation, it becomes significantly harder to do. This is because we return the resulting document and, thus, do not have any information on whether the operation resulted in an update or an upsert.

To maintain the correctness of the method, we decided to leave the method as an insert but include an additional `find` query by the `_id` field of the inserted document to get its final version with the *Timestamp* and store it in the cache.

In the case of concurrent updates or deletes to the document, the method's logic is correct, given that the latest version of the document will be in the cache - we explain how we achieve this in Section 5.6.

To obtain faster insertions, we made the method's portion of querying and storing the newly inserted document into the cache asynchronous. The rationale behind this decision was to be able to return to the client after executing the insert component of the method. After this, the `find` query and the cache insertion are submitted to an *ExecutorService* with a thread pool that runs in the background. This approach enhances the application's responsiveness because it does not block the client from performing other tasks while the background thread from the pool is running.

If, for any reason, the background thread does not execute, the logic of the application remains correct, and the client can continue to function normally. This is a result of being an insert operation, and therefore, it is not critical to go to the cache to invalidate entries, unlike updates and deletes.

`InsertMany`

The method's logic was left unchanged purposefully because it is recommended to insert multiple documents efficiently. Instead of sending the documents individually, the driver sends batches to the database, leading to fewer communication exchanges to insert all the documents.

With this in mind, we chose to leave the method unchanged as an option for the application to write multiple documents, as it would not be sensible to insert each document individually to insert them in the cache. In other words, this method does not add the inserted documents to the cache.

While the logic of the method remains unchanged, we consider that each document must have an `_id` of type *String* and a field `_ts`, where the server will add its *Timestamp*, before sending the operation to the database.

5.4.2 Update Methods

Contrary to the inserts, both update methods receive the same arguments and return the same *UpdateResult* object. They receive two *Bson* arguments, a filter and an update, and one *UpdateOptions* argument with the options for the operation. The *Bson* filter represents the query to get the documents to be updated, and the *Bson* update is the changes to be applied to those documents. `updateOne` only updates the first document returned by the query, whereas `updateMany` updates all the documents returned.

The *UpdateResult* specifies the number of documents that matched the query, the number of documents that were effectively updated, and the `_id` of the document if an upsert occurred.

As mentioned above, while using the `findOneAndUpdate` allows us to get the document after the execution, it also makes it impossible to know if a document existed before the insert without adding extra communications with the database. For this reason, the behaviour would be as similar as possible, compromising some of the information returned by the original method.

`updateOne`

This method uses the `findOneAndUpdate` method to update and read the resulting document in MongoDB. The retrieved document has the fields already updated, including a new *Timestamp* associated, and it is inserted in the cache, checking that the new *Timestamp* is greater than the *Timestamp* in the cached version of the document, if it exists. This check ensures that an older version of a document is not stored over a more recent one. It can happen if a concurrent request runs faster while the first operation is slower between the update and the insertion in the cache.

`updateMany`

Contrary to `insertMany`, we altered the method's behaviour so that when a document update is done on the database, we must act accordingly with the documents in the cache to ensure no inconsistencies between the data on the two layers. The implementation of this method was altered to simulate the original behaviour while also being able to update each required document.

Firstly, to search for the specific documents that are supposed to be affected by the update, we perform a `find` using the filter of the update operation to get the relevant ones. If the list of documents to update is empty and the option to upsert is false, then the operation ends with no document changed. If the option to upsert is true, our implementation of the method `updateOne` is called with the arguments received in the original method, which will insert the new document into MongoDB and Redis with the `Timestamp`. If the list of documents is not empty, we iterate over it and call our `updateOne` method for each one. The filter used as an argument for the `updateOne` consists of the `_id`

field of the document and the original filter to ensure that the specific document exists and is still relevant to the update since between the *find* operation and the moment of the update, other operations could have executed. The update argument maintains the same; the only difference for the options is that the upsert is false, independently of what the client requested.

The object returned is of type UpdateResult and uses the size of the list returned by the `find` operation as the number of matched documents, and the number of modified documents is incremented while iterating through the list if the document update was successful. If the operation was an upsert, it follows the same logic specified in `updateOne`.

5.4.3 Delete Methods

Like the updates, the delete methods receive the same argument and return the same object type. The delete methods only receive a Bson filter as an argument and one DeleteOptions argument with the options for the operation. The Bson filter represents the query to get the documents meant to be deleted. The difference between `deleteOne` and `deleteMany` follows the same logic as `updateOne` and `updateMany`.

This method returns a DeleteResult object specifying the number of deleted documents.

`deleteOne`

Like `updateOne`, this method must alter its behaviour to use the `findOneAndDelete` method, which follows the same logic as `findOneAndUpdate`, but in this case, it always returns the document before the delete operation. This way, retrieving the removed document and removing the corresponding cache entry is possible without sending extra communication requests to the database.

Considering that it is the method `deleteOne`, this value can only be 0 or 1, depending on whether the document returned is `null`.

`deleteMany`

This method also needed altering for the same reasons as `updateMany`. Implementing this method was much more straightforward than the `updateMany` since here it is possible to use our `deleteOne` method with the same filter inside a do-while loop. The cycle breaks when no document is deleted, meaning all documents matching the criteria were deleted.

To know the total number of deleted documents for the *DeleteResult* to return, we utilise a counter that is incremented each time the cycle runs.

5.5 Read Methods

The primary method for performing read operations, the `find` method, was modified to read whenever possible from the cache or otherwise go to the database. Independently

of the method's flow, the type of the object returned to the client must be the same. In this case, the returned type is a *FindIterable* object, where the client can use its methods to specify options for the operation, such as how many and how to sort the returned documents.

5.5.1 Returned Cursor Object

We developed a custom *FindIterableImpl* class that implements the *FindIterable* interface and wraps around the original *FindIterable* instance, passed as an argument, to provide the client with the expected object type, similar to the *MongoClientImpl* class depicted in Listing 5.3.

With the returned object, the clients can call the methods `iterator` or `first` to get the desired documents. The `iterator` method returns an object of type *MongoCursor*, which extends the *Iterator* interface, and the client can use it the same way as any Java iterator. The method `first` returns the first document of the list of documents returned by `iterator`.

Considering that the client expects an object of type *MongoCursor* even for method executions that lead to cache hits and do not require contacting the database, it was need to recreate the creation of an object of the same type.

By analysis, we found that the *MongoCursor* constructor required an object of type *BatchCursor*, whose constructor, in turn, required an object of type *QueryResult*.

To set up this *MongoCursor* object, we create a *QueryResult* object using the list of documents returned by Redis as an argument, create a *BatchCursor* object using this last object as an argument, and then pass this last object as an argument to the *MongoCursor* constructor. The Figure 5.2 may be helpful in following the described procedure/approach.

The only trouble we had during this process was that since the class implementing the *BatchCursor* interface from the MongoDB driver is not public, we could not call its constructor. Consequently, we designed a custom *QueryBatchCursor* class, which implements the *BatchCursor* interface and offers all the necessary methods to be used by the *MongoCursor*.

5.5.2 Iterator Method

With the return type taken care of, we implemented the intended behaviour for the `iterator` method.

Currently, ClearCache only supports searches by the `_id` field of each document, given that the `_id` is used in the document's key in Redis. With this in consideration, we start by checking the filter parameter specified by the client when creating the FindIterable object. If this parameter contains only one condition and its key is `_id`, i.e., if the query that the client intends to perform is a document search only by `_id` value, then the system will go to the cache to look for the document with that value. Otherwise, it goes straight to the database to perform the query.

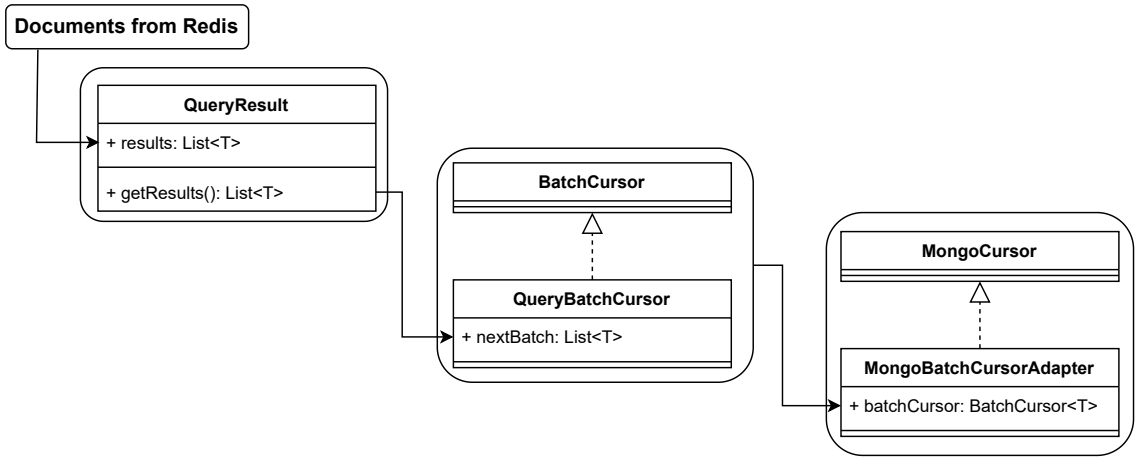


Figure 5.2: Creation of *MongoCursor* object from Redis documents in `iterator` method.

Upon checking in the cache, if we are in a case of a cache hit, the object of type *MongoCursor* is created with the retrieved document, as described in Section 5.5.1, and returned to the client. If we are in a cache miss, the system queries the database, and the retrieved document is stored asynchronously in the cache by submitting a task to the same *ExecutorService* mentioned in the `insertOne` method. Again, this approach increases the method's performance since the client does not need to wait for this step to execute.

There is a specific scenario where, in a cache miss, the document is not added to the cache. A client can specify through a parameter `projection` which fields the returned document should include or exclude. The document is not stored in the cache if the `projection` parameter is defined because doing so would result in the caching of an incomplete copy of the document, preventing subsequent read requests from using the cache or retrieving the entire document.

5.6 Redis Logic

The conditional updates of the cache based on the *Timestamp* play a key role in the function of ClearCache. However, with this verification step, there is a new problem that needs to be addressed. Given that to execute the verification, ClearCache performs a `GET` operation to Redis for the required document, a conditional portion where it decides if it writes or not the new document and the subsequent write, there is a possibility for data inconsistency.

Similarly to the scenario displayed in Figure 4.8, there is a new scenario where two application servers do the `GET` operation one after the other, and both come to the conclusion that they may write the new document to the cache. This can lead to the more recent version of the data to be overwritten by an older version.

5.6.1 Redis Functions

To execute multiple operations, in our case, the `GET` operation and subsequent write atomically, there are two mechanisms offered by Redis: transactions and Lua scripts. We decided to go with Lua scripts [39] because they can do everything transactions do and usually are simpler and faster to perform operations that depend on stored data [35]. They run on the Redis server, and it is possible to build robust and customised functions for each application. They can contain business logic and thus relieve some of the pressure on the application side while doing it efficiently and atomically.

In Redis 7.0, Redis Functions [32] were introduced to manage and run Lua scripts. Redis manages functions as an integral part of the database and ensures their availability via data persistence and replication, making them as durable as the data itself. To use functions, the corresponding script needs to be loaded first, and then they are available for repeated use by all connected clients. In this case, loading a function to the database becomes an administrative deployment task, which separates the script from the application. The execution of a function is still guaranteed to be atomic, given that it blocks all other activities in the server from all connected clients until complete. Consequently, functions are meant to be executed quickly.

ClearCache calls these Redis functions inside the write methods to correctly update the data in the cache.

5.6.2 Document Structure in Redis

Regarding the document structure in Redis, we store each document as a JSON-encoded string in a single key. This is a simple and efficient option, as JSON parsing is quick and allows storage of the complete document without concerns about fields with nested objects inside them or different structures.

The key that identifies each document comprises the collection's *Namespace*, characterised by the database and collection name, and the `_id` field. This makes the key unique across all databases and collections of MongoDB. For example, with a database named `"mydb"`, a collection called `"users"`, and a document with an `_id` equal to `"John"`, the resulting key for Redis would be `"mydb.users.John"`.

5.7 Supporting replicated MongoDB in ClearCache

For supporting a replicated deployment of MongoDB, several problems arise.

Firstly, clients can specify which concerns and settings they desire to use in their operations, which means data can be written with less rigorous options, leading to data that can be rolled back despite MongoDB acknowledging the write operation. This can introduce inconsistencies between the data layers whenever ClearCache stores the data to Redis immediately after receiving the acknowledgement from MongoDB, and then the data rollback occurs in the database.

To solve this limitation, we opted to add a TTL to all documents stored in the cache to ensure that keys are occasionally refreshed instead of cached permanently. The TTL guarantees that data stored in the cache that is rolled back from the database will eventually be overwritten with a consistent version or discarded. The TTL was implemented by utilising the "EX" parameter and an associated value that represents the entry's expiration time in seconds in the `SET` Redis command.

The second problem also arises from the fact that clients may specify different concerns and preferences to use for their operations. This poses a new problem for the cached data, given that data written to the cache with some write concern level may not respect the level of following read requests, which require a stricter concern level. Correctly handling multiple clients with different write and read concerns is required to ensure data consistency. When these concerns are incompatible, it can introduce inconsistencies in the cached data and, thus, in the data returned to the client.

To solve the second problem, we decided to add a field `_w` to all documents stored in the cache. This field consists of a *String* with possible "majority" or "not_majority" values. The value reflects the write concern used by the client who performed the write operation in the primary. If the write concern utilised was "majority", then the cached document will have `_w` with the value "majority". Otherwise, it has the "not_majority" value. This field determines when read operations can or cannot read the data from the cache.

If there is no cached document (i.e. a cache miss), a replica set member is queried with the concern and preference specified by the client for the operation and the returned document is inserted into the cache. This document will have its `_w` determined by the read concern employed in the operation. In the case of "local" or "available" read concern, it is "not_majority". With any other stricter read concern, the field's value is "majority".

The following examples describe the read operations behaviour when confronted with a cache hit and upon the `_w` field of the document currently present in the cache:

(A) If both write and read concerns are compatible, i.e. one of the below conditions is true, ClearCache returns the cached document:

  (i) "not_majority" value and "local" or "available" read concern;

  (ii) "majority" value and any other read concern that is neither "local" nor "available";

(B) If the document was written using "majority" write concern and the client makes a read request with "local" read concern, then ClearCache can also return the cached document. In this example, the client might not have asked for this concern, but since the data in the cache is acknowledged by a majority of replicas and also faster to read from Redis than MongoDB, we return this data that can be a bit stale compared to the data in that instance, if read in a secondary, but much more durable;

(C) Finally, upon a document write with concern "1", the primary acknowledges it, and ClearCache inserts it into the cache with `_w` field as "not_majority". Next, if a

"majority" read request occurs for the same document, the client cannot read directly from the cache as the concerns are incompatible. Therefore, the system queries the database with a stricter concern and uses the resulting document to check the cache consistency. The possibilities are:

(i) If the *Timestamp* of the query's resulting document is lower than the one inside the cache, it means that the latest write has not been replicated yet, so we do not update Redis;

(ii) If the *Timestamp* of the query's resulting document is equal to the one inside the cache, it means that the latest write has been replicated, and thus, we update the `_w` field of the cached document to "majority";

(iii) If the *Timestamp* of the query's resulting document is higher than the one inside the cache, it means that there is a new latest write, and for that reason, we insert the new document to Redis with "majority" value for the `_w` field.

The trade-offs resulting from the implementation changes and the decisions we made include the following:

- Clients reading from a secondary member of the database replica set can quickly read more recent data by accessing the cache before the asynchronous replication takes effect. While this might be considered an inconsistency when secondaries access the cache, we see this as a benefit of using the cache;
- Clientes reading from the primary with "local" read concerns cannot read the most recent data if there is a document with a "majority" value for `_w` in Redis. In this case, reading from the primary may also return stale data;
- Upon a cache miss in a secondary-read client, the cache will contain a version of the secondary's data. This data may be out of sync with the primary, leading to inconsistencies when the primary instance reads from the cache.

5.8 Supporting MongoDB transactions in ClearCache

Although MongoDB provides atomicity out of the box for operations on a single document, an operation affecting multiple documents is not considered atomic. MongoDB supports multi-document transactions to ensure proper and predictable behaviour in scenarios where a sequence of operations is required to perform atomically [26, 28]. Transactions allow multiple read and write operations to be grouped and committed as a single unit. Conflicts between operations can be avoided, and data consistency is ensured by using transactions. However, transactions should be used correctly as they come with additional overhead compared to individual operations. As of version 7.0.0 of MongoDB, multi-document transactions are only available in replica sets (even if with a single member)

or sharded clusters. Additionally, the read preference used for read operations must be primary, i.e. writes and read operations all go to the primary node.

All the methods described in Section 5.4 and the `find` method of Section 5.5 have an optional argument of type *ClientSession* where a client can pass its current session to the database, which can have an active transaction. When creating a session, the client can specify the read and write concerns to use.

To support transactions, some methods would have to be adjusted, and thus, we implemented a class, *ClientSessionImpl*. *ClientSessionImpl* acts as a wrapper of the Java MongoDB driver implementation of *ClientSession*. This way, it was possible to maintain the original methods and respective behaviours while also adjusting others to support the execution of transactions and the simultaneous use of the system since it also needs to manage the cache.

Write operations in transactions are all isolated, meaning that inserts, updates and deletes are only visible to others once the transaction is committed. Consequently, we must also wait for the commit time to act accordingly in the cache for each operation. The `transactionOperations` variable stores executed write operations. It is a map *Map<String, Map.Entry<Document, WriteModel»* where the outer key represents the key used in Redis, and the associated value is an entry that holds information about the resulting document and the type of write operation (insert, update, delete) through the *WriteModel* interface. It uses *InsertOneModel* for inserts and updates and *DeleteOneModel* for delete operations. This structure helps manage and later retrieve data about these operations efficiently.

This was essential for efficiently implementing a read operation inside a transaction. For example, it would be required to go to the database if there were a request to read a document inserted in the same transaction as it is not inserted into Redis yet. However, we can quickly retrieve the stored document using the `transactionOperations` variable. In case of an insert or update, the document stored is returned in a *MongoCursor*. In case of a delete, `null` is returned if the key associated with that document is present, and thus, an empty *MongoCursor* is returned to represent its deletion.

Still considering read operations inside transactions, but in this case, reading a document not written in the transaction, the system cannot simply read the document if present in the cache to preserve the transaction's isolation. To do this, a transaction creates a snapshot of the data upon its first operation; thus, all data reads are from a version at that point in time. To simulate this behaviour, we store a variable `firstOperationTS` in the *ClientSessionImpl* class, which stores the *Timestamp* of the first operation of the transaction. Using this *Timestamp*, the system only reads the document from the cache if the timestamp in the variable is bigger than the *Timestamp* stored in the document. Otherwise, it goes to the database to execute the operation.

To compute the variable `firstOperationTS`, we consider the property "Operation Time" of the *ClientSession*, which returns the *Timestamp* of the last acknowledged operation for the session.

The variable starts as `null`, and all methods that receive a client session (not only the

redefined ones in Section 5.4 and the `find`) call the new method `setTransactionStartTs` that sets the variable to the session operation time if there is an active transaction and the variable is `null`.

A case requiring extra handling was the case where the transaction's first operation was a read operation that read from the cache; thus, it made no requests to the database. Because the read operation did not go to MongoDB, this leads to "Operation Time" returning `null` and the variable incorrectly remaining `null` when the second operation of the transaction is processed and it calls the `setTransactionStartTs` method.

This was solved by manually setting the session's operation time using the method `advanceOperationTime`, which takes a *Timestamp* as an argument. We use the `TIME` method from Redis, which returns the Unix epoch, to set the operation time. This way, the following operation can set the *firstOperationTS*.

Upon committing the transaction to the MongoDB database, we iterate through the map `transactionOperations`, and for each entry, we process it accordingly. If the *WriteModel* in the entry is of type *DeleteOneModel*, we use the key to delete the document from the cache. If it is *InsertOneModel*, we use the key and document stored to put it into Redis, also adding the `_w` field into the respective write concern used for the transaction. In the end, the `transactionOperations` map is cleared, and the variable `firstOperationTS` is set to `null`.

When the transaction is aborted, `transactionOperations` and `firstOperationTS` are reset in a similar way to when committing.

5.9 Other considerations to use ClearCache

In this subsection, we present some key aspects that must be taken into careful consideration when deciding the integration of the ClearCache system within an application. This is especially true if it is an already existing application with a legacy database.

- Any already running application can integrate ClearCache into its own configuration, assuming that the `_id` fields are of type *String* and that there is no `_ts` or `_w` fields in the documents used by the application for its own logic.

- Read operations of a document that does not include the `_ts` field while using ClearCache will prevent the document from being stored in the cache. Since the document is only stored in the database, the operation results in a cache miss, but given that it does not have the `_ts` field, we cannot compare its recency with other subsequent operations that try to store it in the cache. To deal with this setback, documents must be first updated so that the `_ts` field is added with the corresponding update *Timestamp*. Newly inserted documents will already include the `_ts` field.

- When executing a read operation in MongoDB, one possible option for the client is to define a projection that specifies which fields of the document to include or exclude when returning. Since we use all the original options, a read operation where a cache miss occurs will result in writing an incomplete document in the cache. To overcome this issue, the read in MongoDB should be executed without the options set by the client and following the insertion in the cache, apply the desired projection filter.

  This is also relevant for cache hits since the desired behaviour is to have the complete document in the cache and apply each client's desired projection to the resulting document.

- Although it is possible to rename a collection in MongoDB, it is not advisable to do it when using ClearCache since the composition of the Redis keys considers the collection's *Namespace*, composed by both the name of the database and the collection (Section 5.6.2). Therefore, renaming the collection will invalidate all entries in the cache that use its old name.

- Considering the synchronous nature of ClearCache, which depends on the success or failure of operations to update the cache consistently, the write concern level "unacknowledged" must not be used in conjunction with the system, as it does not yet support it. This could lead to situations where the system modifies data stored in the cache based on a write that has not persisted, causing potential data mismatches between MongoDB and Redis. Instead, write concerns that return a response to the client must be used.

5.10 Summary

In this chapter, we have presented the design and implementation phases of the dissertation, including the additional reasoning to enforce data consistency between the database and the cache, as well as the logic flow to access either the database or the cache.

Moreover, we discussed how we extended this data consistency for databases that are deployed in a replica set and demonstrated how different concerns employed by the client affect the execution of their operations. We also present how the ClearCache system supports and leads with the use of transactions.

Attending to the system's limitations, we discussed and presented alternatives that allow to overcome them.

6

Evaluation

In this chapter, we try to answer several key questions. First and foremost, it is essential to determine the prototype's performance to assess how well it meets its intended goals. This should be followed by a comprehensive comparison between the developed system and the established alternative and explore its scalability and reliability to ensure its long-term viability in real-world applications.

Firstly, we introduce the benchmark used to validate the developed system (Section 6.1). Afterwards, we describe the methodology and research approach applied for the validation (Section 6.2) and the environment of the experimental setup (Section 6.3). Finally, we present and discuss the test findings (Section 6.4).

6.1 YCSB

For evaluating the performance of ClearCache, we use Yahoo! Cloud Serving Benchmark (YCSB) [13].

YCSB is a widely used open-source framework designed for evaluating and comparing the performance of various data storage and retrieval systems. It has gained popularity in academic and industry circles as a standardised and practical tool for benchmarking and stress-testing NoSQL and SQL databases, key-value stores, and distributed storage systems [43, 40]. Its primary goal is to provide a common and reproducible benchmarking methodology, enabling users to assess different data stores' throughput, latency and scalability characteristics under similar conditions.

YCSB's versatility enables it to be used for various purposes, such as performance benchmarking of custom workloads, comparative analysis, stress testing, tuning and optimisation. Its ability to simulate real-world workloads and provide meaningful performance metrics contributes to informed database and storage technology decision-making.

YCSB's workloads are organised and executed based on a record's key `_id`, which makes it adequate to test ClearCache and the custom implementations of each method to insert, read, update and delete.

YCSB provides client drivers for different data storage systems. These drivers implement the logic for connecting to the target database and executing the specified operations. The chosen MongoDB driver used for the evaluation was a custom MongoDB client developed and used by MongoDB engineers [44]. To the client, it was added support for write and read concerns and also read preference specifications for the client to use.

6.2 Methodology

The experiments carried out in this evaluation include 2 load phases that populate the database and 6 workloads: the first five are provided by YCSB, and the last one (F) is a workload with customised operation ratios to simulate a more realistic behaviour from an application. A more in-depth description of each workload is presented in Table 6.1.

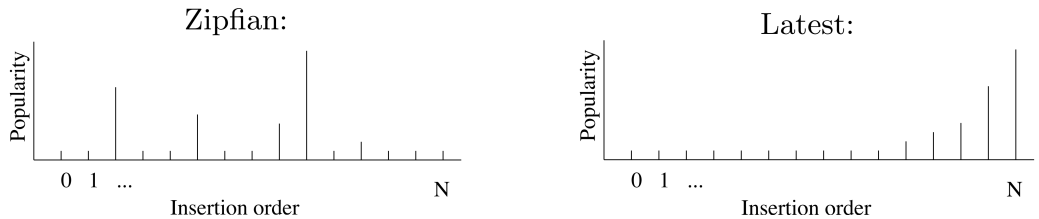
Delete operations were not measured, given that in these types of data stores, deletes are not frequent and, thus, their performance is rarely considered. These data stores are designed and optimised for reads and writes at a large scale.

6.2.1 Distributions

To better understand the workloads used in the testing, we first introduce what distributions are and how they affect the workloads. Distribution refers to which record to choose among the dataset to apply the operation designated by YCSB, simulating the sequencing of these operations in a testing scenario.

Zipfian: Models data access patterns where a few items are heavily accessed while most items are rarely accessed. Used to test database systems under skewed access scenarios. Some records are more popular than others, independently of their insertion order. This distribution is represented in Figure 6.1(a);

Latest: Models the scenario where recently inserted data is more frequently accessed, representing applications where the latest content is the most popular to read. In this distribution, the popularity of a record is more related to its level of recency. The behaviour of this distribution is shown in Figure 6.1(b).



(a) Zipfian Distribution. (b) Latest Distribution.

Figure 6.1: Workload distributions. (Taken from [13])

6.2.2 Workloads

Table 6.1: Workload description.

Workload	Description	Operations ratio	Distribution
Load Phase (Individually inserted)	Write-only	Insert: 100	-
Load Phase (Inserted in batches of 5000)	Write-only	Insert: 100	-
Workload A	Update heavy	Read/Update: 50/50	Zipfian
Workload B	Read mostly	Read/Update: 95/5	Zipfian
Workload C	Read-only	Read: 100	Zipfian
Workload D	Read latest	Read/Insert: 95/5	Latest
Workload E	Read-modify-write[1]	Read/Read-modify-write: 50/50	Zipfian
Workload F	Mixed operations	Read/Update/Insert: 60/34/6	Zipfian

To test the configurations against each workload, we defined the following sequence to run them:

1. Load database;
2. Run workload A;
3. Run workload B;
4. Run workload C;
5. Run workload E;
6. Run workload D;
7. Delete data in the database and in the cache;
8. Reload database;
9. Run workload F.

Before running any workload, a loading phase defines how much data is to be inserted beforehand to populate the database. Given that only workloads D and F insert data into the cache, we execute all other workloads first. Upon running workload D, the database size increased by a number that is not deterministic. Thus, we drop the database and

[1]Operation where a record is read, modified and written back with the new changes

reload its data in order to make a fair comparison of the performance of workload F in the different configurations.

In short, we run the load phase of workload A. After running workload D, we delete the data stored in the database and the cache to keep the database size consistent, and before running workload F, we must reload the data from workload A.

6.2.3 Deployment Architectures

We applied two different database configurations with specific write and read concerns and read preferences to test ClearCache:

- Standalone Database Server:
  - Write Concern: Local;
  - Read Concern: Local;
  - Read Preference: Primary;
- Three-member P-S-S Replica Set Database Server:
  - Write Concern: Majority;
  - Read Concern: Local;
  - Read Preference: Secondary Preferred.

The first configuration consists of a standalone server for the MongoDB database. The clients use the "local" write concern, and for read concern and read preference, clients use the default values. The read concern defaults to "local" level, and the read preference to "primary".

The second experiment includes deploying a replica set with three members, 1 primary and 2 secondaries. Now, the write concern level applied is "majority", and the read concern is the default "local". However, the read preference utilised is "secondary preferred" to simulate the load balance of read requests redirected to secondaries to relieve the primary.

6.3 Environment Setup

6.3.1 Data Stores and Client Configurations

MongoDB and Redis were both deployed within Docker containers, with the following images and versions, respectively:

- MongoDB: mongo 7.0.0
- Redis: redis/redis-stack 7.2.0

It is important to note that the Redis Stack docker image uses version 7.2 of Redis and supports JSON, Search and Query, Time Series, and Probabilistic features. Additionally, it includes RedisInsight, a visualisation tool for understanding and optimising Redis data [1].

Redis was used with the eviction policy "allkeys-lru", where the most recently used keys are kept, and the least recently used (LRU) is removed.

ClearCache and YCSB use the same version of the MongoDB Java Driver. Moreover, ClearCache uses the Redis driver for Java, Jedis, to connect the respective clients to the caching service. The employed driver versions are the following:

- MongoDB: mongodb-driver-sync 4.10.2
- Redis: jedis 4.4.3

6.3.2 YCSB Configurations

Dataset

The number of documents loaded into the database is 1,000,000. The size of each document is the default 1 KB, as it comprises 10 fields with 100 bytes each, plus the key.

For each workload, the number of operations to carry out is also 1,000,000, or until 10 minutes of the workload has passed, whichever came first.

Optimised Thread Count

The default configuration for YCSB uses a single thread. However, this is not a realistic scenario since most applications deal with concurrent requests from multiple clients.

To determine the optimal number of clients for our database configuration, we added threads until we overload the database - when the throughput stops increasing and/or the latency begins to rise.

Finding the balance of latency and throughput is important for fine-tuning MongoDB's performance to prove that the results of the benchmark to the application with ClearCache are superior to the results of the optimised application without ClearCache (i.e. an application that relies solely on MongoDB).

We experimented using 1, 2, 4, 8, 16, 32, 64 and 128 client threads on a single server.

Another important aspect is to make sure the server that hosted the multiple clients was not a bottleneck itself. This was not an issue since the server reported low CPU usage during the executions of the experiments, and thus, we can conclude that the client-server was able to generate the specified load.

Based on the results presented in Figure 6.2, the optimal thread count for the client application is somewhere around 16 threads, given that it was the configuration that led to the most throughput. All tests carried out in Section 6.4 were executed with this number of client threads.

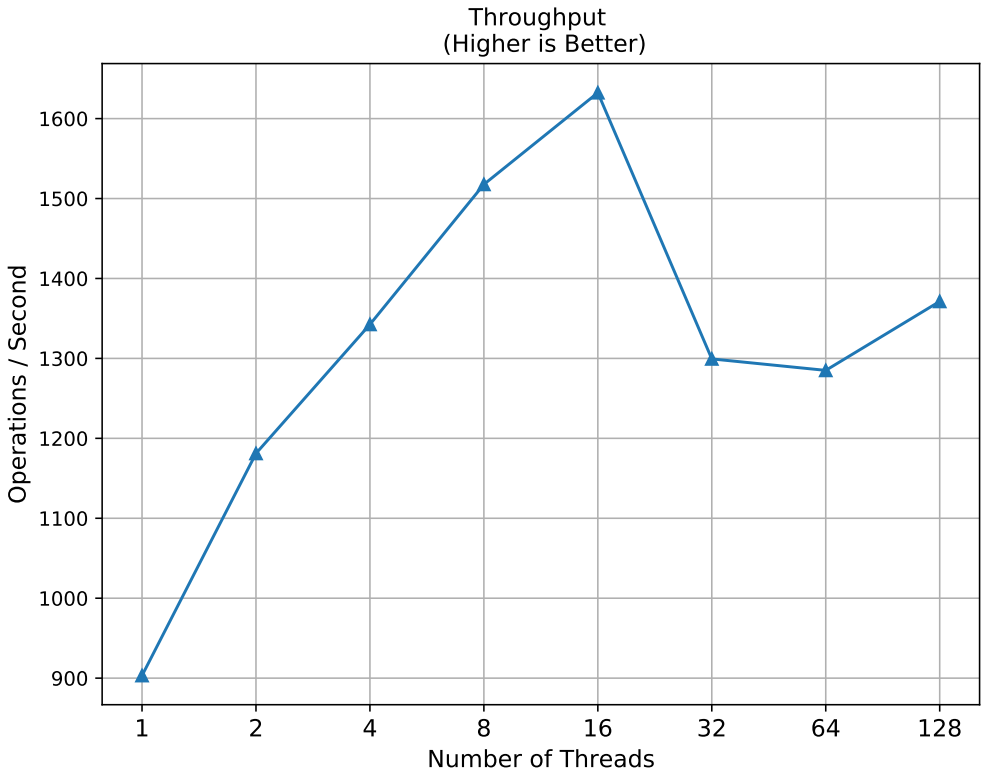


Figure 6.2: Throughput comparison of workload A for varying count of client threads in a standalone deployment.

6.3.3 Hardware Specification

For the experiments without the ClearCache system, an instance of the MongoDB database and an instance of the YCSB client ran on separate machines. For the experiments with ClearCache, there was also a Redis instance on a separate machine. Each replica set member also ran on a dedicated node for the experiments where the MongoDB database is replicated into a three-member replica set.

All experiments were conducted in Grid'5000, specifically at the site in Nancy, where they host several clusters. The *"grisou"* cluster used in the experiments had the following specifications available for each node:

- CPU: Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 2 CPUs/node, 8 cores/CPU, 20MB cache/CPU);
- Memory: 128 GiB;
- Storage: 2x 600 GB HDD SAS Seagate ST600MM0088;
- Network: 10 Gbps;
- OS: Debian GNU/Linux 11 (bullseye) (64 bit).

To conduct the experiments, we aimed to find a hardware setup that would balance the performance of both MongoDB and Redis without one of them gaining a clear advantage

over the other. This is because MongoDB scales better than Redis on multi-core processors due to Redis being a single-threaded server. This means that Redis is not designed to benefit from multiple CPU cores, and, in turn, production environments should launch several Redis instances to scale out on several cores [31].

Therefore, we took this into careful consideration when defining the hardware setup for both systems and ended up limiting the MongoDB and Redis Docker containers to 8 CPU cores and 16 GiB of RAM each.

6.4 Results

In this section, we go through a comprehensive evaluation of the system performance, focusing on critical metrics like throughput and latency. Throughout this analysis, we not only provide a detailed examination of the quantitative outcomes but also offer insights into the justifications and rationales behind the observed patterns for each workload. This diverse assessment aims to prove the system's ability to handle various types of workloads and to explain the implications of these results in real-world scenarios.

YCSB outputs statistics results for the workload running, including runtime, throughput, latency percentiles and latency distribution in time buckets. These results were later rendered to generate the graphs presented throughout this section.

6.4.1 Throughput

Standalone Deployment

Regarding the throughput of the loading phase presented in Figure 6.3, where the documents are individually inserted, we expected a bit more throughput when using ClearCache, given that the procedure for the `insertOne` method does not introduce complex logic. Moreover, with the support of the *ExecutorService* class to asynchronously store the documents into the cache, the throughput should not be far from the provided by MongoDB.

Loading the documents in batches, the throughput with ClearCache is only slightly lower, as expected. The reason for this is the implemented logic of the `insertMany` method that does not deviate from the original implementation, except for the check on the `_id` and the addition of the *Timestamp* field for each document in the batch.

Figure 6.4 shows throughput for each workload for an application with and without ClearCache, using a standalone database server.

Workload A. Workload A is characterised as update-heavy with a 50/50 read/update ratio, meaning that half of the operations are updates. As the results show, the execution of this workload using an application with ClearCache has higher throughput than with an application without ClearCache. That could be because, even with the write operations

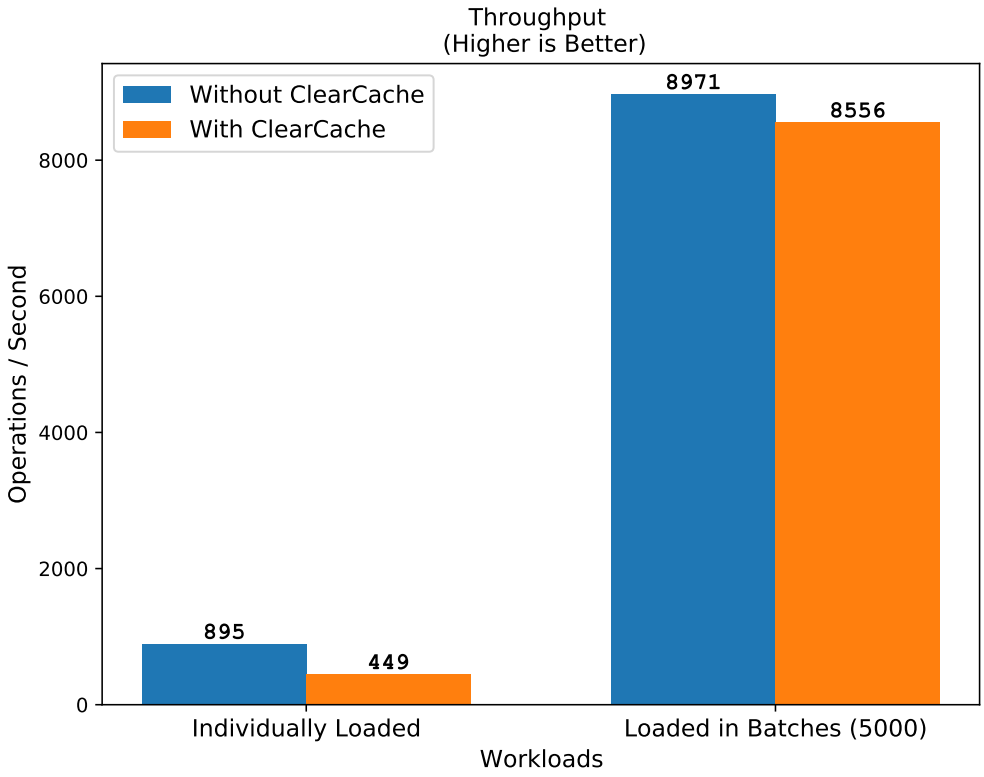


Figure 6.3: Throughput comparison for different loading phase strategies in a standalone deployment.

being heavier when using ClearCache, the read operations are much faster. Thus, the performance of read operations outweighs the performance of the write operations.

In comparison with the other workloads, the throughput of workload A was expected to be lower than that of workloads B and C because it has a higher percentage of updates, which are slower to perform in ClearCache.

Workload B. Workload B is primarily read-focused, with only 5% of the operations updating. For this workload, the throughput of the application with ClearCache is significantly higher than the throughput of the application without ClearCache, as the performance of read operations benefits greatly from having a cache.

This operation ratio results in a significantly lower write load compared to Workload A, and therefore, the throughput for Workload B was expected to be higher.

Workload C. Workload C was expected to have the highest throughput since it does not involve write operations. Among all other workloads, this one produces the biggest throughput difference for an application with and without ClearCache because, once again, read operations greatly benefit from the existence of a cache.

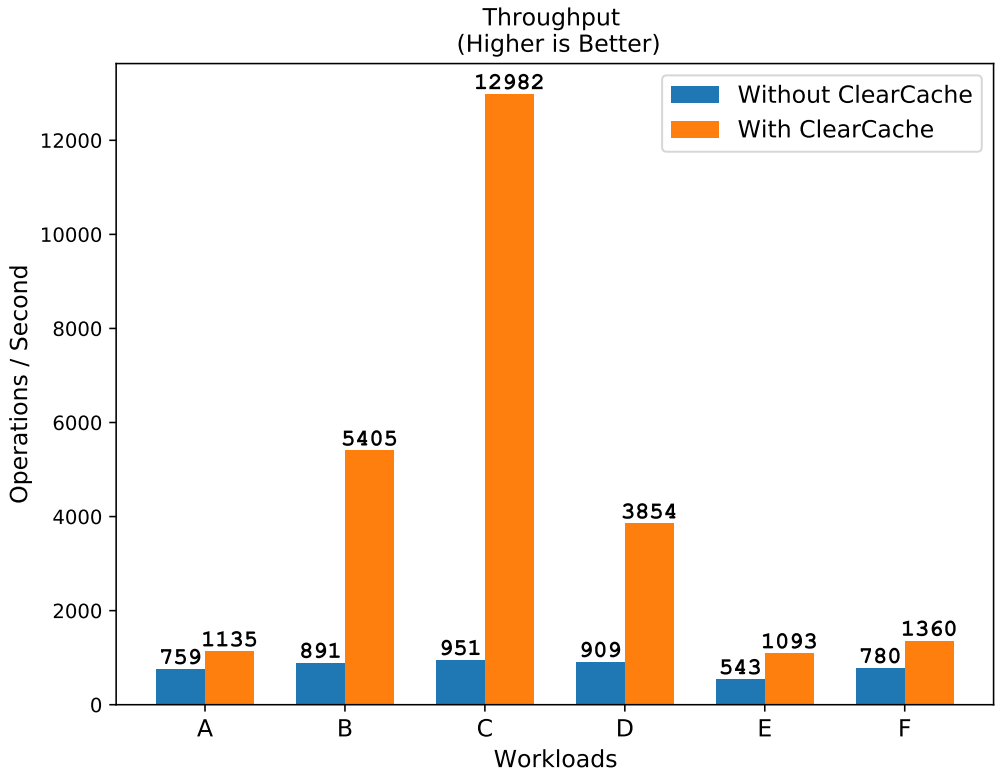


Figure 6.4: Throughput comparison for each workload in a standalone deployment.

Workload D. Workload D is a variant of Workload B that prioritises reading the latest inserted documents. While Workload B consists predominantly of reading with occasional updates following a Zipfian distribution, Workload D focuses exclusively on reading the latest inserted documents.

The throughput for the application with ClearCache decreased while the application without ClearCache achieves almost the same value. Without ClearCache, it appears that an insert operation is heavier than an update. With ClearCache, inserting costs less than updating in the sense that it introduces less additional logic to perform. This is due to, upon inserting, there are no possible conflicts to resolve in the cache, and it is why the following cache insert is made asynchronously. In an update, since the cache may hold a stale version of the updated document, we invalidate it synchronously and only then return it to the client.

However, this asynchronous process may lead to more cache misses upon reading the latest inserted documents, possibly affecting some of the read operations and, thus, the throughput of ClearCache when compared to the achieved through workload B.

Workload E. Workload E is a variant of Workload A, characterised as read-modify-write with a 50/50 read/read-modify-write ratio. Since a read-modify-write operation is heavier

than simply a write operation but includes a reading component where ClearCache can take advantage of the cache to achieve higher throughput when compared to the application without ClearCache.

For both applications, the throughput of workload E was expected to be lower than that of workload A because of the extra cost of a read-modify-write operation, and the results support this hypothesis.

Workload F. Workload F is a custom workload created to combine the operations and respective ratios to mimic a more realistic behaviour from an application. This workload has a majority ratio of read requests, with the remaining 40% dedicated to write operations. The inserts represent only 6% of the total workload ratio, and updates amount to 34%.

Comparing both applications, the fast read operations enable the application with ClearCache to perform this workload with more throughput, even in the presence of insert and update operations.

With the operations ratio in mind, it was expected that both applications would have lower throughput in Workload F when compared to Workload B and D. For the same reason, it was also expected to have higher throughput when compared to Workload A.

Replica Set Deployment

Figure 6.5 presents the throughput for some specific workloads for an application with and without ClearCache, using a P-S-S replica set database server. Recalling that the write and read concern levels used for this deployment were "majority" and "local", respectively, and read preference was "secondary preferred".

For this configuration, we did not execute Workload C and E. The rationale behind this was to highlight the results of the workloads considered to be of more importance, given the deployment type.

For all workloads, there is a significant reduction in throughput. This can be attributed to the network and coordination overhead implied by the system's replication. The use of write concern level "majority" leads to higher latencies to acknowledge this type of operation since data must be propagated to multiple replicas. Furthermore, with the use of read preference "secondary preferred", an overhead regarding query routing may also impact operation response times and throughput.

Load Phase. The results of the throughput for the loading phase decreased, as expected. This reduction in performance is attributed to the inherent overhead introduced by the replication process, which can impact the overall writing efficiency.

Workload A. In the case of Workload A, we were expecting the throughput to be a bit higher with the ClearCache.

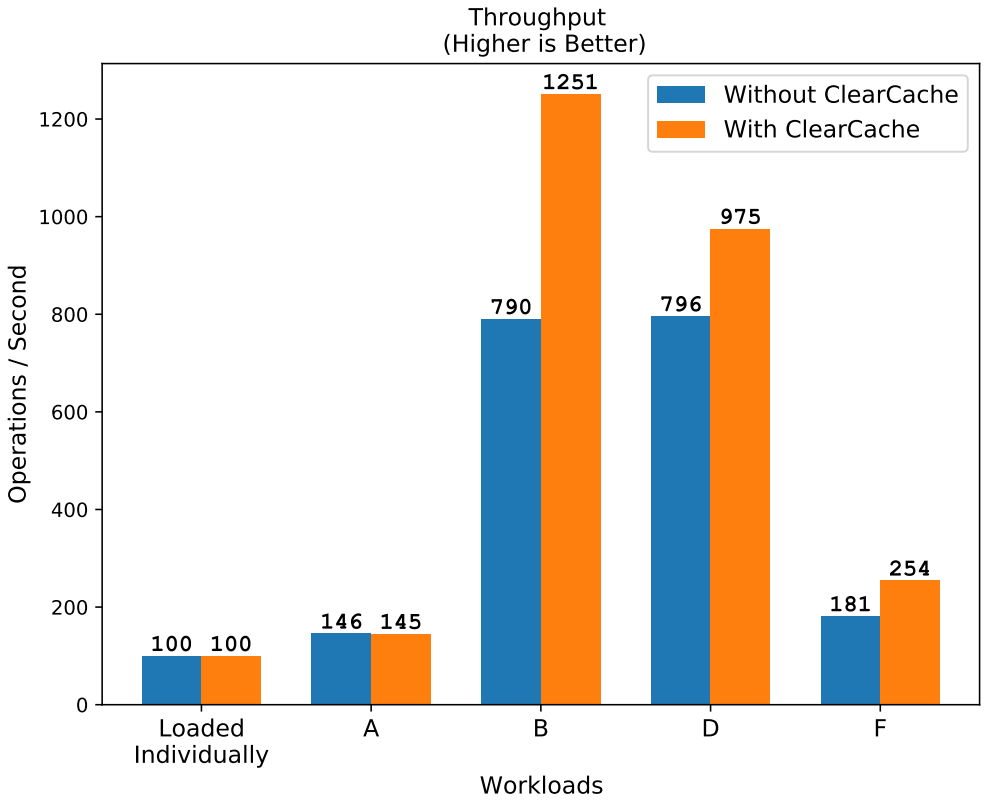


Figure 6.5: Throughput comparison for load and workloads in a replica set deployment.

Nonetheless, the resulting value is justifiable, considering that the advantage that ClearCache had in a standalone deployment of the reads being much faster and outweighing the fact that the writes were slower than MongoDB produced an advantage, which allowed Workload A to execute in ClearCache's favour.

In this configuration, the reads without ClearCache are faster because they are redirected to a secondary, which responds quicker than the primary since it does not execute writes. The advantage gained is not enough to make an impact on the throughput. However, using a bigger dataset may lead to a lower throughput than the one presented for an application without ClearCache due to slower read operations.

Workload B. In this workload, the situation is similar to Workload A, but now the ratio of write and read operations is much steeper, which benefits the fast read operations provided by ClearCache.

Workload D. Workload D was tested mainly for the distribution used in combination with the read concern level "majority" and read preference "secondary preferred". In the case of an application without ClearCache, a replica member where the replication mechanism of a write has not taken effect yet, a following read with concern level "majority"

will return a stale version of the data. In the case of ClearCache, that replica goes to the cache and is able to read from the cache the most up-to-date data. However, in a similar manner to the standalone deployment, in Workload D, cache misses may affect some of the read operations.

Workload F. Workload F also shows that the throughput is higher when executed with an application with ClearCache than without. We also believe that using a bigger dataset may lead to slower reads and an even more noticeable difference in throughputs.

6.4.2 Latency

Since throughput does not always provide a full picture of the system's performance, we plotted the latency distribution for the operations of workload A in each deployment to compare both systems. Through the use of the latency distribution in buckets returned by YCSB, we present the Cumulative Distribution Function (CDF) for an operation's latency and, in essence, show how the latencies are distributed for a given operation.

The CDF provides a way to understand the likelihood of a random variable taking on a value less than or equal to a specific value. The X-axis represents the latency, and the Y-axis represents the cumulative probability, i.e., the proportion of operations that were executed with a latency lower or equal to the corresponding value on the X-axis. At the left end of the X-axis (0 milliseconds), the CDF starts at 0, indicating that none of the operations had a latency less than 0. At the right end of the X-axis (the maximum value), the CDF reaches 1, indicating that all operations had a latency lower than or equal to this value. CDF is especially helpful in identifying various percentiles and in understanding how the operations are spread across different latencies.

In our specific case of latency distribution, we desire a line that, as it moves from left to right on the X-axis, has the quickest and most upward movement on the Y-axis. This means that a larger proportion of operations falls below the corresponding latency on the X-axis.

The displayed plots represent operation latencies from workload A only. Workload A was chosen to be plotted given that it provides the most balanced ratio of read and write operations and because it is the workload in which the performance of ClearCache would possibly not exceed the performance of the application without ClearCache.

Standalone Deployment

To better capture and explain the latency distribution presented in Figure 6.6, we analyse the plot in two specific intervals of time to comprehend the behaviour of the operations. Figure 6.7(a) plots the first interval, which is between 0 and 2 milliseconds. Figure 6.7(b) depicts the second, which is between 85 and 100 milliseconds.

In the standalone deployment, the true performance of MongoDB is visible for most operations, and it is possible to verify that reads and writes behave similarly on MongoDB

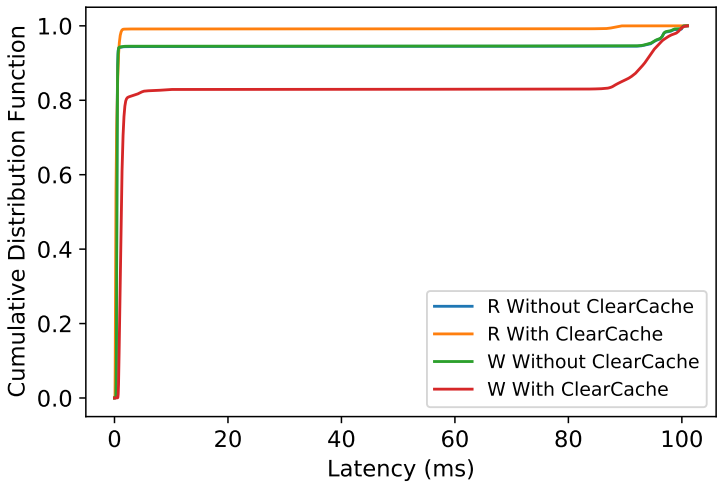
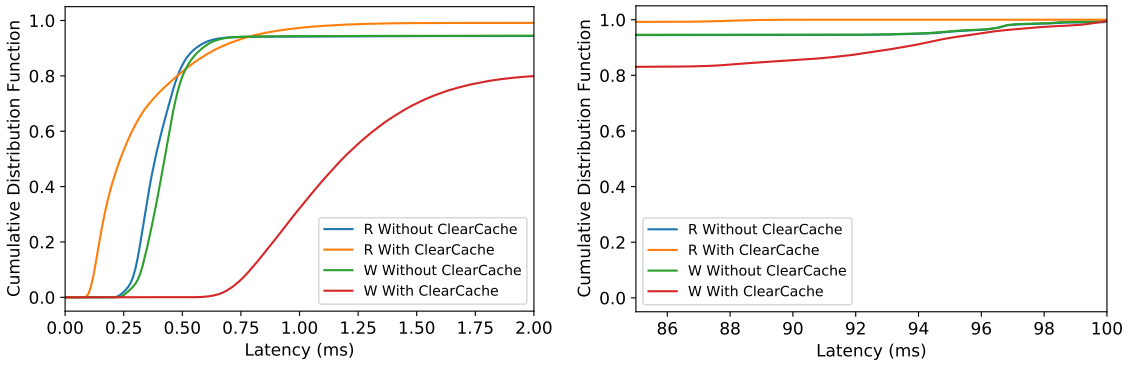


Figure 6.6: CDF of read (R) and write (W) latencies for a standalone deployment in workload A.



(a) Time interval between 0 and 2 milliseconds.

(b) Time interval between 85 and 100 milliseconds.

Figure 6.7: Sub-plot of Figure 6.6 for a specific time interval.

(shown in Figure 6.7(a)). The remaining minority of operations appears to be delayed and is only executed around the 100 milliseconds mark (pictured in Figure 6.7(b)). We suspect that MongoDB became a bottleneck for the applications with the used configuration and specifications.

Regarding the application with ClearCache, the writes have more logic to apply and thus take additional time to execute. The cache introduced by ClearCache helps the reads to be faster than on the database and bypasses the bottleneck problem in the database, executing the read operations more rapidly.

Replica Set Deployment

Figure 6.8 presents the CDF for the P-S-S replica set deployment. It is immediately noticeable the impact of the "majority" write concern level specified for the write operations, where now 50% of the writes take around 40 milliseconds (50th percentile).

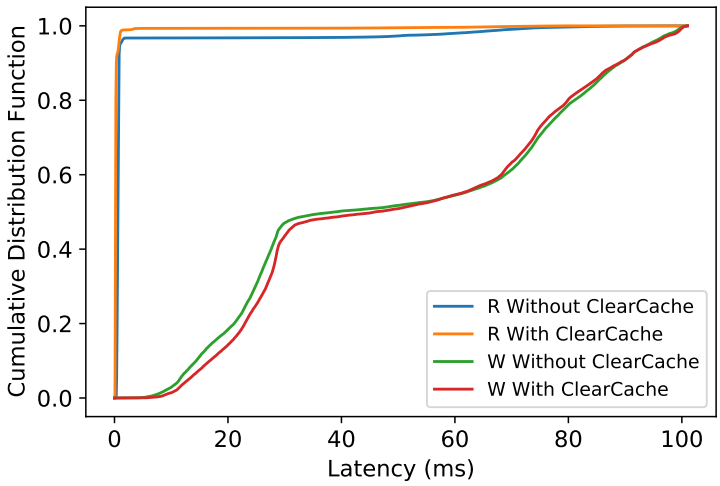


Figure 6.8: CDF of read (R) and write (W) latencies for a replica set deployment in workload A.

For this deployment, Figure 6.9 depicts a closer insight into the latency distribution of the reads. Once again, the read operations are executed faster with the ClearCache system.

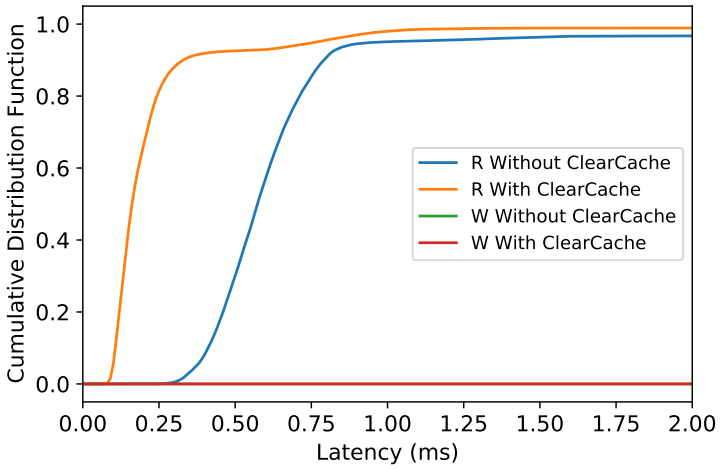


Figure 6.9: Sub-plot of Figure 6.8 for a time interval between 0 and 2 milliseconds.

6.4.3 Read-Heavy Workloads

ClearCache was developed to provide a consistent cache for existing applications.

Thus, to show the cache's impact on the throughput, we decided to generate a graph that relates the write/read percentage of the workload with the associated throughput.

In this series of experiments, write operations ranged from 70% to 0%, while read operations ranged from 30% to 100% of the total workload. The Zipfian distribution and a standalone server configuration of the database were used.

The throughput associated with each experiment is depicted in Figure 6.10. We conclude that for either application, the throughput increases with the read percentage, but the increase is much more significant for an application that uses ClearCache.

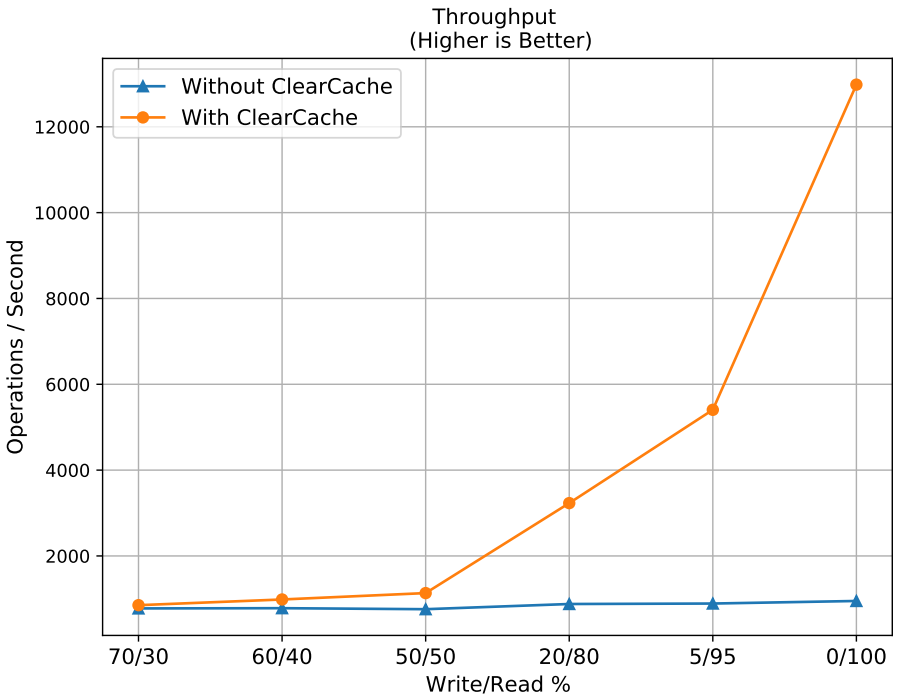


Figure 6.10: Throughput comparison for varying write/read percentages.

6.5 Summary

In this chapter, we have presented multiple benchmarking scenarios to test and compare applications that use ClearCache and applications that do not. We started by characterising the used workloads and then displayed and discussed the obtained results.

7

Conclusion

Application caching entails integrating a caching layer into an existing application to enhance its performance. Caching solutions like Redis are increasingly employed in such applications to store substantial data volumes due to their efficient data retrieval performance.

A caching layer is crucial for application performance, but it poses significant challenges that stem from the concurrent nature of modern applications. One key issue is maintaining data consistency between the cache and the database. Proper synchronisation is required to determine when to update or remove cached data that no longer matches the database state.

This dissertation's primary goal was to simplify the effective use of application caches while maintaining data consistency. It introduces a system design that integrates the database and the cache, providing a single interface for accessing data. The system manages the database and cache transparently to ensure that clients consistently access up-to-date data.

The work involved the development of a versatile library for seamless integration with Java applications utilising MongoDB as their database, ensuring compatibility with the official MongoDB driver API. This integration enabled ClearCache to operate transparently to the client application while introducing essential logic for Redis cache usage.

To maintain data consistency, the solution employed a unique approach based on document IDs and MongoDB timestamps to order operations and their associated documents. Leveraging Redis Functions, ClearCache efficiently managed cached data. The system also supported replica set database deployments, addressing client-specified concerns regarding consistency and availability.

ClearCache further enables transaction support and concurrent cache usage while upholding database-defined isolation and consistency standards. The system's performance was evaluated through testing workloads across various database deployment configurations, including a comparative analysis between applications with and without ClearCache.

7.1 Future Work

In hindsight, the main objectives that were established to develop this system were accomplished. From the transparent nature of the applications that allow for easy integration with existing applications to the verification for data consistency that ensures that the database guarantees were kept even with a supporting caching service. Nonetheless, there is always room for improvement and refinement of what has been developed for systems of this nature.

With this section, we propose other objectives to be accomplished with future work, some of which are for implementation and others for evaluation purposes.

7.1.1 Implementation

For future work regarding implementation, we have established the following objectives:

- Searches by fields other than `_id`: complex queries or at least queries by fields with indexes so that it is possible to ensure that they are unique.
- Implement the remaining write methods offered by the MongoDB interface to offer all available methods in the interface as being compatible with ClearCache to maintain cache consistency. The methods yet to be implemented were `replaceOne`, `findOneAndUpdate`, `findOneAndDelete`, and `findOneAndReplace`.
- Aggregation operations from MongoDB are not yet supported by the ClearCache system. It is possible that utilising the aggregation functionalities offered by the Redis Stack [2] will add support for this type of operation. These operations allow the processing of multiple documents and the return of computed results. Furthermore, since MongoDB offers update methods that take in as arguments an aggregation pipeline, they are also not supported.
- Use of the modules offered by Redis Stack for further benefiting from the data present in the cache. Specifically, RedisJSON [34], which fully supports JSON with specific and atomic methods to efficiently store, manipulate, and query JSON documents directly in Redis, and RediSearch [36] to provide more complex query capabilities.
- Using bulk writes [12] is also not supported in ClearCache. Bulk write operations are designed to efficiently handle multiple write operations in a single request, which can significantly improve performance when dealing with a large volume of data. However, ClearCache depends on synchronously calling and getting the return of each write operation for data consistency guarantees, error handling, and applying its custom logic.
- Allow clients to use concern unacknowledged for write operations by immediately returning a response to the client but making ClearCache execute the necessary

synchronous requests with a stricter write concern in the background to apply the logic in the cache correctly. Inside transactions, unacknowledged operations are not registered and may produce discrepancies regarding "read your writes" inside that transaction.

- Allow for documents written with write concern levels less strict than "majority" to have a lower TTL associated in the cache. In this way, a more durable document may have a bigger TTL, and thus, there is no need for the cache entry to be updated so frequently if no more writes occur.

  If a document was originally written with a lower level but was later read with a "majority" level, then cache TTL is reset, now with a higher value.

- Extend the system beyond standalone and replica set deployments to include sharded clusters. Furthermore, study how other concerns, especially those that are applicable to sharded clusters, affect the consistency guarantees.

7.1.2 Evaluation

For future work concerning evaluation, we have devised the following objectives:

- Benchmark with the systems scaled for their best deployment configuration, i.e. MongoDB with access to all the machine's cores and Redis with several instances launched for the multiple cores.

- Test MongoDB transactions with the ClearCache system. Using the YCSB benchmark made it impossible to test environments with transactions; hence, the functionality implemented by ClearCache to use transactions with the cache was not tested.

  A tool that can help with this testing is an adaption of the benchmark TPC-C for MongoDB. Similar to the YCSB used, this tool was also developed by MongoDB engineers [18].

Bibliography

[1] *About Redis Stack*. URL: https://redis.io/docs/about/about-stack/ (visited on 2023-09-10) (cit. on p. 57).

[2] *Aggregation*. URL: https://redis.io/docs/interact/search-and-query/search/aggregations/ (visited on 2023-09-10) (cit. on p. 69).

[3] *Amazon Cloud Computing Services*. URL: https://aws.amazon.com/ (visited on 2022-11-17) (cit. on p. 16).

[4] *Amazon DynamoDB Accelerator*. URL: https://aws.amazon.com/dynamodb/dax/ (visited on 2023-01-31) (cit. on p. 16).

[5] S. An, Y. Cao, and W. Zhao. "Competitive Consistent Caching for Transactions". In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 2154–2167. DOI: 10.1109/ICDE53745.2022.00207 (cit. on p. 26).

[6] *AWS DynamoDB*. URL: https://aws.amazon.com/dynamodb/ (visited on 2023-01-31) (cit. on p. 16).

[7] *Azure Cosmos DB Documentation*. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/ (visited on 2022-11-03) (cit. on p. 15).

[8] *Azure Cosmos DB Integrated Cache*. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/integrated-cache (visited on 2022-11-03) (cit. on p. 16).

[9] H. Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: Association for Computing Machinery, 1995, pp. 1–10. ISBN: 0897917316. DOI: 10.1145/223784.223785. URL: https://doi.org/10.1145/223784.223785 (cit. on pp. 17, 18).

[10] S. Bouchenak et al. "Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution". In: *Middleware 2006*. Ed. by M. van Steen and M. Henning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–21. ISBN: 978-3-540-68256-1 (cit. on p. 26).

[11] *BSON*. URL: https://bsonspec.org/ (visited on 2023-09-10) (cit. on p. 27).

[12] *Bulk Operations*. URL: https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/crud/write-operations/bulk/ (visited on 2023-09-10) (cit. on p. 69).

[13] B. F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152 (cit. on pp. 53, 54).

[14] *Default Majority Write Concern: Providing Stronger Durability Guarantees "Out of the Box"*. URL: https://www.mongodb.com/blog/post/default-majority-write-concern-providing-stronger-durability-guarantees-out-box (visited on 2023-09-10) (cit. on p. 30).

[15] *Default MongoDB Read Concerns/Write Concerns*. URL: https://www.mongodb.com/docs/manual/reference/mongodb-defaults/ (visited on 2023-09-10) (cit. on pp. 29–31).

[16] S. Ghandeharizadeh and J. Yap. "Gumball: A Race Condition Prevention Technique for Cache Augmented SQL Database Management Systems". In: *Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks*. DBSocial '12. Scottsdale, Arizona: Association for Computing Machinery, 2012, pp. 1–6. ISBN: 9781450314954. DOI: 10.1145/2304536.2304537. URL: https://doi.org/10.1145/2304536.2304537 (cit. on p. 22).

[17] P. Gupta, N. Zeldovich, and S. Madden. "A Trigger-Based Middleware Cache for ORMs". In: *Middleware 2011*. Ed. by F. Kon and A.-M. Kermarrec. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 329–349. ISBN: 978-3-642-25821-3 (cit. on p. 26).

[18] A. Kamsky. "Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB". In: *Proc. VLDB Endow.* 12.12 (2019-08), pp. 2254–2262. ISSN: 2150-8097. DOI: 10.14778/3352063.3352140. URL: https://doi.org/10.14778/3352063.3352140 (cit. on p. 70).

[19] K. Leszczyński Paweł and Stencel. "Consistent Caching of Data Objects in Database Driven Websites". In: *Advances in Databases and Information Systems*. Ed. by B. Catania, M. Ivanović, and B. Thalheim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 363–377. ISBN: 978-3-642-15576-5 (cit. on p. 26).

[20] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[21] *Memcached Homepage*. URL: https://memcached.org/ (visited on 2022-11-21) (cit. on pp. 1, 13).

[22] J. Mertz and I. Nunes. "Automation of application-level caching in a seamless way". In: *Software: Practice and Experience* 48.6 (2018-02), pp. 1218–1237. DOI: 10.1002/spe.2571. URL: https://doi.org/10.1002%2Fspe.2571 (cit. on p. 26).

[23] J. Mertz and I. Nunes. "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches". In: *ACM Comput. Surv.* 50.6 (2017-11). ISSN: 0360-0300. DOI: 10.1145/3145813. URL: https://doi.org/10.1145/3145813 (cit. on pp. 2, 26).

[24] *MongoDB Homepage*. URL: https://www.mongodb.com/ (visited on 2023-01-16) (cit. on p. 27).

[25] *MongoDB Java Driver*. URL: https://www.mongodb.com/docs/drivers/java/sync/current/ (visited on 2023-09-10) (cit. on p. 37).

[26] *Multi-Document ACID Transactions on MongoDB*. URL: https://www.mongodb.com/collateral/mongodb-multi-document-acid-transactions (visited on 2023-09-10) (cit. on p. 49).

[27] R. Nishtala et al. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 385–398 (cit. on p. 24).

[28] H. Ouyang et al. *Verifying Transactional Consistency of MongoDB*. 2022. arXiv: 2111.14946 [cs.DC] (cit. on p. 49).

[29] F. Perez-Sorrosal et al. "Elastic SI-Cache: Consistent and Scalable Caching in Multi-tier Architectures". In: *VLDB* 20 (2011-12), pp. 1–25. DOI: 10.1007/s00778-011-0228-8 (cit. on pp. 20, 22).

[30] D. R. K. Ports et al. "Transactional Consistency and Automatic Management in an Application Data Cache". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 279–292 (cit. on p. 18).

[31] *Redis Benchmark*. URL: https://redis.io/docs/management/optimization/benchmarks/ (visited on 2023-09-10) (cit. on p. 59).

[32] *Redis Functions*. URL: https://redis.io/docs/interact/programmability/functions-intro/ (visited on 2023-09-10) (cit. on p. 47).

[33] *Redis Homepage*. URL: https://redis.io/ (visited on 2022-11-21) (cit. on pp. 1, 14).

[34] *Redis JSON*. URL: https://redis.io/docs/data-types/json/ (visited on 2023-09-10) (cit. on p. 69).

[35] *Redis Scripting and Transactions*. URL: https://redis.io/docs/interact/transactions/#redis-scripting-and-transactions (visited on 2023-09-10) (cit. on p. 47).

[36] *Redis Search and Query*. URL: https://redis.io/docs/interact/search-and-query/ (visited on 2023-09-10) (cit. on p. 69).

[37] *Replication in MongoDB*. URL: https://www.mongodb.com/docs/manual/replication/ (visited on 2023-09-10) (cit. on pp. 28, 32, 33).

[38] W. Schultz, T. Avitabile, and A. Cabral. "Tunable Consistency in MongoDB". In: 12.12 (2019-08), pp. 2071–2081. ISSN: 2150-8097. DOI: 10.14778/3352063.3352125. URL: https://doi.org/10.14778/3352063.3352125 (cit. on p. 28).

[39] *Scripting with Lua*. URL: https://redis.io/docs/interact/programmability/eval-intro/ (visited on 2023-09-10) (cit. on p. 47).

[40] N. B. Seghier and O. Kazar. "Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool". In: *2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI)*. 2021, pp. 1–6. DOI: 10.1109/ICRAMI52622.2021.9585956 (cit. on p. 53).

[41] N. Tolia and M. Satyanarayanan. "Consistency-Preserving Caching of Dynamic Database Content". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: Association for Computing Machinery, 2007, pp. 311–320. ISBN: 9781595936547. DOI: 10.1145/1242572.1242615. URL: https://doi.org/10.1145/1242572.1242615 (cit. on p. 26).

[42] W. Wang et al. "EasyCache: A Transparent in-Memory Data Caching Approach for Internetware". In: INTERNETWARE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 35–44. ISBN: 9781450333030. DOI: 10.1145/2677832.2677837. URL: https://doi.org/10.1145/2677832.2677837 (cit. on p. 26).

[43] *Yahoo! Cloud Serving Benchmark*. URL: https://ycsb.site (visited on 2023-02-08) (cit. on p. 53).

[44] *YCSB Repository Developed by MongoDB*. URL: https://github.com/mongodb-labs/ycsb (visited on 2023-09-10) (cit. on p. 54).