NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DIOGO MIGUEL SANTOS ROSA

BSc in Computer Science

CONSISTENT CACHING FOR APPLICATION SERVERS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

Dec, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CONSISTENT CACHING FOR APPLICATION SERVERS

DIOGO MIGUEL SANTOS ROSA

BSc in Computer Science

Adviser: Nuno Manuel Ribeiro Preguiça
Full Professor, NOVA University Lisbon

Examination Committee

Chairs: Ana Maria Diniz Moreira
Full Professor, FCT-NOVA

Ricardo Manuel Pereira Vilaça
Associate Professor, Uminho university

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Dec, 2024

Consistent Caching for Application servers

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.7) [13].

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, Professor Nuno Preguiça, for his invaluable guidance, patience, and expertise throughout the entirety of this dissertation. His support has been instrumental in shaping this work. I would also like to extend my sincere appreciation to the faculty and staff of the NOVA School of Science and Technology and NOVA University Lisbon for providing the essential resources and facilities that made this research possible. Last but not least I am profoundly thankful to my family and friends for their unwavering support and encouragement throughout these years. Their help and understanding have been a great source of strength and motivation.

This dissertation is the result of work and dedication over the years. I want to express my sincere appreciation to all those who supported me throughout this process, whether through encouragement, mentorship, or insightful discussions. Your contributions have been instrumental in shaping this work, and for that I am thankful for each of you.

”

“The speed of data access is often the limiting factor in the quest for computational efficiency.”

— Donald Knuth, Somewhere in a book or speach
(Astronomer, physicist and engineer)

Abstract

Application caching serves as a widely adopted method for enhancing the performance of web applications and services. Technologies such as Redis or Memcache efficiently store frequently accessed data in memory, lessening the burden of repetitive database queries. However, maintaining data consistency between the cache and the database poses a significant challenge. Concurrent requests accessing the same data may not immediately be reflected updates made to the database in the cache, leading to potential inconsistencies. Resulting in users being presented with outdated information or fresh data being unnecessarily evicted, ultimately impacting performance and user experience.

To tackle this challenge, ClearCache is being designed. Acting as a mediator between the application server and the database, ClearCache ensures a unified and consistent view of the data layer. It operates transparently by storing data and intercepting user requests, effectively managing the read and write operations to a Redis cache instance. Specifically designed as an integrated library for applications utilizing MongoDB as their database, ClearCache leverages timestamps and Redis functionalities to maintain consistency between the database and cache layers seamlessly.

Building on this foundation, this work introduces a mechanism for caching frequently executed queries, specifically those that can be represented as views. By materializing these views and maintaining them in the cache, we aim to reduce database interactions, efficiently handle repetitive query patterns and as a result improve query latencies.

Finally, we provide an experimental evaluation demonstrating that the system enhances application performance by reducing latency and improving overall efficiency.

Keywords: Caching, Application caching, Transparent caching, Consistency, Recurrent queries, *MongoDB*, *Redis*

Resumo

Caching aplicacional é um método amplamente utilizado para melhorar o desempenho de aplicações e serviços web. Tecnologias como o Redis ou Memcache armazenam eficientemente dados frequentemente acessados em memória, aliviando a carga de consultas repetitivas à base de dados. No entanto, manter a consistência dos dados entre a cache e a base de dados representa um desafio significativo. Pedidos concorrentes que acessam os mesmos dados podem não refletir imediatamente as atualizações feitas na base de dados, levando a inconsistências. O que resulta na apresentação de informações desatualizadas aos utilizadores ou no despejo desnecessário de dados, impactando, em última análise, o desempenho e a experiência do utilizador.

Para enfrentar este desafio, ClearCache está a ser desenvolvido. Agindo como um mediador entre o servidor e a base de dados, o ClearCache garante uma visão unificada e consistente da camada de dados. Ele opera de forma transparente armazenando dados e interceptando pedidos de utilizadores, gerindo eficazmente as operações de leitura e escrita para uma instância de cache Redis. Especificamente projetado como uma biblioteca integrada para aplicações que utilizam o MongoDB, o ClearCache aproveita timestamps e funcionalidades do Redis para manter de forma transparente a consistência entre as camadas de banco de dados e cache.

Com base no trabaho já feito, o objetivo é extender o sistema para introduzit um mecanismo de cache para consultas frequentemente executadas, especificamente aquelas que podem ser representadas como views. Ao materializar estas views e as guardar em cache, pretendemos reduzir as interações com a base de dados e assim lidar de forma eficiente com padrões de consultas repetitivos melhorando a latência das consultas como consequência.

Por fim, apresentamos uma avaliação experimental para demonstrar que o sistema melhora o desempenho das aplicações ao reduzir a latência e assim aumentar a eficiência

Palavras-chave: Caching, Application caching, Caching transparente, Consistência, Queries recurrentes, MongoDB, Redis

Contents

List of Figures xi

List of Tables xiii

Glossary xiv

Acronyms xv

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Background and motivation . . . 2
- 1.3 Objectives . . . 2
- 1.4 Expected Contributions . . . 3
- 1.5 Document Organization . . . 3

2 Related Work 4
- 2.1 Databases . . . 4
  - 2.1.1 Transactions . . . 4
  - 2.1.2 Concurrency Control . . . 5
  - 2.1.3 Isolation Levels . . . 6
  - 2.1.4 Phenomena on Isolation Levels . . . 6
- 2.2 Views . . . 7
  - 2.2.1 Concept . . . 7
  - 2.2.2 Materialized Views . . . 8
- 2.3 View Maintenance . . . 8
  - 2.3.1 Concept . . . 8
  - 2.3.2 Recomputation vs Incremental Maintenance . . . 9
  - 2.3.3 Deferred vs Immediate Maintenance . . . 9
  - 2.3.4 Online vs Offline Maintenance . . . 10
- 2.4 Incremental View Maintenance in Databases . . . 10

2.4.1 Algorithms for View Maintenance . . . 12
2.4.2 Incremental View Maintenance in PostgreSQL . . . 13
2.5 Cache . . . 14
2.5.1 What is a cache? . . . 14
2.5.2 Cache Architectures . . . 15
2.5.3 Cache Eviction Policies . . . 15
2.5.4 Memcached . . . 16
2.5.5 Redis . . . 17
2.6 MongoDB . . . 18
2.6.1 Replication . . . 19
2.6.2 Write Concerns . . . 19
2.6.3 Read Concerns . . . 20
2.6.4 MongoDB Pipelines . . . 21
2.6.5 MongoDB Views . . . 22
2.6.6 MongoDB Change Streams . . . 22
2.6.7 MongoDB Sessions . . . 22

3 Design **24**
3.1 ClearCache Base Version . . . 24
3.1.1 Architecture . . . 24
3.1.2 Ordering and Timestamp . . . 26
3.1.3 Supporting Replicated MongoDB in ClearCache . . . 27
3.1.4 Basic Operations Funcionality . . . 28
3.1.5 ClearCache Transactions . . . 30
3.1.6 Handling Varying Client Concerns . . . 31
3.1.7 Considerations and Limitations . . . 33
3.2 Goal and Requirements . . . 33
3.3 Pipeline and Stage Constraints . . . 34
3.4 View Maintenance on Cache . . . 34
3.4.1 Defining Views and Data Structures . . . 34
3.4.2 How to Detect Changes . . . 37
3.4.3 Creating of the Cached Views and View Recomputation . . . 38
3.4.4 View Creation Options . . . 40
3.4.5 Updating the View . . . 41
3.4.6 Handling Stages During View Updates . . . 42
3.4.7 Returning the View . . . 47
3.4.8 Dealing with Varying MongoDB Client Concerns . . . 48
3.4.9 Dealing with MongoDB Sessions and Transactions . . . 49
3.5 Summary . . . 49

4 Implementation **50**

4.1 Creating the View . . . 50
4.1.1 Pipeline Validation and Modification . . . 50
4.1.2 Execute the Create View . . . 51
4.2 Load View Documents into Cache . . . 52
4.3 Client-Side Logic for View Updates . . . 54
4.4 Redis-Side Logic for View Updates . . . 55
4.4.1 Handling old Document and Timestamp Verification . . . 55
4.4.2 Post-processing Validation . . . 58
4.4.3 View Inserts for Grouped Views . . . 59
4.4.4 Delete Logic . . . 61
4.5 Fetching the View . . . 62
4.6 Other Considerations and Limitations . . . 63
4.6.1 Redis Cluster . . . 63
4.6.2 Stage-Specific Limitations . . . 64
4.6.3 Other Considerations . . . 65
4.7 Summary . . . 65

5 Evaluation **66**
5.1 Socialite . . . 66
5.1.1 User Management and Social Graph Operations . . . 66
5.1.2 Timeline Aggregation and Management . . . 67
5.1.3 Operations in Socialite . . . 68
5.1.4 Benchmarking Framework . . . 68
5.1.4.1 Data Loading and Workload Generation . . . 69
5.1.5 Modifications to Socialite . . . 69
5.2 Methodology . . . 69
5.2.1 Workload Configuration and Description . . . 70
5.3 Environment . . . 72
5.3.1 Hardware Specifications . . . 72
5.3.2 Hardware Configuration for Experimental Setup . . . 72
5.3.3 Environment Setup . . . 72
5.3.4 Deployment Architectures . . . 73
5.3.5 Configuration of the Socialite Workload . . . 73
5.4 Results . . . 74
5.4.1 Workload A . . . 75
5.4.2 Workload B . . . 77
5.4.3 Workload C . . . 79
5.4.4 Workload D . . . 80
5.4.5 Workload E . . . 82
5.4.6 Workload F . . . 84
5.4.7 Workloads D, E, and F with Fanout-on-Write . . . 85

5.4.8 Results for Workloads G, H, and I 87
5.4.9 Workloads J, K, and L 88
5.4.9.1 Workload J 89
5.4.9.2 Workload K 90
5.4.9.3 Workload L 91
5.5 Summary and Discussion 92

6 Conclusion **94**
6.1 Future Work 95

Bibliography **96**

List of Figures

2.1 View Maintenance Timing. 10
2.2 Three-member replica set architecture(Taken from [22]) 19

3.1 ClearCache Architecture (Taken from [15]) 25
3.2 Example of Ordering Inconsistency (Taken from [15]) 27
3.3 Original documents without _ts timestamp 28
3.4 ClearCache-modified documents with added _ts timestamp 28
3.5 Comparison of original and ClearCache-modified documents (Source: [15]) 28
3.6 View Cache Structure 36
3.7 Inconsistencies created by intertwining get and set operations 47

5.1 Schema design for the Socialite User Graph Service (From [32]) 67
5.2 Deployment Architecture for ClearCache 73
5.3 Throughput vs. Latency for Workload A 75
5.4 Throughput vs. Latency for Workload A with Six MongoDB Nodes 76
5.5 Latency for Workload A Operations 77
5.6 Throughput vs. Latency for Workload B 78
5.7 Throughput vs. Latency for Workload B with Six MongoDB Nodes 78
5.8 Latency for Workload B Operations 79
5.9 Throughput vs. Latency for Workload C 80
5.10 Throughput vs. Latency for Workload D 81
5.11 Throughput vs. Latency for Workload D with Six MongoDB Nodes 81
5.12 Latency for Workload D Operations 82
5.13 Throughput vs. Latency for Workload E 83
5.14 Throughput vs. Latency for Workload E with Six MongoDB Nodes 83
5.15 Latency for Workload E Operations 84
5.16 Throughput vs. Latency for Workload F 84
5.17 Throughput vs. Latency for Workload D with Fanout on Write 85
5.18 Throughput vs. Latency for Workload E with Fanout on Write 85
5.19 Throughput vs. Latency for Workload F with Fanout on Write 86

5.20 Throughput vs. Latency for Workload F with Fanout on Write and Six MongoDB Nodes 87
5.21 Throughput vs. Latency for Workload G 88
5.22 Throughput vs. Latency for Workload H 88
5.23 Throughput vs. Latency for Workload I 89
5.24 Throughput vs Latency for Workload J in ClearCache 90
5.25 Throughput vs Latency for Workload J in MongoDB 90
5.26 Throughput vs Latency for Workload K in ClearCache 91
5.27 Throughput vs Latency for Workload K in MongoDB 91
5.28 Throughput vs Latency for Workload L in ClearCache 92
5.29 Throughput vs Latency for Workload L in MongoDB 92

List of Tables

2.1 SQL Isolation Levels Defined in terms of the Three Original Phenomena (Taken from [2]). 7

Glossary

ACID ACID is a set of properties of database transactions intended to guarantee validity even in the event of errors or major failures. It stands for Atomicity, Consistency, Isolation, Durability. *(pp. 4, 22)*

Journal A sequential, binary transaction log used to bring the database into a valid state in the event of a hard shutdown. Journaling writes data first to the journal and then to the core data files. *(p. 20)*

RESTful RESTful is an architectural style for designing networked applications. It stands for Representational State Transfer and is based on a set of principles that emphasize simplicity, scalability, and statelessness. RESTful architectures are commonly used in the design of web services, and they leverage the existing HTTP protocol for communication. *(p. 1)*

Acronyms

API Application Programming Interface *(pp. 1, 25)*

BSON Binary JSON *(p. 19)*

CDN Content Delivery Network *(p. 15)*

DBMS Database Management System *(pp. 4, 5, 7)*
DIVM Deferred Incremental View Maintenance *(p. 9)*

I/O Input/Output *(p. 14)*
IIVM Immediate Incremental View Maintenance *(p. 9)*
IVM Incremental View Maintenance *(pp. 9, 10, 12, 13)*

TTL Time To Live *(pp. 16, 27, 28, 40, 48, 54, 72)*

1 Introduction

In this initial chapter, we begin by providing an overview of the context (Section 1.1), background and motivation (Section 1.2) for the undertaken work. Following this, we outline the objectives (Section 1.3) and expected contributions (Section 1.4). Lastly, we present the organization of the subsequent sections in the report (Section 1.5).

1.1 Context

In the contemporary era of web applications and services, the quest for high-speed data retrieval and delivery has become synonymous with the success of digital platforms. Studies consistently show a strong correlation between website reponse time and user engagement [10]. Therefore, optimizing website performance is crucial to enhance user experience.

Many modern applications rely on RESTful services and application servers, like Application Programming Interface (API), that must respond quickly to meet user expectations. However, these applications often deal with large, presenting challenges for optimization. Despite developers' efforts to overcome these limitations, a critical bottleneck remains, primarily due to the speed constraints of accessing databases.

One approach to improving performance is through application caching, which involves adding a caching layer within applications. Application caching is a technique for improving performance that entails adding a caching layer inside applications. Large-scale dataset caching is made possible by the emergence of key players like Redis[29] and Memcached[14]. Caching, which works within a machine's local memory instead of writing to disk like traditional databases do, promises increased efficiency. Because there are no concerns about durability, operations can be streamlined, which does away with the necessity for data replication that exists in databases.

These high-traffic applications often experience repeated requests for the same data, such as retrieving recent posts in a social network or displaying top-selling products on an e-commerce platform. These recurrent queries can place a significant burden on databases, which can lead to more performance bottlenecks and slow response times. As stated in

[16], along with when, choosing what content to cache is a important consideration. Caching recurrent queries may significantly improve application performance by storing the results of these frequently accessed queries in a temporary repository, such as memory or a dedicated caching system. By retrieving cached data instead of re-executing the queries on the database, applications can respond to requests much faster and reduce the load on database. The more opportunities there are to cache recurrent queries, the greater the potential performance benefits.

However, implementing caching in high-traffic applications introduces challenges, notably regarding data consistency[16]. Synchronizing the introduction of new data and the discarding of outdated cache entries becomes a complex task. Many applications struggle with maintaining consistency, either by ignoring the issue, exposing inconsistent states, or excessively invalidating the cache, leading to performance degradation. The concurrent nature of these applications, handling multiple read and write requests, further complicates this problem.

1.2 Background and motivation

ClearCache is a consistent cache system developed at NOVA School of Science and Technology to tackle the challenges associated with data caching in modern applications. It seamlessly integrates with MongoDB-based applications, aiming to enhance overall system performance by efficiently managing cached data while ensuring MongoDB guarantees and consistency across operations.

ClearCache provides essential cache operations, facilitating efficient data retrieval and storage. Leveraging MongoDB's read and write concerns, ClearCache ensures consistency by allowing developers to specify the desired level of consistency for read and write operations. This guarantees that data retrieved from the cache reflects the current state of the underlying database. Furthermore, ClearCache also provides support for transactions, enabling developers to group multiple operations into atomic units of work.

However, there is still room for improvement in ClearCache. One notable limitation of ClearCache is its lack of support for recurrent queries originating from complex operations such as aggregations. Addressing this limitation could enhance ClearCache's suitability for a broader range of applications and use cases.

1.3 Objectives

This work seeks to enhance the **ClearCache** system by integrating support for **recurrent queries**. The core objective is to optimize performance in scenarios characterized by repetitive query patterns, reducing response times and resource consumption. By caching frequently accessed data, the system aims to minimize the repeated execution of identical queries, improving both efficiency and scalability.

Additionally, the system will maintain these query results as **materialized views**, ensuring that cached data remains up-to-date without requiring modifications to the underlying MongoDB database. The goal is to provide a solution that can seamlessly integrate with existing MongoDB deployments, enhancing both system reliability and performance in a **transparent and non-intrusive** manner.

1.4 Expected Contributions

The following summarizes the main contributions that are expected:

- **Designing support for recurrent queries** by caching frequently requested data, the system can greatly diminish the need for repeatedly executing underlying queries. However, this presents the challenge of keeping the views updated and consistent, requiring the development of a mechanism to achieve this.
- **Implementing** the proposed extensions to the ClearCache system.
- **Evaluating support for recurrent queries** involves assessing the performance improvements achieved through recurrent query caching and identifying any potential shortcomings in the implementation phase.

1.5 Document Organization

The remainder of this document is organized as follows:

- **Chapter 2** provides essential background knowledge on the key concepts and prior work relevant to the topics addressed in this thesis.
- **Chapter 3** presents the design of the system or methodology proposed in this thesis. It outlines the architectural decisions, the core components, and the rationale behind key choices.
- **Chapter 4** details the practical implementation of the proposed design.
- **Chapter 5** presents the evaluation of the proposed system, including experiments, performance benchmarks, and analysis of the results.
- **Chapter 6** concludes the report by summarizing the main findings and contributions of this work. It also outlines potential areas for future work, identifying opportunities to extend the research and enhance the proposed system.

2

Related Work

In this chapter we present and discuss related work and background concepts that is relevant for the context and objectives of this thesis. We explore on the following topics: database concepts (Section 2.1), such as transactions and concurrency control, views (Section 2.2), view management (Section 2.3), caching (Section 2.5) and introduce MongoDB mechanisms and concepts that are needed to understand the system (Section 2.6).

2.1 Databases

This section delves into some fundamental database concepts, covering transactions (Subsection 2.1.1), concurrency control (Subsection 2.1.2), isolation levels (Subsection 2.1.3), and phenomena associated with isolation levels (Subsection 2.1.4).

2.1.1 Transactions

Transactions are a fundamental concept in computer science, particularly in the field of Database Management System (DBMS) [12]. A transaction is a logical unit of work that consists of one or more operations. Typically associated with databases, these operations may include inserting, updating, or deleting data. The main challenge is ensuring that all operations within a transaction execute with success, or none of them are. The lifecycle of a transaction typically consists of four phases: *begin*, *execute*, *commit*, and *rollback*. The beginning phase marks the start of the transaction, followed by executing the necessary operations on the database. Once all operations complete, the transaction is committed, marking the completion of a correct transaction and making its changes permanent. However, if an error occurs during any phase of the transaction or if it fails to meet certain conditions, it can be rolled back, reversing any changes made by an unsuccessful or aborted transaction.

ACID Properties

Transactions, normally, adhere to four properties, known as the ACID properties:

- **Atomicity:** Make sure that a transaction is treated as a single unit of work. Either all operations within it are completed successfully, or none of them are.
- **Consistency:** Ensures that the database remains in a valid state before and after the execution of a transaction. Enforces predefined rules and constraints.
- **Isolation:** Guarantees that each transaction is isolated from other concurrent transactions. Prevents interference or conflicts, maintaining data integrity.
- **Durability:** Ensures that once a transaction is committed and written to the database, its changes will persist even in case of system failures. Typically achieved through mechanisms such as write-ahead logging and regular backups.

2.1.2 Concurrency Control

Concurrency control[12] is a set of techniques that ensure that different processes or threads can access and modify shared data without resulting in data inconsistencies or deadlocks. There are two approaches to dealing with concurrent accesses:

- **Pessimistic Concurrency Control:** This approach assumes that conflicts between transactions are likely to occurand blocks data records when a user begins to update. Other users will be unable to update the data until the lock is released.
- **Optimistic Concurrency Control:** This approach assumes that conflicts between transactions are unlikely to occur, so it allows them to execute without locking the data, but it checks for conflicts when they commit. If a conflict is detected, the transaction is aborted and must be restarted.

There are several concurrency control mechanisms, each with its own advantages and disadvantages, some of these mechanisms are:

- **Locking:** Locking is a concurrency control mechanism that prevents conflicts between transactions by locking the accessed data, preventing other transactions from accessing it until the lock is released. There are two types of locks: shared locks and exclusive locks. A shared lock allows a transaction to read data but not modify it, while an exclusive lock allows a transaction to do both. Locking is commonly used in DBMS that employ pessimistic concurrency control.
- **Timestamp:** Timestamp is a concurrency control mechanism that assigns a unique timestamp to each transaction. These timestamps are then used to determine the order in which transactions execute, with older transactions executing before newer transactions. Timestamps are commonly used in DBMS that employ optimistic concurrency control.

- **Two-phase locking:** Two-phase locking is a concurrency control mechanism that uses two phases to prevent conflicts. In the first phase, called the growing phase, transactions acquire locks on the data they need to access. In the second phase, the shrinking phase, transactions release the locks acquired during the growing phase. Two-phase locking is commonly used in DBMS that employ locking mechanisms.
- **Versioning:** Versioning is a concurrency control mechanism that utilizes multiple versions of data to mitigate conflicts. Each transaction perceives a snapshot of the database, and it solely accesses the versions of data that existed at that particular time. Versioning is commonly employed in DBMS that implement optimistic concurrency control.

2.1.3 Isolation Levels

The isolation property of transactions plays a crucial role in maintaining the independence of each transaction from concurrent ones. This property quantifies the extent to which concurrent transactions can impact one another. The **isolation levels**[2, 1] are key benchmarks in this context, delineating different degrees of transaction isolation. These levels include:

1. **Read Uncommitted:** Allows transactions to read uncommitted changes made by other transactions. This level offers high concurrency but may lead to dirty reads where uncommitted changes are visible.
2. **Read Committed:** Ensures that transactions can only read committed changes made by other transactions. Provides a higher level of data consistency but may still result in non-repeatable reads or phantom reads.
3. **Repeatable Read:** Guarantees that once a transaction has read a particular set of data, it will continue to see the same data throughout its execution. Eliminates non-repeatable reads but may still allow phantom reads where new rows are inserted into a result set.
4. **Snapshot Isolation:** Each transaction sees a snapshot of the database as of the beginning of the transaction, this allows for consistent reads without locking, reducing contention but may still allow some anomalies like phantom reads.
5. **Serializable:** Provides the highest degree of transactional consistency by ensuring strict serializability. Prevents all concurrency anomalies such as dirty reads, non-repeatable reads, and phantom reads but reduces concurrency.

2.1.4 Phenomena on Isolation Levels

As demonstrated on Table 2.1, varying isolation levels inside database transactions introduce distinct phenomena that impact data access concurrency and consistency, and

achieving a balance between preserving data integrity and maximizing system performance requires understanding these phenomena. Remarkably, one of the first articles discussing this was *"A Critique of ANSI SQL Isolation Levels"*[2], which shows that these phenomena and the ANSI SQL definitions fail to properly characterize several anomalies, including the standard locking implementations of the levels covered. The mentioned phenomena are:

- **Dirty Read:** Reading uncommitted changes made by other transactions.
- **Non-Repeatable Read:** Reading different values of the same data within a transaction due to changes by other transactions.
- **Phantom Read:** Seeing new rows in a result set that were inserted by other transactions after the initial read.

Table 2.1: SQL Isolation Levels Defined in terms of the Three Original Phenomena (Taken from [2]).

Isolation Level	Dirty Read	Fuzzy Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

2.2 Views

This section explores views, covering their conceptual basis (Subsection 2.3.1). Additionally, it discusses materialized views and their key differences compared to views (Subsection 2.2.2).

2.2.1 Concept

Database views[11], also known as *virtual tables*, played a significant role in the development and evolution of DBMS. A database view is a saved query that generates a virtual table based on its result set, thereby allowing users to interact with a subset of the data within the database. The use of this techinque has become an essential component in modern database systems, providing numerous benefits such as data abstraction, security, and simplified data access.

The advantage of views comes from the **ability to condense complex searches** from a subset of data that is derived from multiple tables or requires complex joins and aggregations and display the results. By defining a view, users can streamline

their interactions with the data and steer clear of repetitive and error-prone queries. Furthermore, by excluding particular columns or rows, Views can be used to limit the availability of sensitive data and keep some information visible solely to those with permission.

For the same reason views enhance the efficiency by pre-computing results or restricting the amount of data retrieved. Significant performance gains may result from this, particularly in scenarios where complex operations are needed frequently.

2.2.2 Materialized Views

Materialized views build upon the concept of regular views by storing the query results in the database. Index structures can then be created on top of the materialized view, enabling faster access to the data. What is particularly advantageous when dealing with large and complex datasets, or executing computationally expensive queries. By storing the results of the query, the database can avoid the overhead of recomputing the view every time it is accessed. In essence, a materialized view functions like a cache - providing a quicker means of accessing the data.

The **use for materialized views**[7] arises when fast access to data is crucial. Is particularly importing when data access is frequent and views can not be constantly recomputed. Materialized views find utility in various applications such as data warehousing, replication servers, mobile systems, and more.

However, materialized views introduce a new challenge: view maintenance, which involves keeping materialized views consistent with the underlying base tables to ensure their accuracy.

2.3 View Maintenance

View maintenance is the process of keeping a view consistent with its underlying base data. There are a number of techniques and algorithms used to achieve this[7, 11]. These can be classified based on three criteria[11]:

2.3.1 Concept

View maintenance is the process of keeping a view consistent with its underlying base data. There are a number of techniques and algorithms used to achieve this[7, 11]. These can be classified based on three criteria[11]:

- Whether the view is **fully recomputed** or **updated incrementally**.
- Whether the view is updated whenever the base data change or not:**immediate** or **deferred** maintenance.
- Whether queries can be executed while the view is being updated or not: **online** or **offline** maintenance.

2.3.2 Recomputation vs Incremental Maintenance

Recomputation involves regenerating all views at set intervals, prioritizing simplicity. While ensuring that the accessed data is always fresh, the constant recomputation can become expensive.

Incremental View Maintenance (IVM) refers to the process of updating materialized views by applying changes that have occurred to the underlying data, rather than recomputing the entire view from scratch. IVM typically involves tracking changes made to the base relations and efficiently propagating these changes to the materialized views.

The choice between recomputation and incremental maintenance depends on data source characteristics and performance requirements, typically within data warehouses. Recomputation is favored for its simplicity and robustness, while incremental maintenance excels in scenarios with frequent source data changes, limited availability, or prohibitions on recomputation downtime.

2.3.3 Deferred vs Immediate Maintenance

View maintenance can also be classified as immediate or deferred[11].Immediate maintenance involves updating materialized views in real-time, while deferred maintenance can occur periodically or upon client query. As illustrated in Figure 2.1, primary approaches include immediate incremental, deferred incremental, and deferred recomputation. Immediate recomputation is not applicable due to high computational demands.

Deferred Incremental View Maintenance (DIVM) records changes to source tables in a log file and applies them in batches to materialized views. As showcased in [5] various methods exist, these normally involve storing the changes or the summarized information to calculate these changes.

Immediate Incremental View Maintenance (IIVM) updates materialized views by applying changes from source tables as soon as they are committed. This contrasts with DIVM, offering benefits such as improved data freshness. IIVM relies on update notifications, containing information about the modified relation and pre-update/post-update tuples, to implement necessary modifications to affected materialized views.

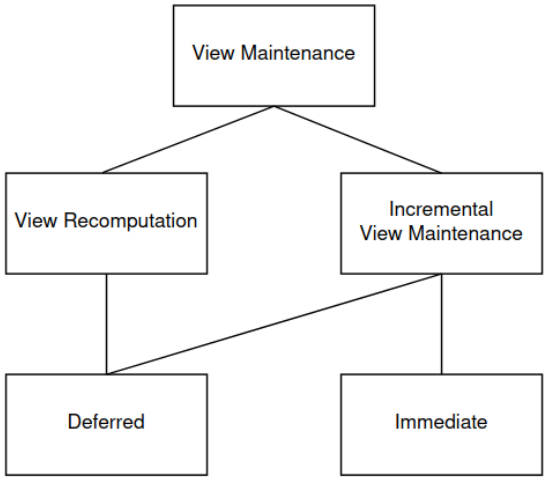

Figure 2.1: View Maintenance Timing.

2.3.4 Online vs Offline Maintenance

View maintenance strategies can also be categorized into offline and online approaches, each offering distinct advantages and trade-offs.

Offline View Maintenance involves updating materialized views without allowing concurrent queries, simplifying maintenance algorithms but potentially delaying user queries. Typically performed during off-peak hours, such as overnight, in data warehousing environments, this approach ensures uninterrupted update processes. However, the downside is the possible delay in query responses due to the need to wait until the materialized views are updated.

Online View Maintenance allows queries to be answered concurrently while materialized views update, ensuring up-to-date information is available to users. In centralized settings, this is often achieved through multi-versioning techniques, while in distributed settings, additional queries are employed to ensure consistency during updates. While offering real-time responses, online maintenance may introduce complexity in maintenance algorithms and require additional resources to manage concurrency effectively.

2.4 Incremental View Maintenance in Databases

IVM is a process used in databases to efficiently update materialized views whenever changes occur in the underlying base tables. Rather than recomputing the entire view from scratch, IVM applies only the changes, deltas, since the last update, which significantly improves performance and reduces computational overhead. The following steps outline how IVM is performed to maintain materialized views incrementally:

1. **Capturing Changes in Base Tables:** The process begins by detecting and capturing changes in the base tables. These changes, referred to as deltas, include insertions, deletions, and updates that modify the data on which the view depends. The database system may use mechanisms like triggers, change data capture (CDC) systems, or materialized view logs to efficiently track these changes. The captured deltas are temporarily stored for further processing during the view update phase.

2. **Determining Affected View Tuples:** After capturing changes in the base tables, the system identifies which parts of the materialized view are impacted. This step involves analyzing the view definition, typically represented as a query, to determine which rows or aggregates in the view correspond to the changed data. For example, if a row is deleted from a base table, the system checks whether that row contributed to any tuple in the materialized view. Similarly, if a new row is inserted, the system checks if it should be added to the view.

3. **Computing Incremental Changes to the View:** Once the affected parts of the view are identified, the system calculates the incremental changes required. This computation is based on the type of operation performed on the base tables:

   - **Insertions:** For each new row inserted into the base tables, the system checks if it should be included in the materialized view and adds it if necessary.
   - **Deletions:** For each row deleted from the base tables, the system verifies whether it exists in the view and removes it if it does.
   - **Updates:** For updated rows, the system determines which parts of the view are affected and computes the necessary changes, such as removing the old value and inserting the updated one.

   The incremental changes are computed using techinques and/or algorithms designed to minimize the number of operations required to update the view.

4. **Applying the Incremental Changes:** The computed incremental changes are then applied to the materialized view. Depending on the database system configuration, the view update may occur immediately or be deferred. The choice between these approaches depends on the requirements for consistency and performance; eager updates provide real-time consistency, while lazy updates reduce the computational load by batching changes.

Maintaining Consistency and Correctness. Ensuring the consistency and correctness of the materialized view throughout the IVM process is critical. Techniques used to achieve this include:

- **Concurrency Control:** Uses locking mechanisms or version control to ensure that multiple changes to base tables do not lead to inconsistencies in the view.

- **Validation Checks:** Periodically verifies the integrity of the materialized view against the base tables to detect any discrepancies caused by unforeseen circumstances.

2.4.1 Algorithms for View Maintenance

IVM is crucial for efficiently updating materialized views in response to changes in the underlying base tables. This process relies on several key algorithms, each tailored to address specific scenarios, some examples are:

Counting-Based Algorithms. Counting-based algorithms, such as the **Counting Algorithm**[8], are designed for non-recursive views. They maintain a count of all possible derivations for each tuple within a view. When a change occurs in the base tables, the algorithm adjusts these counts accordingly. If the count for a tuple drops to zero, it is removed from the view, and if new tuples appear, they are added. This technique ensures efficient updates by limiting the recalculations to only those parts of the view directly affected by the changes.

Delete and Re-Derive Algorithm. The **Delete and Re-Derive Algorithm**[8] is used to handle more complex view maintenance scenarios, particularly for recursive views. The DRed algorithm initially overestimates deletions by assuming that all tuples might be deleted and then selectively re-derives the necessary tuples by determining which ones should remain. This method reduces the need for complete recomputation and is effective for views with multiple levels of dependencies.

Propagation and Filtering Algorithm. The **Propagation/Filtering Algorithm**[9] optimizes the view maintenance process by directly propagating changes from the base tables to the views and filtering out irrelevant updates. This method is beneficial for both recursive and non-recursive views, as it strikes a balance between performance and accuracy by applying only necessary changes while ignoring redundant computations.

In Data Warehousing. While IVM techniques in general database environments focus on maintaining views tightly coupled to the base tables, data warehousing environments require distinct strategies due to their decoupled nature. Algorithms like the **Eager Compensating Algorithm**[36] or **Strobe Algorithm**[35] maintain view consistency through compensating queries. These additional queries are run to update materialized views after changes in the base tables, ensuring that the views remain accurate. They identify changes, such as inserts, updates, or deletes, and apply corresponding adjustments to the views without fully recomputing them. Although these techniques are more niche, they highlight the importance of adapting view maintenance approaches to different database environments.

2.4.2 Incremental View Maintenance in PostgreSQL

PostgreSQL is a powerful and widely-used relational database management system known for its robustness, extensibility, and adherence to standards. One of its performance-enhancing features is *materialized views*, which allow users to store the results of complex queries, enabling faster retrieval of data. This section explains how PostgreSQL implements IVM to efficiently update these materialized views by focusing on only the changes, or *deltas*, rather than recalculating the entire view.

PostgreSQL implements IVM through a combination of **automatic triggers**, **transition tables**, efficient update algorithms, and specialized techniques for handling duplicates or the *DISTINCT* clause. The following components play a crucial role in managing IVM:

Automatic Triggers. When a materialized view is created with support for IVM, PostgreSQL automatically sets up **triggers** on the underlying tables. These triggers act as event-driven mechanisms that are activated whenever changes (inserts, updates, or deletes) occur in the base tables. The triggers capture the details of these changes and initiate the incremental update process for the materialized view.

Transition Tables. Transition tables are temporary tables used by PostgreSQL to store the state of the data before and after a change occurs. These tables hold the "old" and "new" values, which help PostgreSQL determine the exact impact of a change on the materialized view. By leveraging transition tables, PostgreSQL can efficiently compute the *delta* between the old and new states, which is essential for applying the correct incremental updates.

Efficient Update Application. Using the information gathered from triggers and transition tables, PostgreSQL applies the necessary updates to the materialized view incrementally. This involves calculating the rows that need to be **added**, **modified**, or **removed** from the view to reflect the latest state of the underlying tables. This incremental approach minimizes the computational overhead associated with view maintenance, providing substantial performance benefits over a full view refresh.

Handling Tuple Duplicates and the DISTINCT Clause. To support materialized views that include duplicate tuples or the *DISTINCT* clause in their definition, PostgreSQL requires additional considerations. For example, if a view contains two identical tuples and only one needs to be deleted, a simple `DELETE` statement is insufficient, as it would remove both instances.

Similarly, for views defined with the *DISTINCT* clause, PostgreSQL must manage updates carefully when duplicates are present in the base tables. If a tuple is deleted from the base table, a tuple in the view should be removed only when all instances of that tuple are eliminated from the base table. Conversely, when inserting tuples, PostgreSQL

must ensure that duplicates are not reintroduced into the view if a matching tuple already exists.

To handle these cases, PostgreSQL employs a **counting algorithm**. This algorithm tracks the number of duplicate tuples in the materialized view. When a new tuple is inserted, the count is increased if a duplicate already exists; otherwise, a new tuple is added. When a tuple is deleted, the count is decreased, and if the count reaches zero, the tuple is removed from the view. This approach allows PostgreSQL to efficiently manage views with duplicates or *DISTINCT* clauses.

Limitations. While IVM provides significant performance and efficiency benefits by updating only the changed portions of materialized views, there are several limitations to its current implementation in PostgreSQL. One of the main challenges is handling complex queries and view definitions, such as those involving multiple joins, subqueries, or window functions, which can introduce complexities that are difficult to manage with IVM. Additionally, not all SQL constructs are supported by PostgreSQL's IVM implementation, restricting its applicability to a subset of simpler queries.

2.5 Cache

This section provides an overview of caching mechanisms within software systems. We begin by defining what a cache is (Subsection 2.5.1) and proceed to explore various cache architectures (Subsection 2.5.2) and eviction policies (Subsection 2.5.3). Additionally, we delve into two prominent caching solutions, Memcached (Subsection 2.5.4) and Redis (Subsection 2.5.5).

2.5.1 What is a cache?

A cache is a hardware or software component that stores data temporarily in an accessible storage medium that is local to the cache client and distinct from the main storage. It enhances the performance of recently or often accessed data. Operating systems, apps, web browsers, and CPUs frequently utilize caching.

The importance of caching lies in the increase of Input/Output (I/O) operations per second and the decrease in latency and data access times. Since practically all application workloads rely on I/O operations, lowering data access latency is essential for improving efficiency and performance. Caching is used in scenarios such as:

- **CPU Cache:** The CPU cache is a small but fast memory component integrated within the CPU. It stores frequently accessed data and instructions to speed up processing.
- **Web Caching**: Web caching, which includes browser, proxy, and gateway caching, aims to reduce network traffic and latency. Browser caching stores data locally on

users' devices, while proxy and gateway caching store data on intermediary servers to serve multiple users.

- **Database Caching:** Database caching helps improve website or application performance by storing frequently accessed data in local memory. This reduces the need for repeated database queries, resulting in faster response times.
- **CDN:** Content Delivery Network (CDN) is a distributed network of servers strategically placed around the world. It delivers web content to users from the nearest server, reducing latency and improving performance.
- **Application Caching:** Application caching optimizes performance by storing frequently accessed data or computations locally. This reduces the need to retrieve data from the database, improving response times and reducing server load.

2.5.2 Cache Architectures

- **Cache-Aside:** In Cache-Aside architecture, the application code manages the cache directly. The application explicitly loads and stores data to and from the cache. This approach is suitable for scenarios requiring fine-grained control over caching or when integrating with legacy systems.
- **Read-Through:** Read-Through caching automatically fetches data from the main storage if it's not present in the cache when a read operation occurs. This is useful when minimizing latency for read operations is a priority, and the cache can be automatically populated on cache misses.
- **Write-Through:** Write-Through caching involves writing data to both the cache and the underlying storage for write operations. This ensures that the cache and storage remain synchronized. It is suitable for scenarios where maintaining real-time consistency between the cache and storage is critical.
- **Write-Around:** Write-Around caching involves writing data directly to the underlying storage, bypassing the cache. Data is brought into the cache only on subsequent read requests. This is useful for write-intensive scenarios where not all data needs to be cached, and the cache is reserved for frequently read data.
- **Write-Behind or Write-Back:** Write-Behind caching involves initially writing data to the cache and then asynchronously updating the underlying storage. This improves write performance and is appropriate when optimizing for write performance is crucial, and eventual consistency with the underlying storage is acceptable.

2.5.3 Cache Eviction Policies

Eviction Policies Define strategies to efficiently manage cache entries, these policies are crucial for optimizing resource utilization and ensuring effective data retention.

- **Random:** The simplest technique for cache replacement involves randomly removing an entry from the cache.

- **First-In-First-Out (FIFO):** FIFO policy removes the oldest entry from the cache based on the order it was added, making room for new data. Simple to implement but may not always be optimal for dynamic data access patterns.

- **Least Recently Used (LRU):** Requires maintaining a timestamp for each cache entry. The entry with the lowest timestamp, indicating least recent usage, is removed when the cache needs to free up space.

- **Most Recently Used (MRU):** Similar to LRU, but removes the entry with the highest timestamp, indicating most recent usage.

- **Least Frequently Used (LFU):** Utilizes a counter for each entry, representing the number of accesses since the entry was stored in the cache. Removes the entry that has been accessed the least when cache space is required.

- **Most Frequently Used (MFU):** Similar to LFU, but removes the entry that has been most frequently accessed.

- **Least Time To Live (LTTL):** Leverages Time To Live (TTL) for cache entries. Removes the entry with the lowest TTL, which indicates the least remaining time for validity among all entries.

2.5.4 Memcached

Memcached[14] is a widely embraced open-source distributed memory caching system designed to enhance the performance of dynamic web applications by reducing database loads. By operating on a simple key-value pair strategy, Memcached ensures consistently rapid query speeds with an impressive $O(1)$ optimization for all commands, regardless of data size or the number of items stored. A key feature of Memcached is its efficient cache invalidation, eliminating the need for clients to broadcast changes across hosts. Memory management is optimized through a mechanism called *slab*, a fixed-size memory allocation unit tailored for key-value pairs, enhancing efficiency by minimizing overhead associated with small memory chunk allocation. Object categorization into `HOT`, `WARM`, or `COLD` tiers, with automatic transitions based on access history, employs a least-recently-used (LRU) eviction policy. Memcached also offers flexibility with TTL values for specific keys.

Versatile scalability, achieved through vertical and horizontal expansion, sets Memcached apart. Its multithreaded nature allows seamless vertical scaling, while achieving horizontal scaling by adding more servers to the cache cluster. This adaptability makes Memcached well-suited for diverse workloads in dynamic web environments.

2.5.5 Redis

Redis[29], short for Remote Dictionary Server, has firmly established itself as a high-performance, in-memory data store and caching solution. This open-source, key-value store is characterized by a minimalist yet powerful design, addressing diverse use cases with exceptional efficiency. Beyond its role as a caching solution, Redis is also capable of functioning as a database or message broker, further broadening its application scope.

At its core, Redis operates on a straightforward key-value data model, where keys are strings, and values can range from simple strings to complex data structures. Despite this simplicity, Redis's in-memory storage architecture guarantees rapid data retrieval, making it an optimal choice for scenarios that demand low-latency responses.

One of Redis's distinguishing features is its support for complex data types[28]. With atomic operations for setting and retrieving values, manipulating lists, sets, and hashes, managing sorted sets, handling streams, geospatial indexes, bitmaps, bitfields, and HyperLogLog, Redis equips developers with a comprehensive set of tools for efficient data manipulation and storage. Each operation is designed with a focus on simplicity and speed, enabling effortless handling of complex data modeling tasks.

Redis Eviction Policies

Redis supports a few evition policies mentioned in Subsection 2.5.3, including: `No Eviction`, `lru`, `lfu`, `Random`, and `ttl`, with the exception of `No Eviction` and `ttl` all these have variants for both using all keys when considering what keys to evict or just the ones with a expiration time, and if there are no keys to evict matching the prerequisites it will always behave as `No Eviction`.

Redis Cluster

As demand for high availability and horizontal scalability has grown, the need for a distributed implementation of Redis has become increasingly important. This led to the development of **Redis Cluster**[27], which extends Redis's capabilities by enabling distributed, fault-tolerant, and scalable deployments.

Redis Cluster allows data to be sharded and replicated across multiple nodes, ensuring data availability even in the event of node failures. This distributed architecture provides high availability and scalability, which are essential for applications with large data volumes and stringent uptime requirements. However, it also introduces the complexities and trade-offs typical of distributed systems, such as the need for careful management of consistency, partitioning, and failover scenarios.

Redis Functions and Lua Scripting

Redis offers robust mechanisms for data manipulation through two key features: Lua Scripting and Redis Functions[30].

Lua Scripting is a powerful tool within Redis, offering exceptional versatility for executing complex tasks. This embedded scripting language provides a quick and flexible means to implement intricate data validation, transformation, and aggregation logic directly within the Redis environment. Lua scripts enable the creation of secure and reliable transactions with clearly defined rollback points, ensuring data consistency across multiple processes. These scripts can encapsulate business logic, thereby reducing the load on the application side while performing operations atomically and efficiently.

With the introduction of Redis version 7, Redis Functions were introduced as named, reusable modules that are tightly integrated with the data. These functions inherit the advantages of Lua Scripting while offering additional benefits, such as enhanced persistence, improved modularity, and guaranteed cluster consistency. Redis Functions ensure that operations remain consistent and predictable across a Redis Cluster, providing a seamless way to execute complex operations with efficient and intricate data integration.

Redis Data Structures

Redis supports a variety of data structures, each tailored for specific use cases, enabling developers to model their data in ways that best suit the application's requirements. Below is an overview of the most commonly used Redis data structures:

- **Strings:** The fundamental Redis data type, representing text or binary data.
- **Lists:** An ordered collection of strings, akin to a linked list. Lists are used for implementing queues, stacks, and other ordered sequences.
- **Sets:** An unordered collection of unique strings, suitable for membership testing, deduplication, and other set-based operations.
- **Sorted Sets:** Similar to sets, but with an associated score for each member, allowing the set to be ordered. Sorted sets are useful for creating leaderboards, ranking systems, or any scenario where an ordered collection is needed.
- **Hashes:** A collection of key-value pairs, similar to a dictionary or hash map. Hashes are ideal for storing objects or records.

In addition to these data structures, Redis also supports other more niche data structures and functionalities such as Streams, Bitmaps, HyperLogLog (for approximate cardinality estimation), and Geospatial Indexes (for handling geospatial data).

2.6 MongoDB

MongoDB[19], a popular open-source NoSQL database, is tailored to efficiently manage large volumes of data with flexibility and scalability. Unlike traditional relational databases, MongoDB stores data in a document-oriented format, using a flexible, JSON-like structure

called Binary JSON (BSON)[4], a binary-encoded serialization. This allows for easy representation and storage of complex data types.

One of MongoDB's key features is its ability to scale horizontally by distributing data across multiple servers. This sharding capability enables MongoDB to handle massive amounts of data and traffic by adding more servers to the database cluster. Additionally, the support for automatic sharding makes it easier to manage and scale as your data grows. Another notable aspect of MongoDB is its dynamic schema. Unlike traditional relational databases that require a predefined schema, MongoDB allows for dynamic and on-the-fly schema changes. This flexibility is particularly beneficial in scenarios where data structures are subject to frequent modifications.

2.6.1 Replication

MongoDB replication[22] provides high availability, fault tolerance, and read scalability by maintaining multiple copies of data across servers. In a replica set (see Figure 2.2), there is one primary node and multiple secondaries. The primary node receives write operations, and changes are replicated to the secondaries. In the event of the primary failing, a secondary can become the new primary. This replication process is asynchronous, which may result in 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 particularly applicable for replica sets to control consistency and availability properties in replicated environments.

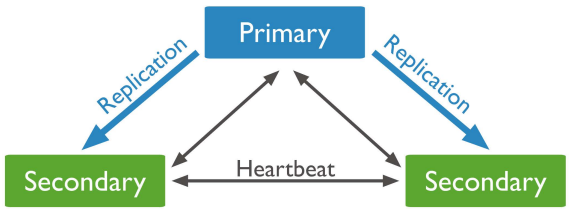

Figure 2.2: Three-member replica set architecture(Taken from [22])

In certain scenarios, you might opt to add a MongoDB instance as an arbitrator to a replica set, even though it doesn't store data. Despite lacking data redundancy, an arbiter participates in elections. MongoDB offers write and read concerns tailored for replica sets, providing greater control over consistency and availability. By effectively utilizing both concerns, you can adjust the balance between these attributes as required. For instance, you may reduce consistency requirements to enhance availability, or vice versa[31].

2.6.2 Write Concerns

Within MongoDB, write concerns[25] dictate the level of acknowledgment required for write operations, influencing the durability and performance characteristics of the write

operations.

Supported write concern levels include:

- **Unacknowledged:** With an unacknowledged write concern, MongoDB does not acknowledge the receipt of the write operation. This means the client does not wait for any acknowledgment. Although errors are ignored, drivers attempt to handle network errors when possible.

- **Acknowledged:** Acknowledged write concern ensures that the mongod confirms the receipt of the write operation. Clients wait for acknowledgment of success or exception. MongoDB uses acknowledged write concern by default (after the releases outlined in Default Write Concern Change).

- **Journaled:** With journaled write concern, the mongod acknowledges the write operation only after committing the data to the Journal. This ensures data recovery following a shutdown or power interruption.

- **Replica Acknowledged:** Replica sets introduce replica acknowledged write concerns, guaranteeing that the write operation propagates to the members of a replica set.

Journaling Option. The boolean journal option in MongoDB specifies whether the database 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 feature involves write-ahead logging to on-disk journal files, providing additional durability in case of power or hardware failures, albeit at the expense of slower write operations. MongoDB strongly recommends enabling this feature when running in production, particularly for applications that necessitate stronger durability guarantees. This is because, even with majority writing, a write can be rolled back in the event of a transient loss (such as a crash and restart) of a majority of nodes if the journaling option is set to false.

The **Default Write Concern Level**, since MongoDB 5.0, is **majority**, except in some cases involving arbiters [18].

2.6.3 Read Concerns

MongoDB introduces the concept of read concerns[21], offering developers fine-grained control over the consistency and isolation of read operations.

Supported read concern levels include:

- **Local:** This returns the most recent data available from the node but acknowledges the possibility of rollback. It is a suitable option when the most recent data is essential, even if it might not be durable in the long run.

- **Available:**[1] This read concern level provides data from the most recent data available from the node, acknowledging the possibility of rollback. It is a suitable option when the most recent data is essential, even if it might not be durable in the long run.
- **Majority:** This read concern guarantees that read operations have accessed majority-committed data. This level of consistency is vital for scenarios where a synchronized view of majority-committed data is necessary.
- **Snapshot:** This read concern level provides data from a snapshot of majority-committed data if the transaction commits with write concern **Majority** It ensures a synchronized snapshot across shards in sharded clusters, offering a high level of data consistency.
- **Linearizable:**[1] This read concern guarantees that read operations return the most recent data available, acknowledging the possibility of rollback. It is a suitable option when the most recent data is essential, even if it might not be durable in the long run.

The **Default Read Concern Level** is **Local** and as previously stated, it can return data that may be rolled back.

2.6.4 MongoDB Pipelines

MongoDB pipelines are a powerful tool for data processing and transformation within MongoDB. A pipeline consists of a series of stages, each performing a specific operation on the documents that pass through it. These stages can include operations such as filtering, grouping, sorting, and aggregating data. By chaining multiple stages together, developers can create complex data transformations that allow for detailed manipulation and refinement of data. This modular approach facilitates precise control over the output and simplifies testing and debugging of data processing workflows.

Concept of Pipeline Stages. Pipeline stages are essentially individual processing steps that handle different aspects of data transformation. They are executed sequentially, with each stage receiving the output of the previous one. This modular approach allows for complex data operations to be performed efficiently.

Pipeline Stages. Here is an overview of several key pipeline stages commonly used in MongoDB:

- **`$project`:** This stage reshapes each document in the pipeline. It can include or exclude fields, rename fields, or add new computed fields. This is useful for preparing data for subsequent stages by focusing on the necessary fields.

[1] This concern is unavailable for use with causally consistent sessions and transactions.

- **`$match`:** The `$match` stage filters documents based on specified criteria. It works similarly to the `find` operation and is often used early in the pipeline to reduce the number of documents processed in later stages.
- **`$limit`:** This stage restricts the number of documents passing through the pipeline. It is useful for controlling the size of the result set, particularly in scenarios requiring pagination or limiting results.
- **`$skip`:** The `$skip` stage skips over a specified number of documents. It is often used in combination with `$limit` to implement pagination.
- **`$group`:** This stage groups documents by a specified field and performs aggregate calculations, such as summing or counting values. It is essential for generating summary data and aggregations.
- **`$sort`:** The `$sort` stage orders the documents based on one or more fields. It can sort data in ascending or descending order, allowing for organized results.

2.6.5 MongoDB Views

MongoDB version 4.2 introduced a views feature[24]. This mechanism allows developers to construct virtual collections based on the output of an aggregation pipeline, Subsection 2.6.4. These views can be used to create read-only views dynamically from others, eliminating the need to modify the original data for conversion, filtering, and aggregation. You can query views like regular collections, inherit security attributes, and support composition with additional aggregation stages for complex transformations. In version 4.4, MongoDB introduced materialized views[20], known as **On-demand materialized views**. This new feature offers a way to store pre-computed results on disk, leading to faster query performance for complex operations. Unlike standard views, materialized views require manual updates.

2.6.6 MongoDB Change Streams

MongoDB Change Streams[17] enable real-time monitoring and response to changes in a MongoDB database. Leveraging MongoDB's replication system, Change Streams provides a continuous flow of events, including inserts, updates, and deletes. Applications can subscribe to these streams to receive updates as they happen, eliminating the need for frequent polling. Each change event contains details such as the operation type, the affected document, and the timestamp of the change, making it an ideal solution for scenarios requiring immediate reaction to database modifications.

2.6.7 MongoDB Sessions

MongoDB Sessions[23] allow a series of operations to be grouped into a single logical unit, which is crucial for managing transactions and ensuring ACID compliance. By

supporting multi-document transactions, sessions help maintain data consistency and integrity, ensuring that either all operations succeed or none are applied. This capability is essential in scenarios requiring atomic updates, such as financial transactions or inventory management, where partial updates can result in data corruption. A unique session identifier tracks the transaction state across these operations, enabling MongoDB to handle distributed transactions across replica sets or sharded clusters and to automatically retry operations in the case of transient errors, thereby improving robustness.

3

Design

This chapter outlines the design of the ClearCache system. It starts with a description of the base version of ClearCache, covering its architecture, handling of operations, and support for a replicated MongoDB environment (Section 3.1). Various aspects such as transaction management, timestamp ordering, and dealing with client-specific requirements are discussed to establish a foundational understanding.

The chapter then defines the goals and requirements for the extended ClearCache implementation (Section 3.2). It addresses the design of the pipeline and stage constraints necessary (Section 3.3) for view maintenance on the cache, elaborating on how views are defined, created, updated, and recomputed (Section 3.4). Special considerations, such as handling MongoDB sessions, client concerns, and maintaining consistency during updates, are also explored. Finally, the chapter concludes with a summary of key design decisions and their implications for system functionality.

3.1 ClearCache Base Version

In this section we delve into the design decisions, architecture and mechanisms used and described in the version of ClearCache that was used as the basis of our work[6, 15].

3.1.1 Architecture

As noted above, ClearCache is designed to be a transparent caching layer that seamlessly integrates with MongoDB applications. However, it's crucial to understand that its design diverges from conventional single MongoDB deployments. The implemented alterations are necessary to accomodate the consistent data management between the cache and database layers and follow the **Cache-Aside** architecture. These modifications effectively align the architecture with the representation depicted in Figure 3.1.

Database

The underlying database used by ClearCache is MongoDB (Section 2.6), which is chosen deliberately because of its many advantages. Its NoSQL architecture, for example, offers

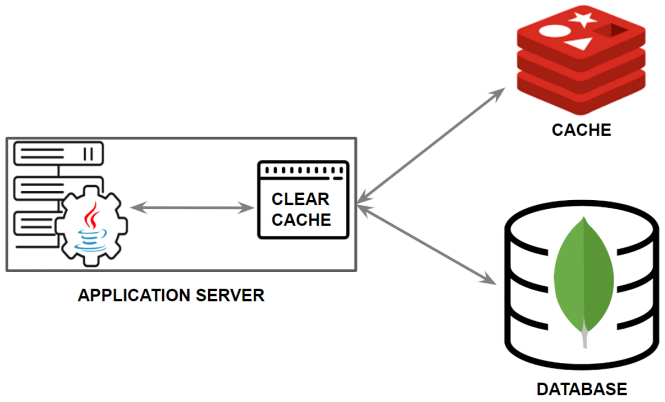

Figure 3.1: ClearCache Architecture (Taken from [15])

the flexibility required to support changing data structures without being constrained by a strict schema. Another reason is MongoDB's performance prowess, especially in handling large and diverse datasets, and the ability to scale horizontally ensures responsiveness even under increasing workloads, a crucial aspect for meeting the demands of high-traffic web applications.

Application Programming Interface

ClearCache's core design philosophy revolves around maintaining the same API that client applications use to communicate with MongoDB. This approach simplifies integration and ensures that developers can continue using the same syntax without any code modifications to their application layer. By matching the official MongoDB driver API, ClearCache effectively intercepts client requests and routes them to either the cache or the underlying database, a decision made dynamically based on the consistency model employed by the system.

Fundamentally, ClearCache seamlessly integrates caching capabilities into MongoDB applications without disrupting the existing development workflow or requiring significant code modifications.

Cache

ClearCache integrates the Redis (Subsection 2.5.5) caching system due to its feature set, outshining alternative systems such as Memcached (Subsection 2.5.4). Memcached exclusively supports simple key-value pairs of unstructured binary data, whereas Redis surpasses this limitation by accommodating a variety of data structures. This versatility allows Redis to offer advanced functionalities such as leaderboards or lists of recent topics. Additionally, Redis provides a more extensive range of eviction policies, with eight alternatives, while Memcached solely relies on the Least Recently Used (LRU) policy.

Redis is renowned for its exceptional speed, often surpassing Memcached in various scenarios, and has scalability support built right in, which makes the process easier.

Memcached, on the other hand, requires extra libraries or tools to provide scalable solutions.

Additionally, the ability to perform complex actions on stored data in the form of **Redis Functions and Lua Scripting** is a significant advantage. While Memcached has some advantages, particularly with large datasets, Redis stands out as the most feature-rich choice, providing a comprehensive set of capabilities for advanced data manipulation and efficient operations.

3.1.2 Ordering and Timestamp

Ensuring data consistency across multiple layers, such as a database and a cache, becomes progressively more complex in concurrent environments. While a single server scenario simplifies data synchronization—by caching data upon insertion into the database and removing it upon deletion—this approach is unrealistic for systems with numerous clients and/or multiple application servers. In such cases, ensuring data consistency across servers is crucial but problematic.

The Problem of Data Inconsistency

When multiple application servers handle write requests for the same data, the lack of coordination between servers can lead to inconsistencies between the data layers. Consider a scenario where two application servers receive write requests for the same variable. Each server will attempt to write the data to the database and then update the cache. However, if one server's request to the database is delayed, the second server might complete both its database and cache writes before the first server updates the cache. In Figure 3.2, we can see how this sequence of events results in the cache holding outdated data, causing a mismatch between the database and the cache.

Timestamp-Based Solution

To address this issue, a timestamp is integrated into each document. This timestamp reflects the document's creation or last update time, enabling better management of cached data. The timestamp ensures that the most recent data is readily accessible. When a document is updated, its timestamp changes accordingly. The system, termed *ClearCache*, uses this timestamp to determine whether to write to the cache. Specifically, ClearCache writes to the cache only if the new document has a more recent timestamp than the currently cached version, thereby minimizing the risk of serving stale or inaccurate data.

Implementation Details: Choosing the Timestamp Type

The timestamp is stored in a field named `_ts`, which the system automatically updates to reflect when the document was inserted or last modified. Two data types were considered for this field: `Date` and `Timestamp`.

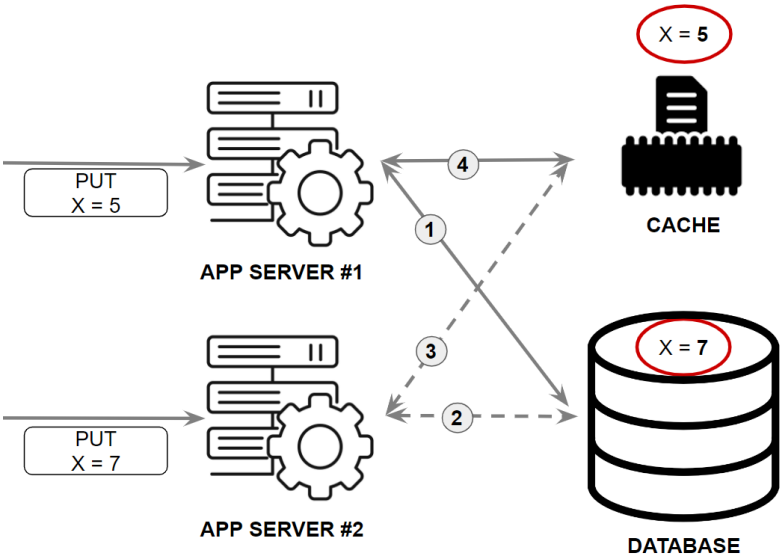

Figure 3.2: Example of Ordering Inconsistency (Taken from [15])

In distributed systems, one of the main challenges is that clients may have inconsistent system clocks, which can lead to incorrect sequencing of operations if timestamps are generated on the client side. Therefore, timestamps must be generated by the database server instead. Although MongoDB generally recommends using the `Date` type for most scenarios, it requires converting the insert operation into an upsert and using the `currentDate` update operator to ensure the timestamp accurately reflects the database server's time. This method, however, adds extra database queries and processing overhead, which can negatively impact performance.

The `Timestamp` type was chosen because it allows the timestamp to be added directly during the insert operation, ensuring faster execution and proper error handling. The `Timestamp` object has two components: `t`, which represents the number of seconds since the Unix epoch, and `i`, which orders operations occurring within the same second. Since timestamps are generated on the server, they inherently order operations correctly, avoiding ambiguity when handling concurrent write operations.

3.1.3 Supporting Replicated MongoDB in ClearCache

Maintaining data consistency in a replicated MongoDB setup involves addressing two primary challenges, data rollbacks and handling varying client concerns. These arise by the fact that when accessing a replicated MongoDB, it is possible to use different consistency levels expressed by read and write concerns, as explained in Subsection 2.6.2 and Subsection 2.6.3. ClearCache addresses these issues by introducing a TTL mechanism for cached documents and implementing a strategy to manage varying client concerns.

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

Figure 3.3: Original documents without _ts timestamp

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

Figure 3.4: ClearCache-modified documents with added _ts timestamp

Figure 3.5: Comparison of original and ClearCache-modified documents (Source: [15])

Data Rollback

When MongoDB confirms a write operation, there is no guarantee that the written data will be permanently committed. Due to different write concern levels, a write operation acknowledged by MongoDB might still be subject to rollbacks under certain conditions, such as network partitioning or replica set member failures. This behavior can lead to discrepancies between the database and the cache, as the cache might store an outdated or invalid version after a rollback.

To mitigate this, ClearCache introduces a TTL mechanism for all cached documents. By assigning a TTL to each cached entry, the system ensures that cached data is periodically refreshed or removed based on age. This strategy helps reduce the risk of stale or inconsistent data lingering in the cache, as rolled-back changes are naturally corrected over time. If a document in MongoDB is rolled back, the corresponding cache entry will eventually expire and be replaced by the updated data from the database, or discarded. Consequently, this mechanism helps maintain consistency between MongoDB and the cache, even in the presence of rollbacks or other transient inconsistencies.

3.1.4 Basic Operations Funcionality

In this section we will detail the basic operations that were modified by the ClearCache system to maintain data consistency between the cache and the database. The operations are divided into two categories: write operations and read operations.

Write Operations

In the ClearCache system, the write operations are designed to maintain consistency between the database and the cache. This involves ensuring that data inserted into the database is correctly reflected in the cache, particularly considering that timestamps are added by the server and must be included in the cached version.

Insert Methods. The *insertOne* method adds a single document to the database. Once the document is inserted and the operation confirmed, it is added to the cache. To maintain data consistency, a timestamp is attached to the document during this process. The wrapper then performs a `find` query using the document's `_id` to retrieve the final version, now containing the server-generated timestamp. This approach involves two database operations—one for insertion and one for retrieval—but it ensures that the cached document is always the most up-to-date version. Although not the most efficient solution, it is necessary because the *insertOne* operation cannot combine insertion and timestamp retrieval in a single step. This timestamp is critical for comparisons on the Redis side to guarantee that only the most recent version is stored.

To improve efficiency, the cache update is performed asynchronously, allowing the application to continue handling other tasks without blocking the client while the cache is updated in the background.

In contrast, the *insertMany* method improves efficiency by sending multiple documents to the database in a single batch, thereby reducing communication overhead. Due to the bulk nature of this operation, caching each document individually would be inefficient. As a result, the *insertMany* method does not update the cache with newly added documents. This omission is not a significant issue since any missing documents in the cache will result in a cache miss, triggering a retrieval of the document from the database when needed.

Update Methods. The *updateOne* method is responsible for updating a single document in the database. It uses `findOneAndUpdate`, which allows it to return the updated document. This returned document includes the timestamp of the modification, which is then used to update the cache. In the cache, the updated document is compared to the existing version based on timestamps, and only the document with the most recent timestamp is retained.

The *updateMany* method updates multiple documents by iterating over the list of documents to be updated, calling *updateOne* for each. This approach ensures that the cache is synchronized with the database after each document update, maintaining consistency across both systems.

Delete Methods. The *deleteOne* method removes a single document from the database using `findOneAndDelete`, which retrieves the document before deletion. The retrieved document, along with its timestamp, is then used to remove the document from the cache. The timestamp helps ensure that any subsequent operations with an older timestamp are ignored, maintaining cache consistency.

Similarly, the *deleteMany* method uses a loop to call *deleteOne* for each document to be removed. This ensures that all specified documents are deleted from the database and that the cache is synchronized accordingly.

Read Operations

In ClearCache, read operations are designed to efficiently leverage both the cache and the database, depending on the nature of the client's query. When a client issues a query using the _id field as the sole condition, ClearCache first attempts to find the requested document in the cache. This is the most efficient path, as a cache hit allows for instant retrieval and significantly reduces response time.

If the document is not found in the cache, the system then queries the database. After retrieving the document from the database, ClearCache stores it in the cache asynchronously. This asynchronous caching is managed by an executor service that utilizes threads, allowing the main application thread to return the requested document to the client immediately, without waiting for the caching process to complete. This design choice ensures that the system remains responsive, even during data retrieval from the database.

Special consideration is given when the client specifies a projection, which limits the fields returned in the document. In such cases, the retrieved document is not cached. This decision is made to prevent storing incomplete copies of documents in the cache, which could lead to inconsistencies and inefficiencies in future read operations that might require the complete document. By not caching documents with projections, ClearCache maintains the integrity and completeness of the cached data, ensuring that it can serve accurate and comprehensive information for subsequent queries.

It is important to note that the cache is designed to work only when queries use only the _id field. Queries that include additional conditions or projections will bypass the cache and query the database directly.

3.1.5 ClearCache Transactions

MongoDB supports multi-document transactions to ensure atomicity across multiple operations, guaranteeing that all operations within a transaction are committed together or not at all. To integrate this capability with ClearCache, the system intercepts transactions and manages them by maintaining a structure used for storing details of executed write operations. This structure, which we refer to as a map, holds information about each operation, including the document involved and the type of operation (insert, update, or delete).

When a write operation occurs within a transaction, ClearCache temporarily stores the operation details in a local map, until the commit or abort of the transaction. This step is essential because transactions ensure isolation, meaning that changes made within a transaction are not visible to other transactions until it is committed. Consequently, these changes cannot be reflected in the cache until the transaction is finalized. Moreover, the operation is also executed directly in the database.

For read operations within a transaction, ClearCache checks the timestamps to ensure consistency and isolation. If a document was modified within the same transaction, the

system retrieves it from the stored map of operations rather than from the database or cache, ensuring that the transaction's operations remain isolated until committed. This behavior is crucial for maintaining the correct sequence of operations and preserving data integrity during concurrent transactions.

To handle the case where the first operation in a transaction is a read from the cache, ClearCache manually sets the session's operation time. This is necessary because, without a direct interaction with MongoDB, the operation time remains null, which would otherwise prevent accurate timestamp checks for subsequent operations. This is done by simply setting the time with the Unix epoch of the redis server.

Upon committing the transaction, ClearCache iterates through the map of stored operations and applies the appropriate changes to the cache. For example, it inserts or updates documents in the cache, reflecting the results of the transaction. In the case of an aborted transaction, ClearCache resets the map and associated variables, ensuring that no partial or inconsistent data is left in the cache.

This approach ensures that transactions are handled efficiently while maintaining data consistency and atomicity across both MongoDB and the cache layer.

3.1.6 Handling Varying Client Concerns

In distributed systems, clients can specify different levels of read and write concerns based on their consistency requirements. Read concerns (e.g., `local`, `majority`) determine the freshness of the data returned, while write concerns (e.g., `1`, `majority`) define how many replicas must acknowledge a write before it is considered successful. When the cache is used alongside a database, discrepancies can arise if a client's read concern is incompatible with the write concern used for a cached document.

For instance, a client may cache a document using a low write concern, where the operation is acknowledged by only a single replica, while another client might later query the same document with a high read concern, expecting the data to have been acknowledged by a majority of replicas. In such scenarios, returning the cached document could lead to inconsistencies, as it may not reflect the state agreed upon by the majority of replicas. These inconsistencies become more problematic when dealing with network partitions or node failures, where the cached data might represent an outdated version no longer considered valid by the database.

To handle varying client concerns, ClearCache introduces a mechanism for tracking and validating the consistency level of cached documents. Each cached entry is assigned a `_w` field, which indicates the write concern level used at the time of caching. The `_w` field can hold values such as `majority` or `not_majority`, representing whether the document was acknowledged by a majority of replicas or not. This information allows ClearCache to evaluate whether the cached data satisfies the read concern of the current query.

ClearCache follows these rules when handling cache reads:

- **Compatible Concerns**: If the write concern of the cached document and the read concern of the query are compatible (e.g., `not_majority` with a `local` read concern), the system returns the cached document directly. Compatibility in this context means that the read concern does not impose stricter requirements than what the write concern guarantees.

- **`Majority` Write with `Local` Read**: If a document with a `majority` write concern is cached but a `local` read is requested, ClearCache can return the cached data immediately, as the `local` read concern is less stringent and does not require acknowledgment from multiple replicas.

- **Incompatible Concerns**: If a document cached with a lower write concern (e.g., `1` with `not_majority`) is queried using a higher read concern (e.g., `majority`), ClearCache performs a database query to validate the data. Depending on the timestamp of the database document:

  - **Timestamp Lower**: If the database document's timestamp is older, the cache remains unchanged, as it already holds a more recent version.
  - **Timestamp Equal**: If the timestamps match, ClearCache updates the `_w` field of the cached document to `majority` to reflect its compatibility with the stricter read concern.
  - **Timestamp Higher**: If the database document is more recent, ClearCache replaces the cached entry with the new version from the database.

By maintaining the `_w` field for each cached document, ClearCache ensures that the consistency guarantees expected by different clients are upheld, thereby preventing potential inconsistencies caused by varying read and write concerns.

Trade-Offs and Benefits

- **Secondary Reads:** Clients reading from secondary replicas benefit from faster access to data, though this data may be slightly out of sync.

- **Primary Reads with "Local" Concerns:** Clients with "local" read concerns may encounter stale data if the cache contains documents with a "majority" write concern.

- **Cache Misses:** A cache miss in a secondary-read scenario may result in data that is out of sync with the primary, leading to potential inconsistencies.

This approach ensures that ClearCache effectively manages data consistency across replicated MongoDB deployments while balancing performance and accuracy.

3.1.7 Considerations and Limitations

At this point, it is important to highlight some considerations and limitations of the ClearCache system. These considerations are essential for understanding the system's behavior and will persist for the proposed system.

- ClearCache can be integrated into existing applications if the `_id` fields are of type `String` and the documents do not contain `_ts` or `_w` fields.
- Documents lacking the `_ts` field will not be cached, resulting in cache misses. To avoid this, update existing documents to include the `_ts` field.
- When executing read operations with field projections, incomplete documents may be cached. To prevent this, perform the read operation without the projection options, then apply the desired projection after caching.
- Renaming collections in MongoDB is not advisable when using ClearCache, as it invalidates cache entries tied to the original collection name.
- Avoid using the `unacknowledged` write concern with ClearCache, as it may cause data inconsistencies between MongoDB and Redis. Use write concerns that return a response to the client.

3.2 Goal and Requirements

The primary goal of this work is to extend ClearCache by adding **support for recurrent queries**, a feature that addresses the performance limitations of the current system iteration. This extension is especially relevant for complex queries, such as aggregation pipelines, which are frequently executed in MongoDB applications.

Unlike simple caching, recurrent queries often produce dynamic results that are costly to compute and frequently updated, making it challenging to maintain cached data that is both accurate and efficient. To tackle this issue, the development will focus on utilizing MongoDB views as a mechanism for storing and maintaining the results of these complex queries. A key innovation in this approach is the implementation of ideas of incremental view maintenance, which ensures that cached views are updated incrementally as the underlying data changes, thereby retaining the value of the view in the cache. This method minimizes the need for repeated full computations and effectively reduces query latency. In light of these goals, the system must meet the following key requirements:

- **Transparency:** The system should integrate seamlessly with existing MongoDB-based applications, providing performance improvements without requiring code modifications from developers.

- **Consistency:** Cached views must remain consistent with the underlying database, preserving MongoDB's expected behaviors, particularly its *consistency guarantees* during read and write operations.

3.3 Pipeline and Stage Constraints

MongoDB's aggregation pipeline provides developers with significant flexibility in defining a sequence of stages for data processing. Each stage offers distinct functionalities, allowing for various data transformations. However, we made a deliberate choice to limit the stages supported by our system and enforce a specific execution order, simplifing the implementation and making the execution more streamlined.

Targeted Subset of Stages. Given these considerations, we opted to support a focused subset of stages that balances essential functionality with system simplicity. The chosen stages include: `match`, `project`, `group`, `sort`, `skip`, and `limit`. These stages were selected for their practicality and alignment with the core objectives of our system. By concentrating on these key stages, we ensure coverage of the most commonly needed operations without introducing unnecessary complexity.

Order and Frequency Constraints. In addition to limiting the stages, we imposed further constraints on the pipeline's structure. Specifically, we enforce a strict order for the supported stages, and each stage can only appear once per pipeline. This prescribed order also aligns with MongoDB's optimization practices[3], such as positioning the `project` stage at the beginning to minimize unnecessary data processing early in the pipeline.

3.4 View Maintenance on Cache

In this section, we will explore the design decisions and mechanisms used to maintain views in the cache. This process involves updating the cache whenever the underlying data changes, ensuring that the cached views remain consistent with the database. To achieve this, we must address several considerations, including detecting changes in the database, updating the cache accordingly, and managing the state of the views.

3.4.1 Defining Views and Data Structures

To materialize and manage views in the cache effectively, we need to define their representation in both the database and Redis. The choice of data structures in Redis plays a crucial role in performance and efficiency. We use several Redis data structures for representing and managing views, each selected based on its characteristics and the specific requirements of the view.

Data Structures for Representing Views in Redis

The following Redis data structures are utilized to represent and manage views, also represented in Figure 3.6.

- **Set or Sorted Set:** Sets are employed to store unique elements and facilitate subset operations and retrieval. For views with aggregation pipelines that include a sort stage, sorted sets are used to maintain documents in a specified order based on scores. This structure will be the one used to represent the "view".

- **Hash for lookups:** Redis hashes store metadata and document attributes, offering efficient lookups. They are preferred over sets for fetching on large datasets due to their speed and efficiency. This choice may lead to some duplication of documents across the set and the hash, this redundancy is a trade-off for performance and efficiency.

- **Lists for Accumulating Operations:** Lists are employed to manage sequences of operations, effectively functioning as queues. These are situational and used exclusively to accumulate operations that cannot be executed immediately during recomputation, with all accumulated operations being executed subsequently.

- **View State Hash:** This hash structure captures the state of the view.

View State Hash. The view state hash is a important data structure that maintains the view's status and configuration. Fields included in the view state hash are:

- **Valid View Flag:** Indicates whether the view is considered valid.

- **Limit Threshold Field:** Defines the score threshold for the subset of documents guaranteed to be present, relevant when the view includes both limit and sort stages.

- **Data Completeness Flag:** Shows whether all documents in the database that the view references are fully represented in the cache. This condition holds when all documents matching the pipeline are present in the cache, eliminating the need to query for additional documents, even if the logic indicates that more should be fetched.

- **Maintenance Flag:** Indicates whether the view is undergoing maintenance, ensuring that no updates are performed during this period. We define maintenance as the time when the view is being recomputed.

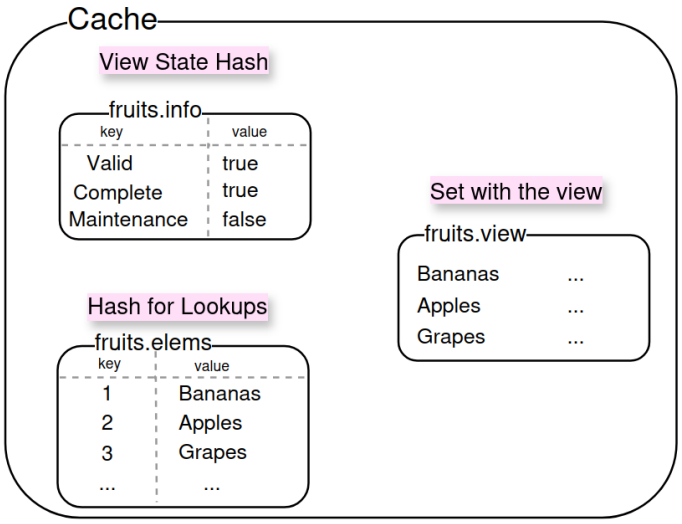

Figure 3.6: View Cache Structure

The following sections will explain how these fields are utilized during view maintenance and cache updates.

Metadata Document in MongoDB

To support the creation and ongoing maintenance of cached views, comprehensive metadata for each view is stored in MongoDB. This metadata encompasses the following elements:

- **Name of the View:** A unique identifier for the view, facilitating its management and retrieval.
- **Originating Collection:** The name of the collection from which the view derives its data.
- **Aggregation Pipeline:** The aggregation pipeline used to generate the view, detailing the sequence of operations applied.
- **Modified Aggregation Pipeline:** A version of the aggregation pipeline that may include adjustments or optimizations specific to view creation, as will be discussed in a subsequent section.
- **Creation Options:** Any specific options or parameters applied during the creation of the view, which influence its behavior and characteristics.

Storing this metadata in MongoDB of this metadata in MongoDB provides several key advantages. It ensures that cached views can be reconstructed in Redis if they are inadvertently removed, by leveraging the saved metadata. Additionally, this metadata

serves as a crucial reference for identifying which views require updates and informs the update process itself, ensuring all necessary details are considered.

3.4.2 How to Detect Changes

A crucial aspect of our system design is detecting changes in collections and ensuring that the corresponding views are updated accurately. We evaluated two primary approaches for this task: one based on the existing logic in ClearCache and another leveraging MongoDB's *change stream* feature.

Approach 1: Replicating ClearCache Logic. The first approach consists in replicating the view logic in the ClearCache client, verifying for each update which view are affected. This verification is essential to ensure that only relevant changes are processed and reflected in the cached views.
This approach has several limitations:

- **Ordering Issues:** Concurrent operations still present a significant challenge for ClearCache. Although MongoDB timestamps help mitigate these issues, they do not fully resolve all scenarios, particularly with deletions. When a document is deleted, the timestamp retained corresponds to the last insert or update. This generally works well, as we can assume that the document to be deleted will have a timestamp that is equal to or lower than that.

  However, this approach falls short when we need to know the exact time of the delete operation. For instance, if we want to determine whether a delete operation affects a specific group, we cannot do so accurately. The grouping process loses the individual timestamps of the documents, retaining only the timestamp of the most recent update or insertion. As a result, while we might identify whether a document is part of that group, we cannot definitively determine if it has already been deleted.

- **Lack of Previous State Retrieval:** This aproach lacks a way to have the previous state of a document, which can lead to increased latency and reduce the overall effectiveness of the system.

Approach 2: Leveraging MongoDB Change Streams. The second approach explored utilizing MongoDB's change streams, Subsection 2.6.6, a feature designed to monitor real-time changes within a MongoDB collection or database. Change streams provide a mechanism to receive notifications about modifications as they occur, which includes a range of useful information such as the affected collection or database, the timestamp of the operation, and in some cases, the state of the document both before and after the change, solving the problems with the current aproach.

However, for our use case change stream approach also presents notable drawbacks:

- **Commitment and Acknowledgment:** Change streams emit notifications only after an operation has been acknowledged by the majority and cannot be rolled back, which can limit flexibility in cache layer configurations.
- **Potential Inconsistencies:** Since change streams operate asynchronously, there is a risk of inconsistencies. For instance, if a majority write concern is used, the principle of *read your own writes* could be violated if the cache does not immediately reflect the latest write.
- **Performance Overhead:** Monitoring change streams continuously introduces significant overhead. MongoDB requires that these stream channels remain open, which could impose a substantial performance burden on the system.
- **Channel Management Complexity:** Keeping a change stream channel permanently open can substantially affect performance. This continuous open channel can lead to high resource consumption, potentially degrading the overall system performance.
- **Additional Requirements:** Change streams require a replica set, which introduces additional complexities and constraints on the system.

Despite the initial appeal of change streams, several significant drawbacks made this option unsuitable for our system's implementation. Consequently, **we decided to proceed with the first approach**, despite its limitations. This approach aligns better with our system's requirements and performance considerations, offering a more practical solution given the constraints and objectives we have identified.

3.4.3 Creating of the Cached Views and View Recomputation

A key objective of our system is to ensure that ClearCache closely replicates the functionality of the MongoDB driver's view creation process. In MongoDB, views are created using a dedicated function that defines an aggregation pipeline. To achieve compatibility, we planned to extend this functionality to incorporate the logic required for cached views.

Steps for Creating a Cached View The process for creating a cached view involves several steps:

1. **Initiate View Creation:** Begins by extending the view creation options to indicate that a cached view is desired. This step involves invoking MongoDB's method for view creation. Executing this step first is crucial because MongoDB imposes specific requirements on the aggregation pipeline.
2. **Verify Pipeline Constraints:** After creating the view on the MongoDB side, we conduct our verification of the pipeline, as detailed in section 3.3 are carry out. If they are not met the process terminates.

3. **Create the View in Cache:** With the pipeline validated, proceed to create the view in the cache. This involves generating the information hash discussed in subsection 3.4.1 and creating a metadata document in the database. This step is crucial because it ensures that the view is properly initialized and visible. Documents are then added to the cache, either immediately or stored for later processing.

4. **Populate the Initial View:** Next, the process queries the database for documents relevant to the newly created view. These documents are then prepared and sent to the cache in a batch, where they will be processed as a series of *upsert* operations. Specific considerations for this operation will be discussed in the section on updating the view.

5. **Complete View Maintenance Setup:** The last step it to signal the completion of the maintenance process, indicating that the view has been successfully created and is now operational and ready for use.

Redis ensures that only one operation is executed at a time, and all operations are atomic, which simplifies the process. This means we can design a function that, when executed successfully, will result in a valid view. If an error occurs, the view will be marked as invalid and will not be used.

A critical aspect of this process is the order of operations. As discussed in section 3.4.1, the metadata document in MongoDB is used to identify the view and determine which views require updates. If this document is not inserted, the view will not be "visible," and any attempt to update it will fail because the system cannot locate the view.

To avoid this issue, it is essential to first set up the state of the view in Redis before inserting the metadata document into the database. Only after these steps are completed should the view be populated with documents. If these steps are done in that order and, for example, we populate the cache first, there is a risk of losing operations, which could lead to an inconsistent view.

Refresh of Cached Views

The refresh process is triggered when a cached view becomes invalid or requires additional documents. Both scenarios involve querying the database, but they differ in their approaches. The full refresh process centers on re-populating in the same way it is done in Subsection 3.4.3, when it becomes invalid, meaning it cannot be used until the full refresh occurs. In contrast, the process of fetching additional documents is more particular; it retrieves only the documents that are missing from the cache. This approach is only relevant when the limit stage is present in the pipeline, and if we don't have all the documents in the cache, we need to fetch the missing ones to ensure the view is complete.

3.4.4 View Creation Options

While creating a view, MongoDB allows users to define collation options to specify language-specific rules for string comparison, such as letter case and accent sensitivity. We extended this feature to include additional options that control cache behavior. These options were introduced to accommodate diverse use cases and provide more flexibility. By configuring these parameters, the behavior of cached views can be adjusted to balance trade-offs like speed and space usage more effectively.

Using Set or Hash for Fetching

The first step in the process of updating a view involves fetching existing documents to compare and update the cache. Typically, a hash structure is used for this purpose, as it allows faster retrieval of documents by their `_id`. However, in scenarios where the number of documents included in a view is relatively small, using a set may be preferable. This is because the retrieval time during iteration is not significantly impacted by the reduced number of documents, making the difference in performance negligible.

Additionally, this approach avoids duplication, as documents are not stored both in the view itself and in a separate hash structure. Regardless of this choice, a hash is always employed to store temporary documents, such as those that were deleted or when dealing with stages like `limit` and `sort`, where documents may fall outside the subset of cached documents.

View Time-to-Live

The second option available to developers is to specify the TTL for each cached view. This setting determines the duration for which a cached view remains valid. A lower TTL value results in more frequent recomputations of the view, thereby reducing the chances of presenting stale data that may arise from changes in the database that are not directly detectable, such as conflicts during data replication. Conversely, a higher TTL reduces the recomputation frequency, at the cost of potentially serving outdated data for longer periods. Setting the TTL too low, as it can lead to excessive recomputation.

Control Variables for View Limiting

This set of options controls various aspects when the view is limited, enabling developers to manage the size and scope of cached views more effectively.

- **Initial Spare Documents to Fetch:** This option specifies the number of additional documents to retrieve beyond those initially required.
- **Maximum Spare Documents:** This variable determines the maximum number of documents that can be stored in the cache above the number needed to have a valid view. If this limit is exceeded, the system will automatically trim the cache to

the appropriate size removing the less revelent documents if there is a condition to determine that. This ensures the cache does not grow indefinitely, conserving memory.

- **Capped Spare Document:** This option specifies whether the number of spare documents should be capped. If enabled, the system will limit the number of spare documents to the maximum specified, using the `Maximum Spare Documents` variable. If disabled, the system will fetch all available spare documents, regardless of the limit set by `Maximum Spare Documents`.

The following sections make more clear how these options are used in the system.

3.4.5 Updating the View

Updating a view is a critical component of our system and was designed in two parts: client-side logic and server-side (Redis-side) processing. Each part plays a distinct role in maintaining consistency, accuracy, and performance.

Client-Side Logic

The update process on the client side, where changes are detected based on operations performed through the MongoDB driver. This design aims to maintain cache consistency by responding to these operations in real time.

When an update or insert operation occurs, it triggers a process similar to an *upsert*, allowing the system to handle updates and inserts in a unified manner. The client identifies which views are affected by querying MongoDB for view metadata, which contains information about the source collections associated with each view. The client saves this metadata for future use, but each time an operation is performed, the client queries the database to identify any differences between the total metadata documents and those saved locally.

Finally, for each relevant view, the client collects the necessary data and sends it to the server for processing.

Server-Side Logic

Upon receiving data from the client, the Redis server executes the update logic on its side. The server first checks if a document with the same `_id` already exists in the cache.

Handling Inserts and Updates. Both inserts and updates are processed in a similar manner to an upsert. The server follows these steps to manage inserts and updates:

1. **Check for Document Existence:** When an update operation is received, the server first checks if the document already exists in the cache.

2. **Compare Timestamps:** If the document exists, the server compares the timestamps (`_ts`) of the cached document and the incoming document.
    - If the cached document is more recent, the incoming document is considered outdated and is discarded.
    - If the incoming document is more recent, the server deletes the older document from the cache and inserts the new one.

This process ensures that the cache always contains the most up-to-date version of each document, even when updates arrive out of order.

Handling Deletes. Deletes are handled with additional care to prevent inconsistencies due to possible out of order operations. The server follows these steps to manage delete requests:

1. **Check Document Version:** When a delete request is processed, the server first checks if the document to be deleted is older than the version currently in the cache.
2. **Insert Deletion Marker:** To avoid inconsistencies that might arise if operations arrive out of order, the system inserts a special document into the cache to mark the document as deleted. This marker includes the `_id` and a timestamp (`_ts`).

Without a **Deletion Marker**, if a document is deleted and an update with an older `_ts` arrives afterward, the update might go through even though is not supposed too, causing inconsistencies since there's no record of the deletion.

3.4.6 Handling Stages During View Updates

The logic in Subsection 3.4.5 covers the general update and delete operations. However, additional logic is required when the update involves pipeline stages.

Handling the `Match` Stage

The `Match` stage is a fundamental component of database querying, used to filter documents based on specified criteria. Since this stage is always placed at the start of the pipeline and is the only stage where filtering logic is applied on the client side, we leverage it to offload the filtering operation from the server to the client. This reduces the server's computational burden and avoids latency due to server communication.

When the `Match` stage the line of thinking is:

- **Inserts and Deletes:** The client checks if the document meets the `Match` criteria. If it does, the document is sent to the cache to update the view.

- **Updates:** The client checks if the document still meets the `Match` criteria, but it does not match the criteria, a signal containing the document's `_id` and `_ts` fields is sent to the cache to invalidate any stale entry, thereby removing outdated documents from the cached view.

What can be summarized by the following logic:

```
1: if document meets Match criteria then
2:     Send document to cache to update view
3: else if operation is an update then
4:     Send {document._id, document._ts} to cache to invalidate stale entry
```

Handling the `Project` Stage

The `Project` stage is used to reshape documents by including or excluding specific fields. Like the `Match` stage, the `Project` stage is processed on the client side. If every condition on the client side is met, the client applies the projection to the document before sending it to the server. This reduces the amount of data transmitted over the network and minimizes the server's processing load.

Handling the `Sort` Stage

The `Sort` stage organizes data based on specified sorting criteria, which is essential for queries requiring ordered results. However, implementing the `Sort` stage presents obstacles, particularly when dealing with multiple sorting criteria. Redis's sorted sets are effective for single-criteria sorting but have limitations with multiple criteria.

Three main approaches were considered to address this limitation:

- **Normalization of Scores:** This approach involves combining multiple sorting criteria into a single score, enabling Redis to sort documents based on that score. However, if one criteria has a disproportionately large value compared to others, it can cause a loss of precision in the data type, resulting in incorrect sorting. This makes it infeasible to apply this method universally.

- **Reorganization Post-Retrieval:** This approach involves reordering the data after retrieval to apply multiple sorting criteria. Although it allows for complex sorting, it introduces extra processing steps that can reduce the performance benefits of using Redis's sorted sets.

- **Partial Sorting:** This approach involves sorting documents based on the most significant criteria and then applying additional sorting on the client side.

Given these constraints, we decided to limit sorting to a single criteria for now, as it is the most viable option. While alternative approaches exist, they introduce significant overhead, especially for handling large numbers of requests or when dealing with extensive views.

As for the changes to the logic, it narrows down to finding the **score** of each document and inserting it into the **sorted set**. Redis will handle the rest, maintaining the documents ordered by the score.

Handling the `Limit` Stage

The `Limit` stage restricts the number of documents returned by a query. Balancing cache efficiency with this stage involves two main approaches:

- **Ignoring the Limit:** In this approach, the view is constructed without considering the `Limit` stage. The limit is applied only when retrieving the final set of documents. This method simplifies caching but can lead to excessive memory usage due to unnecessary documents and additional filtering during retrieval.
- **Enforcing the Limit:** This method involves maintaining only the exact number of documents specified by the limit, discarding any excess. While this reduces memory usage, it complicates management a bit, especially with document deletions, which may require additional queries to ensure compliance with the view's criteria.

Given these constraints, we chose to enforce a limit during view computation. This decision arises naturally when considering the trade-offs involved. Imagine working with a dataset of 10,000 documents, but only needing to display the top 100 based on a particular sorting criteria. If we were to cache all 10,000 documents, the resource consumption would be significant, especially if the cache only ever needs to access a small subset of these documents. The result would be wasted memory and unnecessary computational effort.

Instead, by setting a reasonable limit, such as caching the top 200 documents, we ensure that only the most relevant data is stored, minimizing resource utilization while still retaining some flexibility. This strategy comes with additional benefits as well. For one, maintaining slightly more documents than required provides a buffer, which can be advantageous for handling future queries that may extend beyond the initial limit. As a result, the need for frequent database fetches is reduced, and query performance is improved.

Moreover, while deletions are a concern when dealing with large datasets, they are generally less frequent than updates or inserts. Therefore, any excess documents in the cache can be gradually adjusted by these more common operations, stabilizing the cache's state over time. In essence, by enforcing a limit, we are not just mitigating the overhead of deletions but also enhancing the overall efficiency of the system by reducing redundant interactions with the database. This approach ensures that the cache is not overloaded

with unnecessary data while still being agile enough to adapt to varying query patterns. It is a strategic compromise that balances efficiency and practicality, ultimately allowing the system to maintain high performance with minimal overhead.

Cache Trimming and Document Fetching. When the cache exceeds the document limit, a trimming process removes excess documents to align with the limit. Conversely, if the cache contains fewer documents than required, additional documents are fetched from the database.

The hash storing the view's state, detailed in section 3.4.1, includes a field indicating whether the cached data is complete. If this field is set to false, additional documents need to be fetched from the database to meet the limit. If true, no further action is necessary as all possible documents are already in the cache.

Handling `Sort` Stage Interactions. The presence of a `Sort` stage adds some complexity to managing the `Limit` stage. Documents must be cached in the order specified by the `Sort` stage, requiring precise ordering to maintain the correct sorted arrangement.

In the hash storing the view's state, a **threshold field** tracks the worst score observed during view computation. This threshold ensures that documents are included within the range [best score, threshold], where the `best score` is represent by $+\infty$ for ascending order and $-\infty$ for descending order.

During document fetching, the threshold t is updated whenever a newly fetched document has a score s that falls within a more inclusive range. For ascending order, this means $s < t$. Conversely, for descending order, $s > t$. This update is **only** applied to documents retrieved through the refresh process, ensuring that there are no missing documents in the interim.

When updating the cache, only documents with scores in the range [best score, threshold] are added, while those outside this range are excluded to maintain sorted order. The threshold is also updated during view recomputation to ensure accurate document inclusion.

Handling the `Skip` Stage

The `Skip` stage in a query pipeline excludes a specified number of documents from the beginning of the result set. When creating the view, we do not skip any documents; instead, we store all relevant documents in the cache. Subsequently, all operations—such as inserts, updates, and deletes—execute without additional logic to handle the `Skip` stage. The only extra logic introduced occurs when fetching the view, where the `Skip` stage is applied to the returned documents in the cache, ensuring the correct documents are excluded as specified.

Handling the `Group` Stage

The `Group` stage is essential for aggregating data and generating summary statistics, such as counts, sums, and averages, based on grouped attributes. Implementing this stage within cached views presents several considerations, particularly concerning consistency and the handling of updates and deletions. This section discusses these issues and the strategies adopted to address them.

Timestamp Preservation. A significant challenge in the `Group` stage is the loss of individual timestamps for each document, which are crucial for maintaining consistency and ensuring that the cache does not contain stale information. To mitigate this issue, we modify the `Group` state to include the maximum timestamp encountered during the grouping process.

This modification allows the system to track the most recent timestamp for each group, providing context about when the documents were last updated. As a result, when the grouping is complete, each group retains a static last-seen timestamp that reflects the most recent update.

This is particularly important for handling updates that may arrive out of order. By having a reliable timestamp, we can prevent the same document from being grouped multiple times, ensuring that our view remains accurate and consistent.

Handling Inserts, Updates, and Deletes. Managing inserts, updates, and deletes in the `Group` stage involves the following considerations:

- **Inserts:** When a new document is inserted, the system identifies the appropriate group and updates the cache by adding it to the group.

- **Updates:** Updates require "ungrouping" the document's previous state before reassigning it to a group. However, a limitation arises with the current approach to detecting changes: the MongoDB driver does not provide a method to update a document while simultaneously retrieving both the new and previous states. This dual-state retrieval is crucial for accurate updates. Alternatives, such as retrieving the document before updating, can also lead to inconsistencies due to potential conflicts between multiple clients performing simultaneous get and set operations. This issue is illustrated in Figure 3.7.

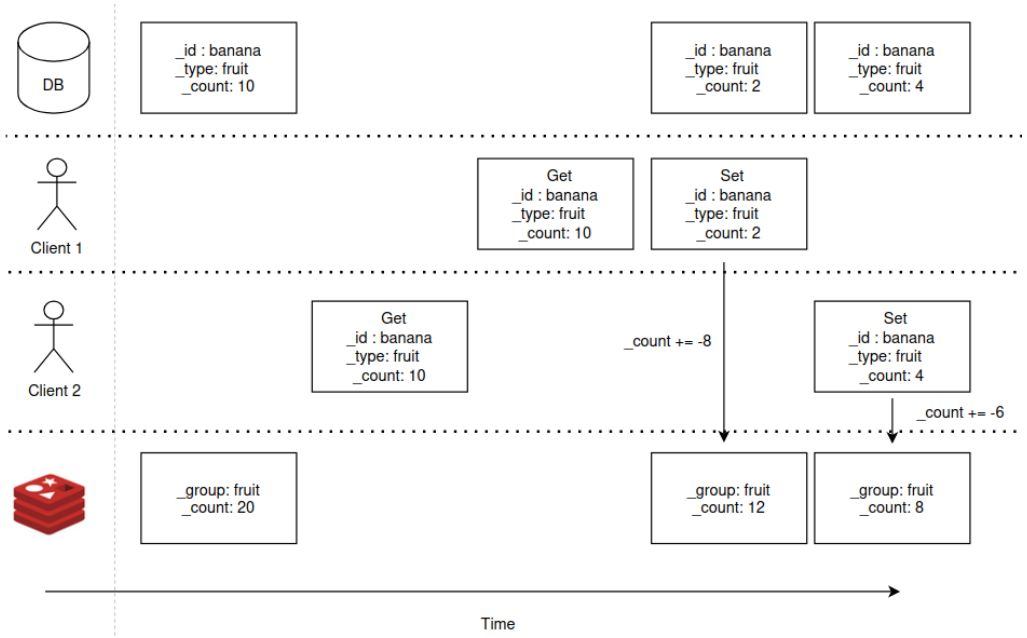

Figure 3.7: Inconsistencies created by intertwining get and set operations

- **Deletes:** Deletes suffer from the problem stated in subsection 3.4.2, as the timestamp represents the last update, not the delete operation time, making it impossible to know if a document deletion was already included on the group.

Given these complexities, the most reliable approach is to **fully recompute the entire view** after any update or delete operation. This ensures that the view remains consistent with the source collection, although with the cost of latency.

Interactions with `Limit` and `Sort` Stages. When both the `limit` and `sort` stages are present in the pipeline, managing the `group` stage is crucial for maintaining consistency. The issue arises because some groups, which are not cached, might satisfy the `sort` criteria and should be included in the result set. However, there is no efficient way to identify these groups during updates. To address this, the `limit` stage is initially bypassed, including all groups in the view. The `limit` is applied later, when fetches to the cached view are made, to ensure correct sorting and adherence to the specified limit.

3.4.7 Returning the View

The final operation involves retrieving the view from the cache, where it is stored as a set to facilitate efficient access, whether to the entire view or to a limited subset. At this stage, no decisions need to be made; the focus is solely on meeting the query requirements, particularly regarding the `Sort`, `Limit`, and `Skip` stages. The `Sort` stage poses no issues, as documents are already sorted in the cache using Redis's sorted sets. For the `Limit`

and `Skip` stages, the system applies the necessary operations to adjust the result set accordingly.

3.4.8 Dealing with Varying MongoDB Client Concerns

MongoDB's read and write concerns are crucial for ensuring data consistency and durability in distributed systems. These concerns dictate the level of acknowledgment required from the database before a read or write operation is considered successful. As mentioned in Subsection 3.1.3, to address varying levels of read concerns, ClearCache uses both a TTL for the documents as well as a specialized field, `_w`, to track write concerns for cached documents, however this approach is not feasible for the whole view structure.

The Problem with `_w` field. The core issue is that a view is made up of multiple documents, each with potentially different write concerns. While using the `_w` field to filter documents by read concern could be feasible, but it fails to handle replication events at the database level, which the cache cannot track this could lead do missing documents in the cache. For individual documents, a cache miss can trigger a database fetch if the read and write concerns do not match. However, for a view, ensuring consistency would require fetching all related documents from the database, which undermines the benefits of caching.

Enforcing Local Read Concern. Given these constraints, we opted to enforce a local read concern for accessing cached views. This choice alleviates certain restrictions and addresses most issues related to rollbacked documents or delays in reflecting changes in the underlying data. If a client requires a higher read concern, it bypasses the cache and interacts directly with the database.

While we considered implementing a majority read concern, it presents consistency issues. For example, if a client using a local read concern writes to the database, we cannot immediately reflect this change in the cache due to the majority read concern requirement. Although data will eventually replicate across the majority of nodes and be available in reads, our current change detection method does not detect this. As a result, we will be breaking the majority read concern.

Validation Flag for Cache Consistency. To address potential inconsistencies between the cache and the database resulting from rollbacks, we introduced a **validation flag** associated with the TTL of the cache. When the TTL expires, the view is marked as invalid, ensuring that any rollback operations do not persist indefinitely in the view. This mechanism maintains both consistency and reliability.

Additionally, we apply a little longer TTL to the structures to clear space in the cache. While we could have restricted the TTL solely to the structures, this approach grants us greater control over the system's behavior. By managing the TTL of the structures in this

manner, we also reduce the risk of operations failing due to the deletion of structures mid-execution.

3.4.9 Dealing with MongoDB Sessions and Transactions

MongoDB ensures atomicity for single-document operations but does not provide inherent atomicity for multi-document operations. To maintain consistency across multiple documents, MongoDB supports multi-document transactions, allowing multiple read and write operations to be grouped into a single unit of work.

While ClearCache supports basic operations within transactions (Subsection 3.1.5), extending this support to cached views presents additional complexity. MongoDB sessions can provide snapshot isolation for reads, ensuring a consistent view of the data at the start of the transaction, even amid concurrent operations. However, maintaining this level of isolation for cached views is challenging because managing multiple versions of the view within the cache and ensuring they remain consistent with the underlying database is computationally complex and often unfeasible.

Given these considerations, we opted not to support transactions for cached views unless snapshot isolation is explicitly not required. In this cases, ClearCache retrieves the cached view and applies any pending changes from the current transaction. This approach allows the system to avoid complex snapshot management while somewhat still benefiting from the performance gains of caching. If snapshot isolation is necessary, the read operation is executed directly on the database to ensure consistency.

3.5 Summary

This chapter presented the design of the ClearCache system, starting with an overview of its base version, including its architecture, operation handling, and support for a replicated MongoDB environment. Key aspects such as transaction management, timestamp ordering, and client-specific requirements were examined to establish a foundational understanding.

The chapter outlined the goals and requirements for the extended ClearCache implementation, detailing the necessary pipeline and stage constraints for effective view maintenance in the cache. It discussed the definition, creation, updating, and recomputation of views, along with important considerations such as managing MongoDB sessions and ensuring consistency during updates. Ultimately, the chapter summarized key design decisions and their implications for the system's overall functionality, illustrating how these choices enhance the performance and reliability of ClearCache.

4

Implementation

This chapter details the implementation of the ClearCache design introduced in Chapter 3. It provides a step-by-step explanation on how the design decisions are translated into the functional system. Key areas of focus include creating views (Section 4.1), loading data into the cache (Section 4.2), and implementing the necessary logic on both the client (Section 4.3) and Redis side (Section 4.4) to handle view updates. The chapter also covers how the view fetch is implemented (Section 4.5), as well as assessing the known limitations of the system (Section 4.6).

4.1 Creating the View

The first step in managing views within the cache is to create the view itself, this process reflects the discussion in Subsection 3.4.3.

4.1.1 Pipeline Validation and Modification

The process of pipeline validation and modification is crucial for ensuring that the pipeline adheres to the system's requirements and performs optimally. This process involves several steps:

- **Validation of Pipeline Stages**: The pipeline is first validated to ensure it contains only permissible stages and are in the correct sequence. Each stage is checked against a predefined list of valid stages to confirm its legitimacy. Additionally, the stages are validated to ensure they are in the correct order according to the specified pipeline model and only appear once. This involves maintaining an index to track the expected sequence and ensuring each stage appears exactly once in its designated position.
- **Modification of Specific Stages**:
  - **Group Stage**: If a *Group Stage* is present, it is modified to include additional fields. Specifically, when the pipeline contains average operations, a *group*

count field is added to the group stage. Additionally, a *_ts* field is appended to ensure accurate timestamp handling during updates. These modifications are essential because the system must track the number of documents in each group for the average to be calculated correctly. We utilize the timestamp of the most recent document in the group to prevent the repeated insertion of a document if it arrives in the cache at different times.

– **Limit Stage**: When a *Limit Stage* is included and no *Group Stage* is present, its value is adjusted based on the *spareOnCompute* option. This means that instead of strictly limiting the fetched documents, the system will fetch a few additional documents. These spare documents are used so that if deletes occur, the system does not have to immediately fetch more documents.

4.1.2 Execute the Create View

The `ExecuteCreateView` algorithm, Algorithm 1, is responsible for creating a new view based on the specified pipeline and source collection. This algorithm ensures that the view is correctly generated and stored in the cache, ready for data retrieval.

Setup and Configuration. The process begins by checking if the `createViewOptions` parameter is of type `CreateCachedViewOptions`. If so, the algorithm validates and potentially modifies the pipeline using the `validateAndModifyPipeline` function. An invalid pipeline results in an "Invalid Pipeline" error.

Metadata Handling and View Initialization. As detailed in Algorithm 1, the algorithm attempts to retrieve the view's metadata from the views metadata collection. If metadata already exists, the process ends. If no metadata is found, a new metadata document is created. This document initializes a `viewInfo` map, indicating that the view is currently under maintenance.

The algorithm examines the presence of various stages—such as sorting, limiting—and creates any fields needed for future update, like the `limitThreshold`.

Algorithm 1 Execute Create View

```
1: function ExecuteCreateView(viewName, sourceCollectionName, pipeline, createViewOptions)
2:     if createViewOptions is of type CreateCachedViewOptions then
3:         isValid ← validateAndModifyPipeline(pipeline, createViewOptions)
4:         if isValid = false then
5:             return "Invalid Pipeline"
6:         if createViewOptions.spareOnCompute > createViewOptions.maxSpare then
7:             return "Invalid Option"
8:         viewMetadata ← Retrieve document from views metadata collection
9:         if viewMetadata ≠ null then
10:            return "Already exists"
11:        else
12:            viewMetadata ← Create new view metadata document
13:        viewInfo ← Initialize empty map
14:        viewInfo.isBeingMaintained ← true
15:        sortOrder ← Retrieve sort order from view metadata
16:        if viewMetadata.limitStage ≠ null AND viewMetadata.groupStagePresent then
17:            viewInfo.hasEveryDocumentInDB ← false
18:            if sortOrder ≠ null then
19:                viewInfo.limitThreshold ← viewmetadata.isAscending ? ∞ : -∞
20:        view ← Execute the modified pipeline on the source collection
21:        if viewMetadata = null then
22:            Create and insert metadata into viewsMetadataCollection
23:        LoadViewDocumentsIntoCache(view, viewMetadata)
24:        return "Ok"
```

Completing View Creation. Once the metadata is updated, the algorithm sets `isValid` to `false` and retrieves documents from the source collection using the modified pipeline. These documents are then loaded into the cache via the `LoadViewDocumentsIntoCache` function.

For newly created views, the algorithm inserts the new metadata into the views metadata collection. Finally, the `LoadViewDocumentsIntoCache` function is called to load the documents into the cache, ensuring optimal access for future queries.

4.2 Load View Documents into Cache

The `LoadViewDocumentsIntoCache` algorithm is responsible for loading view documents into the cache. This process involves inserting the documents into the cache and updating the view metadata to reflect the current state of the view. The whole process can be split into two parts the lgic on the client side and the logic on the Redis side, we will mostly focus on the client side logic as the Redis side logic is mostly about executing the commands.

Algorithm 2 Load View Documents into Cache

```
1:  function LoadViewDocumentsIntoCache(iterable, viewMetadata)
2:      operations ← []                          ▷ Initialize list to store Redis operations
3:      viewInfo ← Retrieve view metadata from Redis
4:
5:      for each doc in iterable do
6:          operation ← operation ∪ New insert command for doc
7:
8:      viewInfo.isBeingMaintained ← false       ▷ Indicate view is not under maintenance
9:      viewInfo.isValid ← true                  ▷ Set view status as valid
10:
11:     if viewMetadata.limitStage ≠ null AND viewMetadata.groupStage = null then
12:         if iterable.size < viewMetadata.limitStage then
13:             viewInfo.hasEveryDocumentInDB ← true
14:             if sort field exists in viewMetadata then
15:                 Set sorting limits in viewInfo
16:         else
17:             viewInfo.hasEveryDocumentInDB ← false
18:     operations ← operations ∪ New operation to update view metadata in Redis
19:
20:     Send operations to execute in Redis
```

Explanation of the Load View Process The Algorithm 2 the procedure for loading view documents into the cache, a crucial step for immediate access. It begins by initializing data structures for Redis commands, processing each document from the iterable to extract relevant fields, such as timestamps, for constructing the necessary commands.

If a `limit` is specified and all the documents in the database are present, it is memorized to save database fetches in future operations. The only exception is when a `group` stage is specified, as noted in Subsection 4.1.1, we need to have all the documents in the database to ensure the view is accurate. It is also during this that the threshold is set, if a `sort` stage is present. As mentioned in section 3.4.6, this threshold is defined as the document in the `iterable`, that has the sort value closer to $-\infty$ for descending order and ∞ for ascending order.

After adding the insertion of documents to the operations that need to be done, an operation is added to update the view state in Redis and everything in sent.

On the Redis Side On the Redis side, the server receives a batch of operations along with the necessary data, processing them sequentially. The initial operations focus on inserting documents into the cache. These insertions differ from typical operations, as they originate from a validated version of the view and are marked to reflect this status. Details regarding these insert operations will be discussed in subsequent sections.

After completing the document insertions, the server updates the state of the information hash associated with the view. This update includes marking the view as valid

and setting the relevant maintenance flags, ensuring that the view is correctly identified as up-to-date and ready to serve queries efficiently. By handling operations in batches, the system minimizes network round-trips between the application and the Redis server, effectively reducing latency and enhancing performance.

Recomputations. The recomputation process is triggered when the view for some view becomes invalid, this can be if the TTL associated with the view expires. The logic executed is the same as in Algorithm 2.

4.3 Client-Side Logic for View Updates

The following pseudocode outlines the steps involved in updating the view on the client side after an insert, update or delete operation has been executed on the server. This process runs concurrently with the document update operations described in Subsection 3.1.4, leveraging threads for parallel execution.

Algorithm 3 Update View

```
1:  function UpdateView(newDocument, operation)
2:      for each viewMetadata in viewsMetadataCollection do
3:          viewMetadataDoc ← Extract metadata for the current view
4:          if viewMetadataDoc.source ≠ the current collection's namespace then
5:              Skip to the next view metadata
6:          if viewmetadata.groupCriteria ≠ null AND (operation = INSERT OR DELETE) then
7:              Invalidate the whole view
8:          if newDocument does not match viewMetadata.matchStage criteria then
9:              if operation = INSERT OR DELETE then
10:                 Skip to the next view metadata
11:             else if operation = UPDATE then
12:                 invalidateCache(viewMetadata.viewkey, document)
13:                 Skip to the next view metadata
14:         if viewMetadata.projectionStage ≠ null then
15:             updateDoc ← updateDoc.apply(viewMetadata.projectionCriteria)
16:         response ← Update the view in cache using updateDoc
17:         if response = "Refresh" then
18:             Refresh the view
```

Overview of the Update Process For each entry in the view metadata collection, the source collection that created the view is checked against the current collection's namespace. If not, it skips to the next view.

Handling Stages The algorithm then evaluates the document against the view's match stage criteria. If the document does not match, the system determines the operation type and proceeds accordingly to section 3.4.6. If it matches, we proceed.

Updating and Recomputing the View Then we apply any specified projections to include or filter fields as needed. The processed document is then used to update the view in Redis. If a recomputation is required, the algorithm refreshes the view, this can be a only a partial refresh, if a limited view needs more documents, or a full refresh if it is invalid

4.4 Redis-Side Logic for View Updates

In this subsection, we explore the Redis-side logic governing upsert/insert and delete operations within Redis views. This logic can be mostly divided into three primary phases, with the exception of grouped views, which require additional handling. The three primary phases are:

1. **Timestamp Verification:** This initial phase ensures that the document being updated or inserted is not stale. It involves checking the timestamp and increment values of the document to confirm that it represents the most recent version.

2. **Execution of the Operation:** During this phase, the actual operation is executed. For upserts and inserts, this involves removing the old document (if applicable) and inserting the new one. For deletions, the document is simply removed from the view. The specifics of this phase vary depending on whether the view is sorted or unsorted and whether it uses hashing for document retrieval.

3. **Handling Pipeline Stages and Post-processing Validation:** The final phase involves managing the different stages of the view and ensuring its overall integrity. This includes determining if the view needs resizing or if additional data needs to be fetched from the source collection in case of the view being limited. Post-processing validation checks are performed to ensure that the view remains accurate and will return the correct results.

4.4.1 Handling old Document and Timestamp Verification

Timestamp verification is a critical step in ensuring that the document being updated is current and that the operation is performed on the latest version of the document. This process helps maintain data consistency and prevents outdated information from being

inserted or updated in the view.

Algorithm 4 Verify Timestamp Function

```
1: function VerifyTimestamp(viewKey, newDocId, newDocTs, newDocInc, usesHashToFetchOption)
2:     document ← null
3:     if usesHashToFetchOption then
4:         document ← getFromHash(viewKey, newDocId)
5:     else
6:         document ← getFromSet(viewKey, newDocId)
7:     if document = null then
8:         return "Not Found"
9:     oldTs, oldInc ← get timestamp and increment from old document
10:    if (oldTs < newDocTs) OR (oldTs = newDocTs AND oldInc < newDocInc) then
11:        return "Stale"
12:    return "Ok"
```

Timestamp Verification Process The Timestamp Verification function updates or inserts a document only if it is the most recent version. It starts by retrieving the existing document using the given `viewKey` and `newDocId`. If the document is not found, it returns `"Not Found"`.

If a matching document is found, the function compares its timestamp and increment values with those of the new document. If the new document has a more recent timestamp or a higher increment value, it returns `"Stale"`, indicating that the existing document should be replaced. Otherwise, it returns `"Ok"`.

Actions Based on Verification If the result is `"Not Found"`, no further deletion is needed. If `"Stale"`, the operation stops as we are trying to insert something stale. If `"Ok"`, the old document is removed before proceeding with updates.

Handling Pipeline Stages

When updating a view, it is crucial to consider the defined pipeline stages. These stages and their interactions may necessitate additional processing, as described in Subsection 3.4.6. In the Redis-side logic, we focus primarily on handling the group, sort, skipNumber, and limitNumber stages. The group stage modifies the processing logic, while the sort stage operates independently by converting the set into a sorted set. Thus, our primary concern is with the interactions of the limitNumber and skipNumber stages with other stages.

Algorithm 5 Handling Pipeline Stages

```
1: function evaluateSortLimit(sortOrder, newDocScore, isRecomputation, limitThreshold, beingMaintained, hasAllDbDocuments)
2:     isBetter ← Evaluate if newDocScore is above limitThreshold considering sortOrder
3:     if isRecomputation = true then
4:         if isBetter = false then
5:             Update the limitNumber threshold if the new document score is more inclusive
6:     else
7:         if isBetter = false OR hasAllDbDocuments = false then
8:             if beingMaintained = true then
9:                 Accumulate the new document for redo the operation later
10:            else
11:                Create a temporary object in the cache
12:                return "Out Of Threshold"
13:    return "Ok"
14:
15: function Handling Pipeline stages(viewKey, newDocID, newDocument, viewMetadata, operationType)
16:    infoHash ← GetInfoHash(viewKey)
17:    if (InfoHash.validView = false OR InfoHash.validView = null) AND operationType ≠ "Recomputation Insert" then
18:        return "Ok"
19:
20:    if viewmetadata.sortCriteria ≠ null then
21:        score← GetScoreFromDoc(newDocument, viewmetadata.sortCriteria)
22:        if viewmetadata.limitStage ≠ null then
23:            status ← evaluateSortLimit(viewmetadata.sortCriteria, score, isRecomputation, InfoHash.limitThreshold, beingMaintained, hasAllDbDocuments)
24:            if status = "Out Of Threshold" then
25:                return status
26:
27:    AddToViewSet(viewKey, newDocument)
28:    if InfoHash.usesHashToFetchOption = true then
29:        AddToLookupHash(viewKey, newDocument)
30:    if InfoHash.beingMaintained = false then
31:        status ← PostProcessingValidation(viewKey, viewmetadata)
32:        return status
33:    return "Ok"
```

Evaluate Sorts and Limits. The `evaluateSortLimit` function handles scenarios where a limitNumber stage restricts the number of documents in the view. It checks if the new document's score exceeds the current `limitThreshold` based on the specified `sortOrder`. If the new document does not exceed the threshold and `isRecomputation` is `true`, the function may update the limitNumber threshold to include a broader range of relevant documents.

When the new document fails to meet the threshold and the view lacks full coverage (`hasAllDbDocuments` is `false`), the function either accumulates the document for later processing or creates a temporary object in the cache, depending on whether the view is actively maintained. This ensures that only the most relevant documents are retained

within the constraints of the limitNumber stage.

Handling Pipeline Stages. The `Handling Pipeline Stages` function coordinates various stages based on document and view metadata. It starts by fetching and validating essential parameters such as sort order, skipNumber, limitNumber thresholds, and the view's maintenance status. If sorting is required (indicated by `sortStageField`), the function computes the new document's score and assesses its eligibility for inclusion. In a limitNumber stage, it calls `Evaluate Sort Limit` to determine whether to insert or update the document within the view. If only a skipNumber stage is present, it uses `Evaluate Sort Skip` for insertion or updating. For unsorted views, the new document is added to the appropriate set. After insertion, the function updates auxiliary data structures, if needed. Finally, if the view is not actively maintained, an integrity check ensures consistency before returning the final status.

4.4.2 Post-processing Validation

After the view has been updated, it is essential to perform post-processing validation to ensure that the view remains consistent and accurate. This validation step involves only execute when there is present a limit in the view. The following pseudocode outlines the steps involved in post-processing validation:

Algorithm 6 Post-processing Validation

```
1:  function PostProcessingValidation(viewKey, viewmetadata)
2:      infohash ← GetInfoHash(viewKey)
3:      viewSize ← GetSetSize(viewKey)
4:      if viewmetadata.limitNumber ≠ null then
5:          if infohash.hasAllDbDocuments = false AND viewSize < viewmetadata.limitNumber then
6:              return "Partial refresh needed"
7:          toDeletedSet ← GetSetRange(viewKey, viewmetadata.limitNumber + viewmetadata.initialSpareDocs, viewSize)
8:          if viewmetadata.usesHashToFetchOption = true then
9:              for each element in the retrieved elements do
10:                 Remove from the lookup hash using the document Id
11:
12:         if  thenviewmetadata.isSorted = true
13:
14:             infohash.threshold ← score of the last element in the deleted set
15:
16:         viewSet ← getSet(viewKey)
17:         viewSet ← viewSet \ toDeletedSet
18:         if infohash.hasAllDbDocuments = true then
19:             infohash.hasAllDbDocuments ← false
20:
21:     return "Ok"
```

Post-processing Validation. The Post-processing Validation algorithm ensures that a cached view remains consistent and complies with constraints after an update.

If a `Limit Stage` is specified, the algorithm first checks whether the view's size is smaller than the number specified by it. If this condition is met and we don´t have every document possible on cache, the algorithm returns a status of `"Recompute view"`, indicating that the view needs to be refreshed to include enough documents to meet the `Limit Stage` constraint.

If the spare documents are capped (meaning there is a maximum limit on how many spare documents can be retained) and the view size exceeds the permissible range (calculated as the sum of `maxSpareDocs` and `limitNumber`), the algorithm identifies the documents that fall outside this limit. It does so by calculating the range of documents from `limitNumber + initialSpareDocs` up to the total `viewSize`.

For each document in this range, if the `usesHashToFetchOption` flag is set to true, the algorithm extracts the document ID and removes it from the hash data structure. This step is critical for maintaining a consistent state in the view's underlying data store. If the view is sorted, the algorithm updates the threshold to reflect the score of the last element in the deleted set, ensuring that any sorting constraints remain valid.

Next, the algorithm deletes the identified elements from the set to keep the view size within the acceptable limit. If documents are deleted and the view has every document possible cached, this status is updated to `false` to indicate that some documents are no longer present. The algorithm then concludes by returning an `OK` status, signaling that the validation and necessary adjustments have been successfully completed.

4.4.3 View Inserts for Grouped Views

For grouped views, the process of inserting is a little different. We only need to worry with dealing with inserts when group stage is present, because both updates and deletes will invalidate the view and recompute it, as stated in section 3.4.6.

Algorithm 7 Handle Grouped Views

```
1: function handleGroupedViews(newDocID, newDocTs, newDocInc, newDoc, viewMetadata, opType)
2:     if opType = "Recomputation Insert" then
3:         status, oldDoc ← VerifyTimestamp(newDocID, newDocTs, newDocInc, viewMetadata.usesHashToFetchOption)
4:         ProcessDocumentInsertion(status, oldDoc, newDoc, viewMetadata.sortCriteria, viewMetadata.usesHashToFetchOption)
5:     else
6:         infohash ← GetInfoHash(viewMetadata.viewKey)
7:
8:         if infohash.isValid = false then
9:             return "Ok"
10:
11:        if infohash.isBeingMaintained = true then
12:            Add the document to the operation accumulator
13:            return "Ok"
14:
15:        groupID ← Resolve the group ID of the new document
16:        status, oldDoc ← VerifyTimestamp(groupID, newDocTs, newDocInc, viewMetadata.usesHashToFetchOption)
17:        if status = "Stale" then
18:            return status
19:
20:        ProcessDocumentInsertion(status, oldDoc, newDoc, viewMetadata.sortCriteria, viewMetadata.usesHashToFetchOption)
21:    return "Ok"
22:
23: function ProcessDocumentInsertion(status, oldDoc, newDoc, sortCriteria, usesHashToFetchOption)
24:    if status ≠ "Not Found" then
25:        Remove the oldDoc from the view
26:    if sortCriteria ≠ null then
27:        score ← Calculate score for new document using sortCriteria
28:        Add the newDoc to the sorted set with the calculated score
29:    else
30:        Add the newDoc to the unsorted set
31:    if usesHashToFetchOption = true then
32:        Update the hash with the newDoc
```

Handle Grouped Views Explanation. The Algorithm 7 manages the insertion and updating of documents in grouped and sorted views based on predefined criteria. It involves multiple steps depending on the action type, including recomputation insertions and other maintenance operations, while ensuring proper handling of document timestamps for consistency.

The algorithm begins by retrieving relevant metadata, such as sorting criteria and the use of hash-based lookups for document retrieval, which dictate how documents are processed and stored. The algorithm branches based on the operation type:

- For a *Recomputation Insert*, the system checks for the existence of a document with

the same ID by evaluating its timestamp. If an old document is found, it is removed before inserting the new document. Depending on the view's configuration, the insertion may occur in a sorted or unsorted set. If sorting is required, a score is computed based on the sorting criteria. Additionally, if a hash is used, it is updated to include the newly inserted document.

- For regular inserts, since deletes and updates will always lead to a recomputation, the algorithm begins by checking the validity and maintenance status of the view. If the view is found to be invalid, the process terminates early to avoid unnecessary actions. If its being maintained the operations will be accumulated to re-run later. Next, the algorithm resolves the group ID, verifies the timestamp and if the document was not already included in the group it is added to it, ending the process by returning an "Ok" status.

4.4.4 Delete Logic

The delete logic in Redis ensures that documents are correctly removed from views while maintaining the integrity of the view. This process involves several steps to verify the timestamp, remove the old document, and update the view data structures. Is worth reminding that any delete on a view with a group stage will result on a recomputation, so we dont have to worry about them in this case. The following pseudocode outlines the steps involved in handling deletions:

Algorithm 8 Delete Logic for Redis Views

```
1:  function HandleDelete(viewkey, newDocID, newDocTs, newDocInc, viewMetadata)
2:      status, oldDoc ← VerifyTimestamp(newDocID, newDocTs, newDocInc,
    viewmetadatausesHashToFetchOption)
3:
4:      if status = "Stale" then
5:          return status
6:      insertMarkerDocument(viewKey, newDocID, newDocTs, newDocInc)
7:      if status = "Not Found" then
8:          return "Ok"
9:      if isMarkerDocument(oldDoc) = false then
10:         beingMaintained ← Check if the view is being maintained (info hash)
11:         if beingMaintained = false then
12:             return PostProcessingValidation(viewmetadata.viewKey,
    viewmetadata.limitStage, viewmetadata.hasAllDbDocuments,
    viewmetadata.baseSpareDocs, viewmetadata.maxSpareDocs,
    viewmetadata.cappedSpareDocs)
13:     return "Ok"
```

Delete Logic for Redis Views. The pseudocode describes the process for handling deletions in Redis views. It calls `VerifyTimestamp`(Alg. 4) to verify if we have a newer version of the document in cache using the timestamp fields. If the document that we are trying to insert is stale, the function returns immediately. Otherwise, it proceeds to

remove the old document from the relevant data structures, if it is found, and then inserts a new deletion marker with the updated timestamp.

If the old document in cache was a marker, meaning that is was a result on a deletion of a failer insert/update, when the view is limited and sorted it ends the process with a "Ok" status. If not, it retrieves several parameters from the view's metadata and info hash, such as the limit stage, whether the view contains all documents from the database, and various details about extra documents. The function then calls `PostProcessingValidation` (Alg. 6) to perform additional checks and returns the status. If all conditions are satisfied, it completes the deletion process by returning an "Ok" status, ensuring the view remains accurate and up-to-date. When the old document is a marker we dont need to do any of this checks just because it did not affect the view.

4.5 Fetching the View

The final step in handling views involves retrieving the actual view data from the Redis. This crucial operation ensures that the data is fetched correctly, the logic is as follows:

Algorithm 9 Get View from cache

```
1:  function GET VIEW(viewkey, viewMetadata)
2:      infohash ← GETINFOHASH(viewMetadata.viewKey)
3:
4:      if (infohash.isValid = false) AND infohash.beingRecomputed = false then
5:          Remove the view elements from Redis
6:          return "Full Recomputation Needed"
7:
8:      if infohash.beingRecomputed = true then
9:          return "Recomputation Ongoing"
10:     view ← GETSETRANGE(viewkey, viewMetadata.skipNumber or 0, viewMeta-
    data.limitNumber or -1)
11:     if view ≠ null then
12:         return view
13:     else
14:         return "View not Found"
```

Retrieving Cached Views from Redis The `Get View` algorithm is responsible for fetching a precomputed view from the Redis cache. It operates by checking the view's metadata and its validity status within Redis. The process begins with parsing the metadata associated with the requested view and retrieving the necessary keys for accessing the Redis cache. Once the view key and metadata are obtained, the algorithm proceeds to verify whether the view is currently being recomputed or if it is in a valid state.

If the view is invalid or not found in the cache and recomputation is not ongoing, the algorithm removes any existing view entries from Redis and returns a message indicating that recomputation is needed. This ensures that stale or invalid views do not persist in the cache. In contrast, if the view is marked as being recomputed, the algorithm simply

returns a message stating that the recomputation is ongoing, allowing the user to handle this scenario accordingly.

When the view is valid and not being recomputed, the algorithm moves forward to the view retrieval process. It examines metadata parameters, including sorting order, skip values, and limit values, which define how the result set should be retrieved. If sorting is required, the algorithm either retrieves the range of elements in the specified order or handles skipping and limiting elements depending on the parameters provided. When no sorting is applied, a simple range fetch from Redis is performed with the appropriate skip and limit operations. All the fetch process uses Redis functions for retrieving the sets or subsets with the needed elements.

Finally, the algorithm checks if the fetched view is non-empty. If successful, it returns the view; otherwise, it returns a "view not found" message, indicating that no matching data was found in the cache. This structured approach ensures that views are fetched efficiently and that invalid or incomplete data is handled gracefully, maintaining system integrity during view operations.

4.6 Other Considerations and Limitations

When considering the integration of the ClearCache system within an application, several key aspects must be carefully evaluated. These aspects include both limitations of the system itself and other factors that may affect its operation. The following discussion addresses additional considerations beyond those outlined in Section 3.1.7.

4.6.1 Redis Cluster

We introduced the ability to use Redis Cluster in ClearCache to support horizontal scaling and enhance overall system performance. This integration can be accomplished seamlessly by configuring ClearCache to utilize a Redis Cluster instead of a single Redis instance. The changes required to support this functionality are not particularly complex and are not the primary focus of this work so they were not included. However, there are several considerations to keep in mind when deploying ClearCache with Redis Cluster.

- **Horizontal Scaling:** Redis supports horizontal scaling through its cluster mode, which distributes data across multiple nodes via sharding. However, each node in a Redis cluster operates independently and does not have direct access to data on other nodes. To be able to update views, all view information is concentraded in one node. Consequently, this limits the ability to evenly distribute the load across all nodes in the cluster as more popular views may become hotspots for traffic in a single node.

- **Time Synchronization:** In a Redis cluster environment, each node may return different server times due to slight variations in their internal clocks. To ensure

consistent behavior and accurate time-based operations, it **is essential to synchronize the time** of all Redis nodes. This can be achieved through the use of **Chrony** and/or **Network Time Protocol**[34], which allows for automatic time synchronization across all nodes, minimizing discrepancies and maintaining consistency in time-dependent processes.

4.6.2 Stage-Specific Limitations

Certain stages in the ClearCache system do not offer the full range of functionality available in on MongoDB. These limitations must be considered when designing and implementing the system.

Match Stage

The `Match` stage in ClearCache is designed to filter documents based on specific conditions, similar to MongoDB's query capabilities. However, while it supports basic operators such as `AND` and `OR`, along with comparisons like `greater than`, `less than`, `greater or equal`, `less or equal`, `not equal`, and `equal`, it does not support more advanced query constructs. For instance, operations such as complex nested conditions or the full range of MongoDB query operators are not available. This reduces the flexibility of the `Match` stage in handling complex queries.

Project Stage

The `Project` stage allows for the inclusion or exclusion of specific fields in the documents returned by a query, akin to MongoDB's projection functionality. However, certain advanced projection capabilities are not supported. Specifically, the ClearCache system does not support operations that add new fields or reset existing fields within a document. Additionally, the system does not handle embedded fields, which means that it cannot project or manipulate fields nested within sub-documents. This limitation may restrict the ability to tailor query results to specific application needs.

Group Stage

The `Group` stage in ClearCache provides basic aggregation functionality, but it is limited in scope compared to the full aggregation framework offered by databases like MongoDB. While it supports simple aggregation operations such as `sum`, `multiply`, and `average`, it lacks support for more complex accumulator functions such as `min`, `max`, `push`, and `addToSet`. Consequently, applications requiring advanced aggregation features may find the ClearCache system insufficient for their needs.

4.6.3 Other Considerations

- **Cache Eviction Policy:** The ClearCache system relies on Redis's default eviction policy, **volatile-lru**. Under this policy, keys that have an expiration time are prioritized for eviction when the cache reaches its capacity. Since views within the ClearCache system do not have an expiration time, they will never be automatically evicted, because in the lack of keys that match the eviction criteria it will behave as a no eviction policy. However, if the cache is full, any attempt to create or update a view may fail, leading to the view being marked as invalid. If a query tries to access an invalid view, the system will revert to querying the primary database and will attempt to recompute the view. This fallback behavior can cause performance degradation, especially if cache capacity limits are frequently reached.
- **Methods that Trigger Updates:** The methods that trigger updates on views are limited to `InsertOne`, `InsertMany`, `UpdateOne`, `UpdateMany`, `DeleteOne`, and `DeleteMany`. Any operation that falls outside of these methods will not trigger an update to the views, potentially leading to inconsistencies.

4.7 Summary

This chapter detailed the implementation of the ClearCache design, illustrating how various design decisions were translated into a functional system. It presented the execution of different operations in practice, both on the client side and within the cache. Additionally, the chapter addressed the known limitations of the system and outlined important considerations. Through this comprehensive exploration, it highlighted the operational aspects of the ClearCache system, demonstrating its effectiveness while laying the groundwork for future enhancements and optimizations.

5

Evaluation

This chapter presents an evaluation of the new version of ClearCache, with a particular focus on its support for views. We will explore various components of the system, including user management, social graph operations, and timeline aggregation, to assess their performance and efficiency.

The discussion is organized into several sections. Section 5.1 introduces Socialite and its core operations, including user management, timeline aggregation, and the benchmarking framework. In Section 5.2, we outline the methodology employed for evaluating the system, detailing the workload distribution and configuration. Section 5.3 provides insights into the experimental environment, including hardware specifications and deployment architectures. Finally, Section 5.4 presents the results of the evaluations across various workloads, highlighting the performance metrics and outcomes.

By the end of this chapter, readers will gain a comprehensive understanding of the capabilities and performance of the new ClearCache version in handling views.

5.1 Socialite

Socialite[33, 32] is a reference architecture designed to benchmark and evaluate scalable data management solutions tailored for social networking applications. It addresses the complexities involved in managing extensive social data, including user graphs, various content types, and timeline aggregation mechanisms, by offering a flexible and extensible framework.

5.1.1 User Management and Social Graph Operations

The system enables robust user management by supporting the creation, retrieval, and deletion of user profiles. Each user is uniquely identified by a **User ID**, which serves as the primary key in the underlying database schema. Additionally, Socialite tracks key information such as the **followers** of each user and the **users they follow** in the user document, optimizing query performance and ensuring data consistency. This approach allows Socialite to manage a large user base while maintaining fast and reliable operations.

Schema Design

The schema design of Socialite's **User Graph Service** is optimized for managing relationships in a social network environment. It utilizes three primary MongoDB collections: `users`, `followers`, and `following`.

- **Users Collection:** Stores basic user information, including the user identifier and profile attributes. The `_id` field is used to store the `userId`, ensuring efficient lookups.
- **Followers and Following Collections:** Manage user relationships as separate documents, on top of embedding them directly within user documents (see Figure 5.1).
- **Content Collection:** Contains documents representing user-generated posts, each with a `_a` field to associate posts with its `author`, which is indexed. MongoDB indexes are applied to specific fields within a collection to enhance query performance, enabling efficient searching, sorting, and filtering of documents based on the indexed field values.

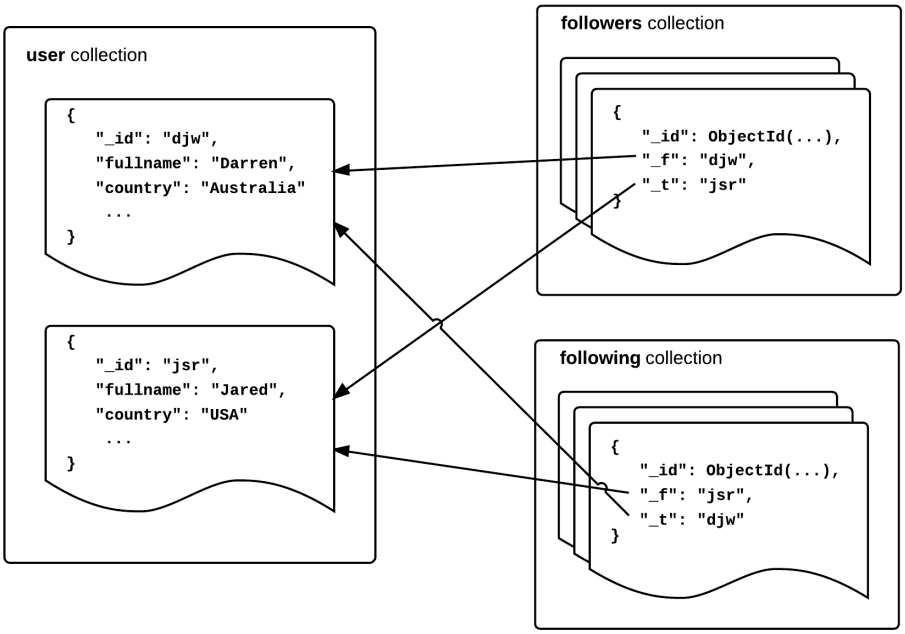

Figure 5.1: Schema design for the Socialite User Graph Service (From [32])

5.1.2 Timeline Aggregation and Management

A critical feature of Socialite is its ability to aggregate content from various sources, creating a personalized timeline for each user. This timeline is dynamically generated based on the user's social graph, ensuring that users see the latest posts from those they follow.

Socialite implements two primary strategies for timeline generation. The **Fanout on Read** model generates timelines **on-demand** when a user requests them, querying recent posts from all followed users. This approach minimizes write operations latency and **storage requirements**, as timelines are not precomputed.

In contrast, the **Fanout on Write** model enhances read latency by distributing new posts across multiple "buckets" to maintain persistent timelines. While this approach improves read performance, it introduces significant complexity and results in increased storage consumption due to the duplication of data across these buckets.

The default implementation that Socialite uses is the **Fanout on Read** model.

5.1.3 Operations in Socialite

Socialite supports a range of operations to interact with the system, among which the most important are:

- **Create User:** This operation involves the registration of a new user within the system.
- **Post Content:** Users can create new posts, which are stored in the content collection.
- **Follow User:** Users can follow other users, establishing a relationship that allows them to view the followed users' posts in their timelines. This operation updates the `following` and `followed` collections and adds the relationships to the respective users documents list.
- **Unfollow User:** This operation allows users to remove a follower relationship, reversing the effects of the **Follow User** operation.
- **Get User Posts:** This operation allows users to fetch their own posts from the content collection. It retrieves all content associated with a specific **User ID**.
- **Get User Timeline:** Users can retrieve their personalized timeline, which aggregates posts from the users they follow.
- **Get Followers:** Users can retrieve a list of their followers by retrieving the list of documents from the respective user document in the `Users` collection.
- **Get Following:** This operation allows users to see whom they are following. It functions exactly like the **Get Followers** operation but retrieves the list of users the user is following.

5.1.4 Benchmarking Framework

Socialite includes a robust benchmarking framework to evaluate system performance under different scenarios. The framework is highly configurable, allowing customization of service behavior, timeline models, and storage strategies to match varying workload requirements.

5.1.4.1 Data Loading and Workload Generation

The benchmarking framework encompasses a few operations, but the two main operations we are going to use is:

- **Bulk Data Loading:** Initializes the system with a large dataset of users, followers, and messages, simulating a realistic environment for testing. This process supports various configurations, such as setting the total number of users, the distribution and maximum number of followers, and the number of messages per user.
- **Workload Generation:** Simulates different traffic models to test the system's performance under varying load conditions. The workload generator allows fine-tuning parameters, including concurrent active users, connection concurrency, operation rates, and the mix of operation types. This enables a comprehensive analysis of how different configurations affect performance metrics like throughput, latency, and resource utilization.

5.1.5 Modifications to Socialite

To evaluate our system in different scenarios, we made a few modifications to the Socialite on top of the change to the driver used. The modifications are as follows:

- **Introduction of Topic-Based Content and TopCategories View:** We added new `topic` field to the content structure for more test cases. To showcase trending topics, we also implemented the `TopCategories` view, aggregating content by topic. This view is created using a pipeline that: filters documents to exclude null topics, groups content by topic, counting occurrences, and sorts topics by frequency to display the most popular themes.
- **Aggregation into Views:** For operations involving the retrieval of user posts or content and the retrieval of a user's timeline, the we converted any aggregation into views. This approach utilizes the same pipeline as before, but enabling testing of the system.
- **Enhancements to Benchmarking Tools:** We updated the benchmarking suite to incorporate the new features, such as testing and retrieving the `TopCategories` view. This allows for evaluation of the performance impact of the modifications.

5.2 Methodology

This section outlines the methodology used for evaluating the performance of views in the Socialite database system. We describe the workload distributions, workload configurations, deployment architectures, environment setup, and hardware specifications used in the experiments.

5.2.1 Workload Configuration and Description

While Socialite provides tools to test the system under various conditions, it does not include predefined workloads. To address this limitation, we designed a series of custom workloads to evaluate the system's performance under different scenarios. Each workload follows a **Zipfian distribution**, where a small subset of users is accessed significantly more frequently than others. The following sections describe each workload in detail:

Workload A: Balanced User Interactions In this workload, there is an equal distribution of read and write operations:

- **Post Content (50%)**: Users actively post new content.
- **Read User Posts (50%)**: Users retrieve posts from specific individuals.

Workload B: Read-Dominant Specific User Retrieval This workload emphasizes reading operations:

- **Post Content (5%)**: Minimal content posting.
- **Read User Posts (95%)**: Users primarily retrieve posts from specific individuals.

Workload C: Pure Read Operations for Specific User Posts This workload consists entirely of reading:

- **Post Content (0%)**: No content posting.
- **Read User Posts (100%)**: All operations are read-based.

Workload D: Balanced User Timeline Interactions Similar to Workload A, but focused on timeline interactions:

- **Post Content (50%)**: Users post new content.
- **Read User Timeline (50%)**: Users read their timeline.

Workload E: Read-Dominant Timeline Retrieval A read-heavy workload similar to B, but focused on the user's timeline:

- **Post Content (5%)**: Minimal content posting.
- **Read User Timeline (95%)**: Users primarily read their timeline.

Workload F: Pure Read Operations for User Timeline Retrieval This workload consists entirely of reading from the timeline:

- **Post Content (0%)**: No content posting.
- **Read User Timeline (100%)**: All operations are timeline reads.

Workload G: Balanced Content Posting, Timeline, and Posts Retrieval This workload mixes content posting with retrievals from both the timeline and specific users:

- **Post Content (50%)**: Users post new content.
- **Read User Timeline (25%)**: Users view their aggregated timeline.
- **Read User Posts (25%)**: Users retrieve posts from specific individuals.

Workload H: Read-Dominant Timeline and Posts Retrieval A read-heavy workload that combines timeline and post retrievals:

- **Post Content (5%)**: Minimal content posting.
- **Read User Timeline (50%)**: Users read from their aggregated timeline.
- **Read User Posts (45%)**: Users retrieve posts from specific individuals.

Workload I: Pure Read Timeline and Posts Retrieval This workload focuses exclusively on reading content from the timeline and specific posts:

- **Read User Timeline (50%)**: Users read their aggregated timeline.
- **Read User Posts (50%)**: Users retrieve posts from specific individuals.

Workload J: Balanced Content Posting and Trending Topics Retrieval This workload balances content posting and viewing trending topics:

- **Post Content (50%)**: Users post new content.
- **Read Trending Topics (50%)**: Users view precomputed trending topics.

Workload K: Read-Dominant Trending Topics Retrieval A read-heavy workload focused on trending topics:

- **Post Content (5%)**: Minimal content posting.
- **Read Trending Topics (95%)**: Users primarily view trending topics.

Workload L: Pure Read Operations for Trending Topics This workload consists entirely of reading trending topics:

- **Read Trending Topics (100%)**: All operations involve reading trending content.

5.3 Environment

5.3.1 Hardware Specifications

The experiments were conducted using a cluster of seven machines, each with the following specifications:

- CPU: 2 x AMD EPYC 9124 processors, providing a total of 64 threads per machine.
- Memory: 251 GiB per machine.
- Storage: 470 GB per machine.
- Network: 2 x 10 Gbps per machine.
- Operating System: Debian GNU/Linux 12 (64 bits).

5.3.2 Hardware Configuration for Experimental Setup

In order to streamline the experimental setup and manage resource allocation efficiently, we downsized the hardware configuration to better reflect a balanced and controlled environment. Specifically, each machine was limited to **4 threads** and **1 GB of memory**. This constraint ensures that the entire dataset cannot fit into memory, thus providing a more realistic scenario similar to real-world applications where memory is often a limiting factor.

In cases where multiple Docker containers were used per machine—such as for deploying the Redis Cluster, which consists of multiple nodes working together to distribute data and ensure availability—resources were divided equally among the containers. This downsizing approach allows us to stress-test the system, as each container only has a fraction of the total resources, making it easier to reach the system's limits.

5.3.3 Environment Setup

The environment for the data stores and clients was set up using Docker containers. The specific images and versions are listed below:

- MongoDB: `mongo:7.0.0`.
- Redis: `redis/redis-stack:7.4.0`.

Redis was configured with the eviction policy **volatile-lru**, where the most recently used keys are retained, and the least recently used keys are removed when fields have a TTL. This evaluation only applies to fields with a TTL. The Redis client uses the Jedis Java driver, and MongoDB uses the mongodb-driver-sync.

- MongoDB Driver: `mongodb-driver-sync:4.10.2`.
- Redis Driver: `jedis:4.4.3`.

5.3.4 Deployment Architectures

The system was deployed using a single database configuration to accommodate Socialite's session management needs, which requires a replica set, what is mandatory for using transactions. The configuration is the default one that socialite uses with the change of the read concern to `Local` from `Majority` to utilize cached views. The deployment architecture is a **three-member P-S-S replica set** database server, with the following configurations:

- Write Concern: **Majority**.
- Read Concern: **Local** (to utilize cached views).
- Read Preference: **Primary Preferred**.

When using ClearCache, a **Redis Cluster** was used. Redis being single-threaded, the cluster was configured with multiple nodes to distribute the load across different instances. This being said each machine will have four nodes to better used the cpu resources, and as each machine will have a total of 1 GB of memory, each node will have 256 MB of memory, to be fair. The cluster was configured with a replica count of one, meaning that half are master nodes and the other half are replicas, this will ultimately provide better read performance at a cost of write performance.

The deployment architecture for ClearCache is illustrated in Figure 5.2. As for only MongoDB deployment, the architecture is the same, but with only the replica set nodes.

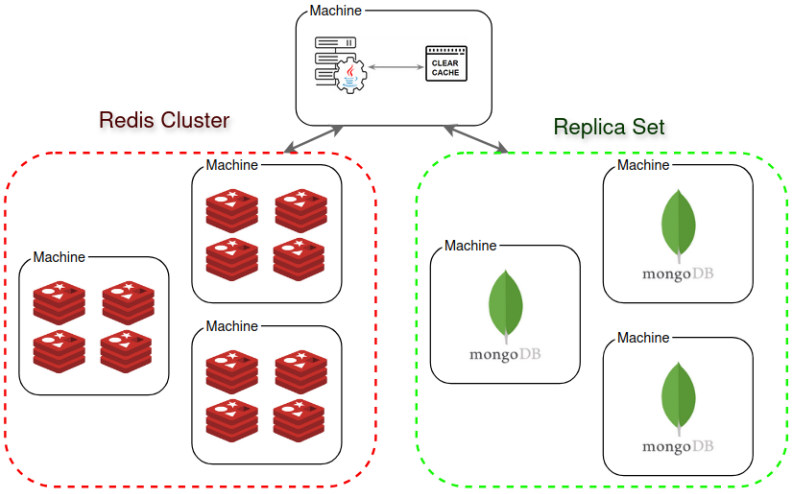

Figure 5.2: Deployment Architecture for ClearCache

5.3.5 Configuration of the Socialite Workload

Each workload execution consisted of two distinct phases: the *load phase* and the *run phase*, with the system being reset between phases to ensure consistency and fairness. The specifics of each phase are outlined below:

Load Phase

The *load phase* initializes the database systems with a representative dataset that simulates a social networking environment. This dataset consists of user profiles, follower relationships, and user-generated posts, which are loaded into the databases using the `load` command. The configuration details are as follows:

- **Number of Users**: The dataset includes a total of `10,000` users, where each user profile occupies approximately `1 KB` of storage.
- **Followers**: Each user may have up to `2,000 followers`, which are assigned randomly following a *Zipfian distribution*. As a result, certain users will have significantly more followers than others. Each follow operation results in two separate documents being created: one for the follower and another for the followee, with each document requiring roughly `100 bytes` of storage.
- **Posts**: Each user generates `500` posts, with each post occupying approximately `1 KB`.

Run Phase

The *run phase* executes the predefined workloads, simulating various user interactions within the social networking platform. Although the specific operations may vary across different workloads, the core configuration remains consistent:

- **Duration**: Each workload is run for `5 minutes`.
- **Active Users**: `100` users are active throughout the duration of the run phase.
- **Target Throughput**: No upper limit is imposed on throughput, allowing the system to operate at its maximum capacity.
- **Session Duration**: Each user session consists of `30 operations`, representing the number of actions each active user performs.

Each workload was executed multiple times with a different number of threads to observe how the system's performance scales as the number of concurrent clients increases.

5.4 Results

The following sections present the performance results for three distinct workloads, categorized by the types of operations they execute. Each workload combines similar operations but varies in the ratio of reads to writes, allowing for a focused analysis of how different operational mixes impact system performance. The two systems under evaluation, MongoDB and ClearCache, adopt different approaches to handling user interactions. For ClearCache, we selected performance-enhancing options: we do not cap the number

of documents in the cache and utilize hashing for document lookups. The results are discussed in the following sections.

The graphs presented in this section illustrate the relationship between throughput and latency for each workload. Throughput is measured as the number of operations per second (ops/sec), while mean latency is expressed in milliseconds (ms). The results were obtained by varying the number of client threads to assess how system performance scales under different conditions.

5.4.1 Workload A

In workload A (50% post; 50% get post), as shown in Figure 5.3, both MongoDB and ClearCache exhibit stable performance up to a certain throughput level. Beyond this point, the behavior of the two systems diverges significantly. MongoDB begins to exhibit increased latency as throughput approaches saturation, indicating a bottleneck in handling concurrent read and write operations. This behavior is expected since all read operations in ClearCache deployment are handled by the cache, this makes it so the database only has to handle write operations, explaining the better performance.

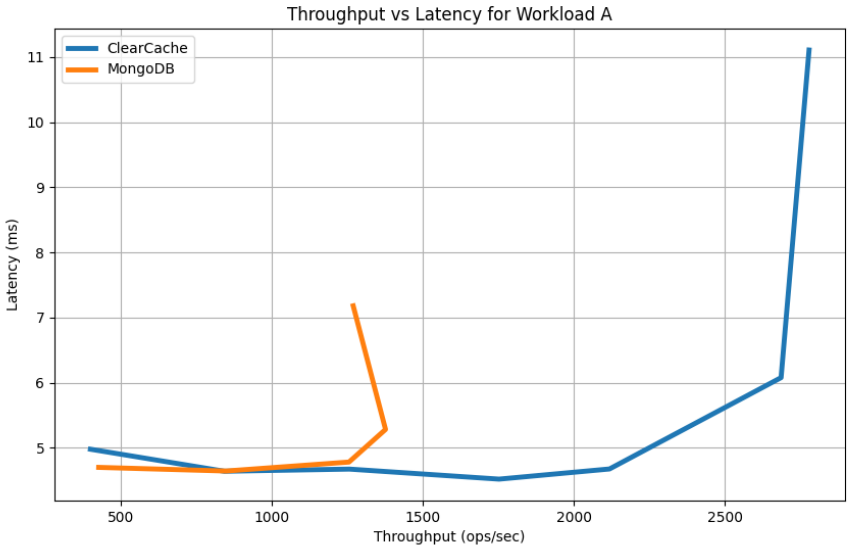

Figure 5.3: Throughput vs. Latency for Workload A

Someone could argue that ClearCache scales better because it is using more resources, as it is using more machines. To investigate this, we repeated the tests using six machines for the MongoDB deployment. The results are displayed in Figure 5.4.

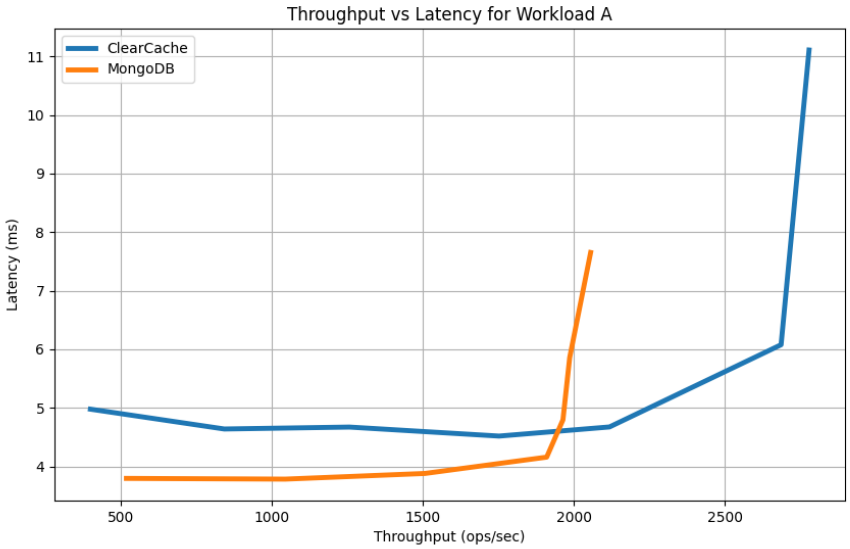

Figure 5.4: Throughput vs. Latency for Workload A with Six MongoDB Nodes

In fact the results show that MongoDB can achieve better throughput and latency, even when using more machines. The latency being better is expected as the writes are very light and for the reads, on top of leveraging MongoDB indexes, we have more machines to handle them, while in the ClearCache deployment only the Redis is used for reads, totalizing only three machines. This might explain why even though the throughput is better it still does not outperform ClearCache, saturating before it. Another point that can be made is that with six machines a lot more overall storage is used as the database is replicated in more machines.

Latency Analysis. It is essential to analyze the latency of operations under test conditions close to saturation, as this can provide a more accurate understanding of performance characteristics. The latencies presented were taken from the point right before MongoDB reached its maximum capacity to ensure fairness in comparison. As illustrated in Figure 5.5, the latency for read operations is lower with ClearCache. While one might expect write operations to be significantly slower due to additional caching overhead, this might not be entirely the case. ClearCache handles most update operations asynchronously using background threads, allowing write operations to complete without waiting for the cache to be refreshed.

Although the latency for write operations in ClearCache may not surpass MongoDB's unless the database is near saturation, it remains comparable under typical workloads. However, there is a potential drawback. Since Redis handles operations sequentially, increased write operation traffic can lead to higher read latency. This sequential processing could explain why the latency for read operations in ClearCache, while improved, is not substantially lower than in MongoDB.

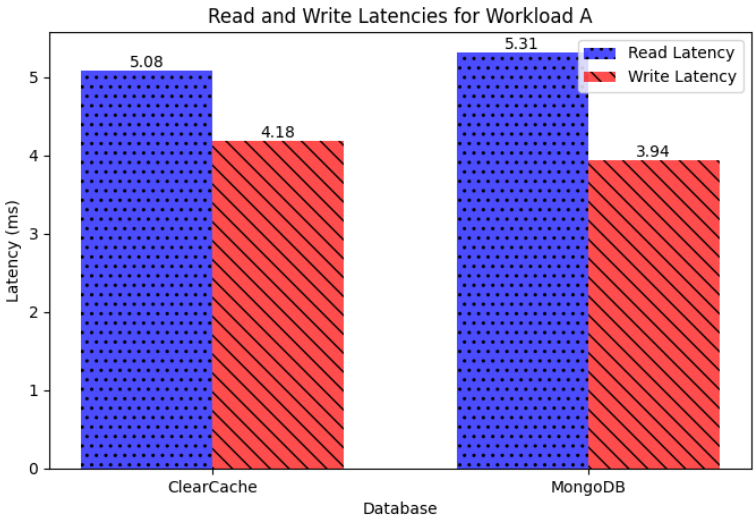

Figure 5.5: Latency for Workload A Operations

5.4.2 Workload B

As anticipated, in workload B (95% post; 5% get post), the results in Figure 5.6 show that the MongoDB deployment struggles more with the increase in read operations. Overall throughput decreases while latency slightly increases, which is expected given that even with indexes, read operations place a greater strain on the system compared to writes. In contrast, ClearCache can handle more operations than in the "Workload A", although overall latency has slightly increased. This could be attributed to measurement errors or the fact that in the previous workload, many operations that were writes could be processed in the background, not being directly measured, while every read operation is.

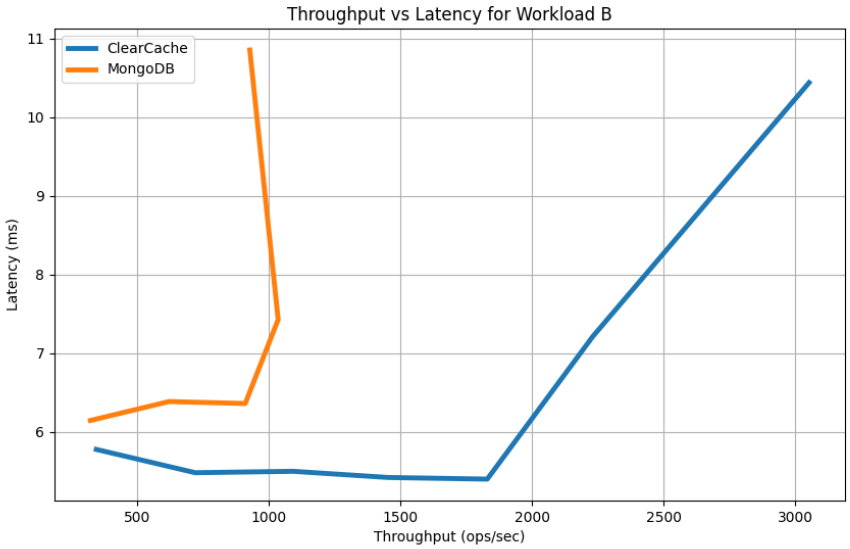

Figure 5.6: Throughput vs. Latency for Workload B

Similar to the previous workload, we repeated the tests using six machines for the MongoDB deployment. The results are illustrated in Figure 5.7. The results align with expectations; latency is improved for the reasons discussed in the "Workload A". However, the saturation point is reached slightly sooner than in the Workload A, which is anticipated since the proportion of reads is significantly higher than that of writes.

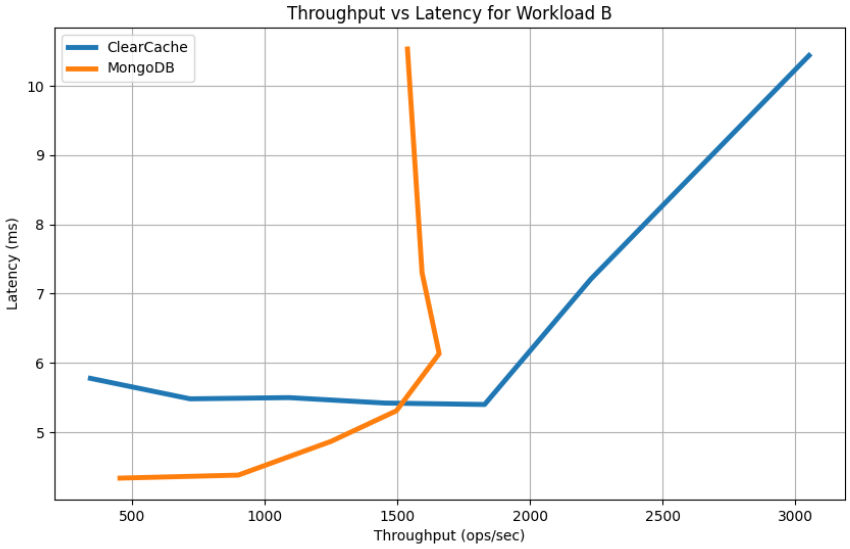

Figure 5.7: Throughput vs. Latency for Workload B with Six MongoDB Nodes

Latency Analysis. Regarding the latency of operations shown in Figure 5.8, we observe that read latency is a lot lower in ClearCache compared to MongoDB, proving that read are in fact quicker in ClearCache. The write operations in ClearCache are slower than the only MongoDB deployment, as anticipated.

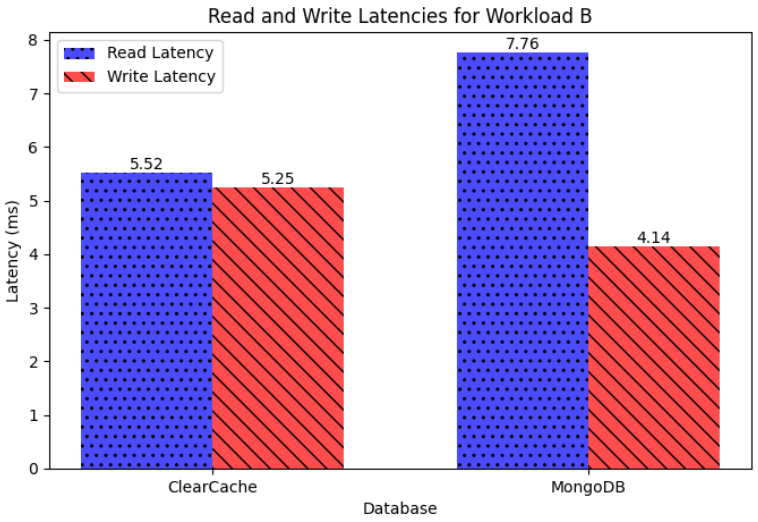

Figure 5.8: Latency for Workload B Operations

5.4.3 Workload C

Workload C (100% post; 0% get post) further emphasizes read operations by eliminating all write operations, resulting in a purely read-oriented workload. The results, as shown in Figure 5.9, are consistent with those observed in "Workload B". Given that the proportions of read and write operations are similar, all previous observations regarding latency and throughput also apply here and there is nothing new to be said.

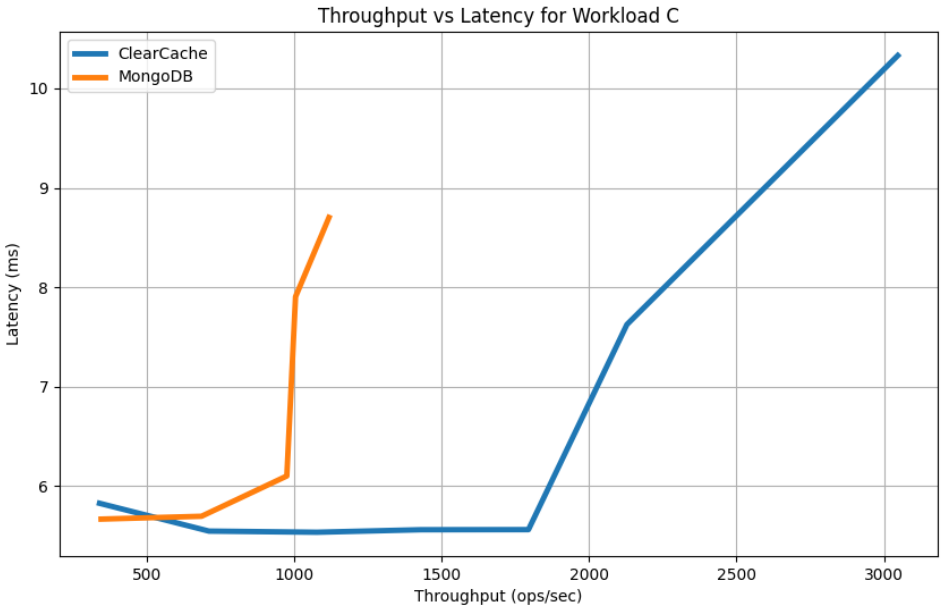

Figure 5.9: Throughput vs. Latency for Workload C

5.4.4 Workload D

The results for Workload D (50% post; 50% get timeline), shown in Figure 5.10, indicate lower overall throughput and higher latency compared to Workloads A, B, and C discussed previously. This is expected, as the read operations are more computationally intensive, putting additional pressure on the system. Both MongoDB and ClearCache exhibit similar latencies up to the saturation point of MongoDB, after which ClearCache is able to sustain a higher number of operations per second until it reaches its own saturation point.

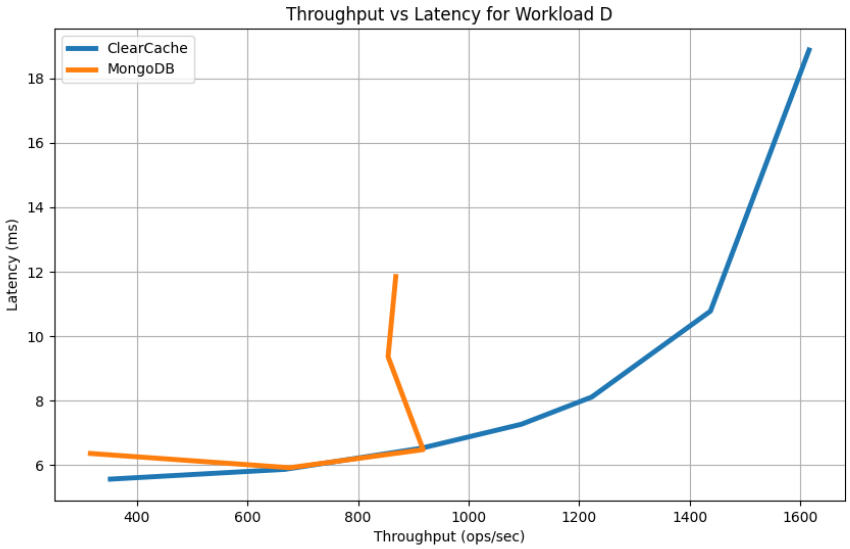

Figure 5.10: Throughput vs. Latency for Workload D

One more time we repeated the test with six machine replica set. The results in Figure 5.11 show that MongoDB achieves better latency and throughput outperforming ClearCache under higher loads. This should be expected as in addiction to the fact that writes only need to be processed one time on the database, we have more machines to handle reads. Moreover, the high number of writes together with the nature of the write operation that will trigger multiple updates in the cache, can explain the lower performance of ClearCache.

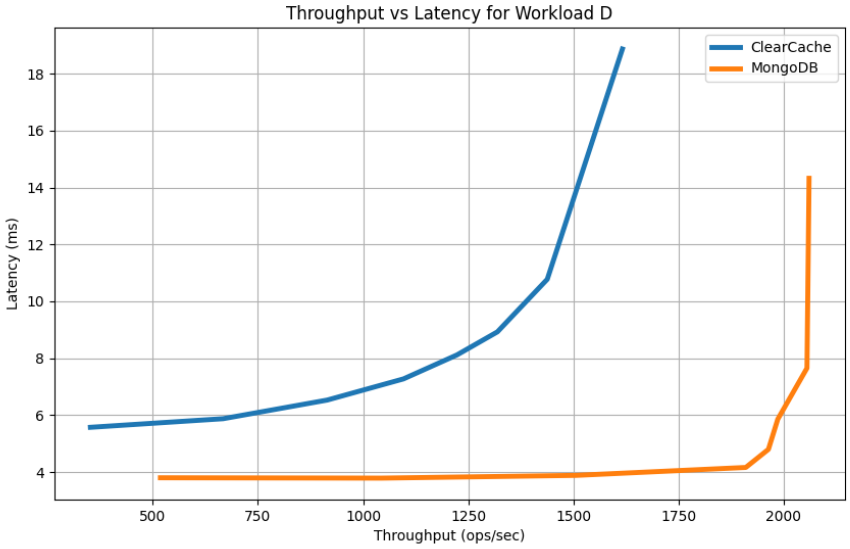

Figure 5.11: Throughput vs. Latency for Workload D with Six MongoDB Nodes

Latency Analysis. The latency results for Workload D, displayed in Figure 5.12, show that ClearCache maintains lower read latencies compared to MongoDB. However, the write operations in ClearCache show slightly increased latency values. This is likely due to the additional load on the cache caused by more view updates triggered by write operations, as one post operation can trigger multiple updates in the cached views, that are the timeline of various users.

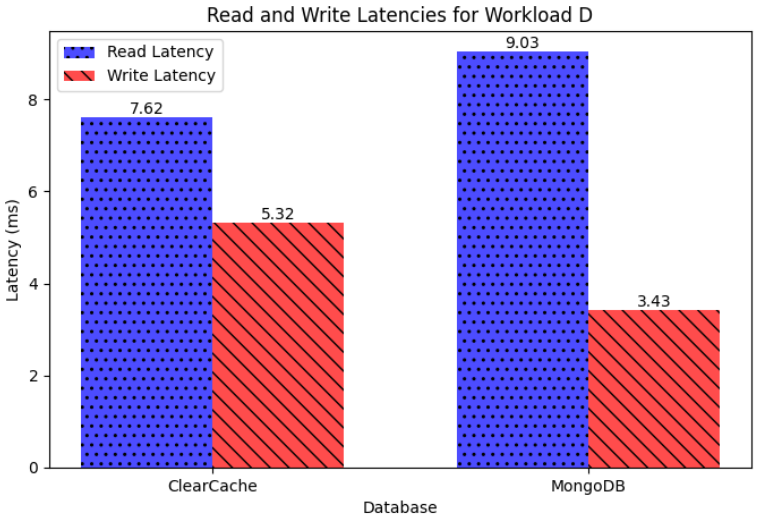

Figure 5.12: Latency for Workload D Operations

5.4.5 Workload E

The results for Workload E (95% post; 5% get timeline), depicted in Figure 5.13, reveal a continuation of the pattern observed before. With an increased number of read operations, MongoDB struggles more with throughput and latency compared to ClearCache. Interestingly, the ClearCache deployment behaves similarly to its performance in "Workload B", we believe that the only reason that its not more similar is due to the way that Redis operates internally, as a retrieval of a set can be slower the more documents it has, as we are storing the timeline of each user in a set, a lot more documents are being stored in the cache.

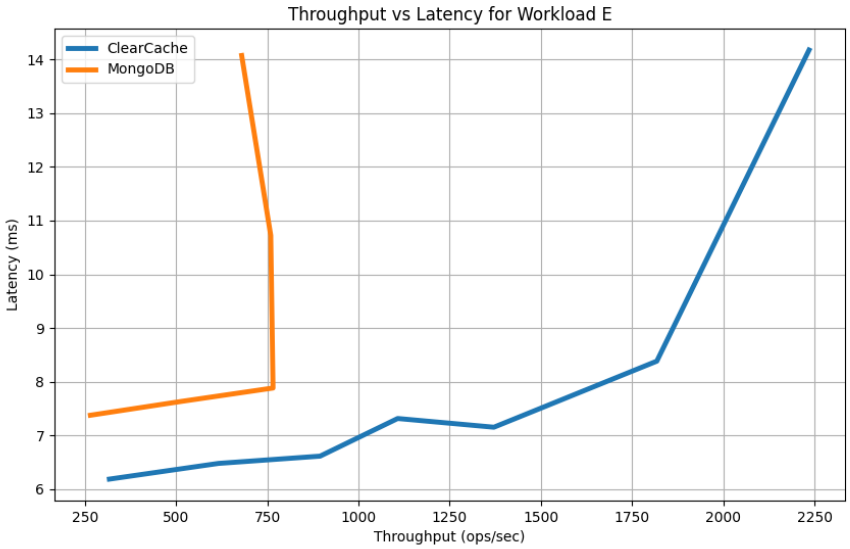

Figure 5.13: Throughput vs. Latency for Workload E

Scaling MongoDB to six nodes, as illustrated in Figure 5.14, yields improved latency and throughput results, one more time outperforming the ClearCache deployment. However, it is interesting to note that the MongoDB deployment reaches saturation earlier than ClearCache, despite the additional resources. This observation supports the notion that the primary bottleneck before lied in the write operations, which cause frequent cache updates. When read operations dominate, as in Workload E, this bottleneck less significant.

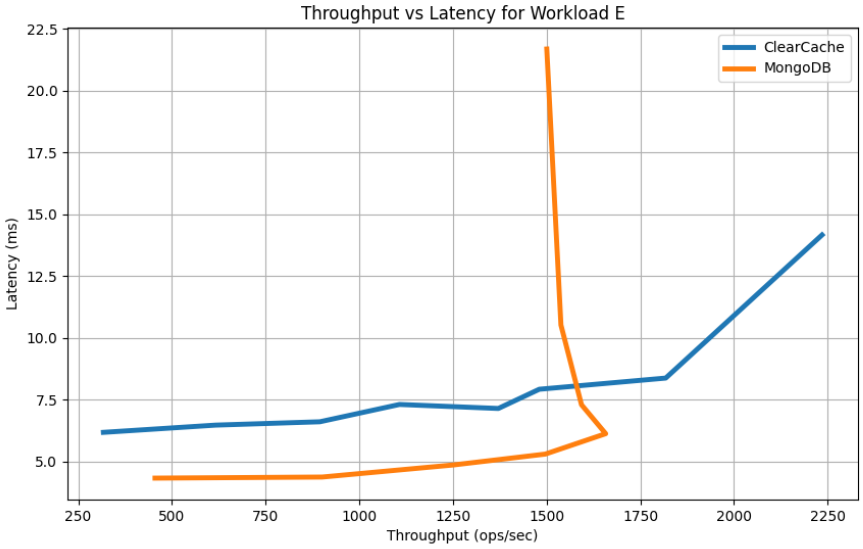

Figure 5.14: Throughput vs. Latency for Workload E with Six MongoDB Nodes

Latency Analysis. The latency values for Workload E, presented in Figure 5.15, are consistent with the observed throughput trends, offering no additional insights beyond what has already been discussed.

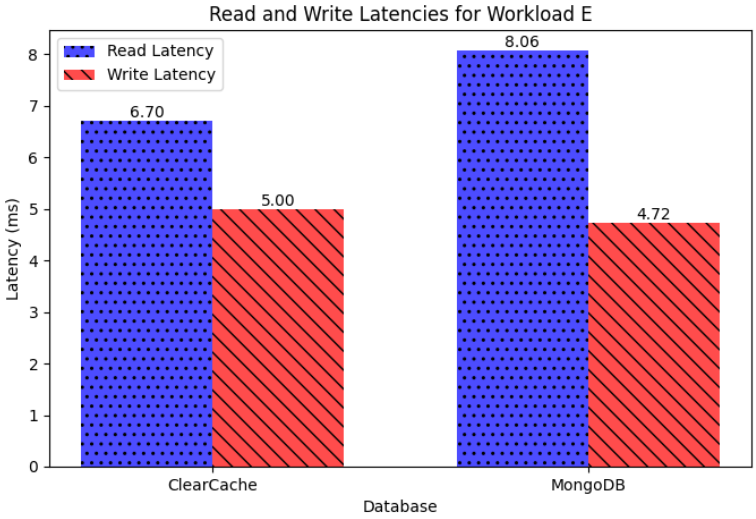

Figure 5.15: Latency for Workload E Operations

5.4.6 Workload F

The results for Workload F (100% post; 0% get timeline), shown in Figure 5.16, reflect a pattern similar to Workload E. Given that the proportion of operations remains relatively unchanged, the performance metrics for ClearCache and MongoDB are consistent with previous observations, and there is no need to further discuss the results.

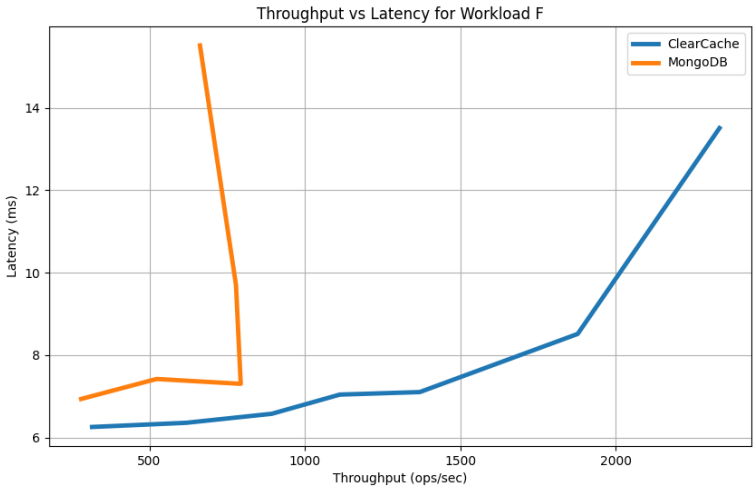

Figure 5.16: Throughput vs. Latency for Workload F

5.4.7 Workloads D, E, and F with Fanout-on-Write

As previously mentioned, Socialite provides alternative methods for modeling timelines. To further assess the performance of ClearCache, we compared its throughput and latency against a MongoDB deployment utilizing the Fanout-on-Write model. This model aims to optimize data retrieval in a manner similar to ClearCache, although it is specifically tailored to this application. It achieves efficiency by pre-aggregating user timelines directly within a database collection, storing them in precomputed documents, thus allowing retrieval of timelines with a single database query.

The Fanout-on-Write approach significantly reduces the overhead associated with frequent timeline retrievals. This comparison is illustrated in Figure 5.17, Figure 5.18 and Figure 5.19, where the MongoDB deployment consistently outperforms ClearCache in latency for lower throughputs. Notably, in Workloads E and F, the MongoDB deployment reaches its saturation point sooner than ClearCache.

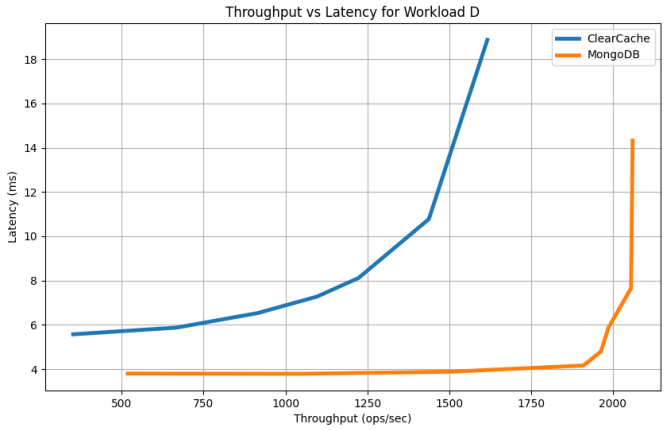

Figure 5.17: Throughput vs. Latency for Workload D with Fanout on Write

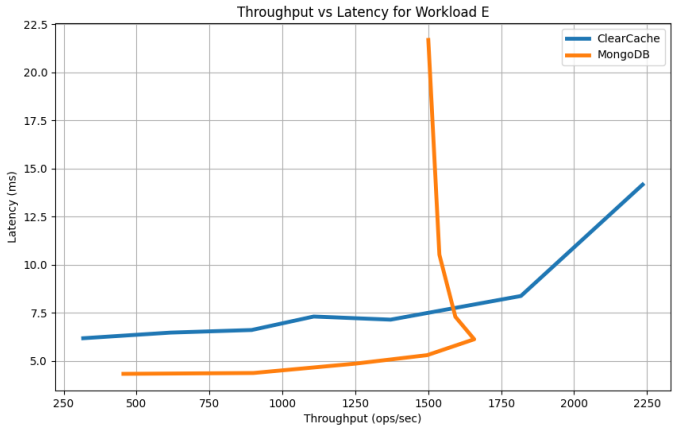

Figure 5.18: Throughput vs. Latency for Workload E with Fanout on Write

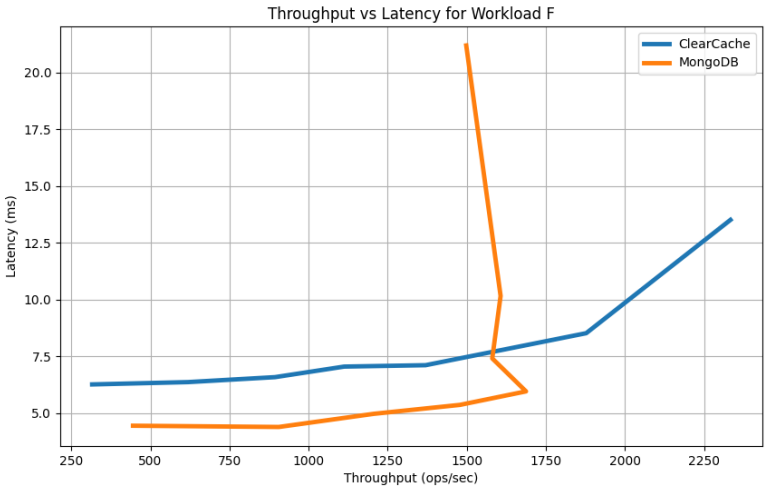

Figure 5.19: Throughput vs. Latency for Workload F with Fanout on Write

To further evaluate the performance, we deployed MongoDB with six nodes to assess its effectiveness for Workloads E and F. The results remained comparable, given that both read and write proportions were consistent. These findings, displayed in Figure 5.20, demonstrate that with this configuration, MongoDB consistently outperformed ClearCache in both throughput and latency.

These results suggest that MongoDB's Fanout-on-Write model, particularly when coupled with additional nodes, offers a more effective caching strategy. This advantage stems from its similarity to the concept of incremental views, but in this case, it occurs directly within the database. This design eliminates the latencies associated with communication with the cache.

Additionally, as timelines are stored as documents, MongoDB also leverages its internal caching for documents, further enhancing performance. In contrast, the ClearCache deployment also faces other delays because views are only updated after receiving confirmation from the database. This adds latency to operations, a drawback not present in the Fanout-on-Write model, where all processes are handled simultaneously within the databas

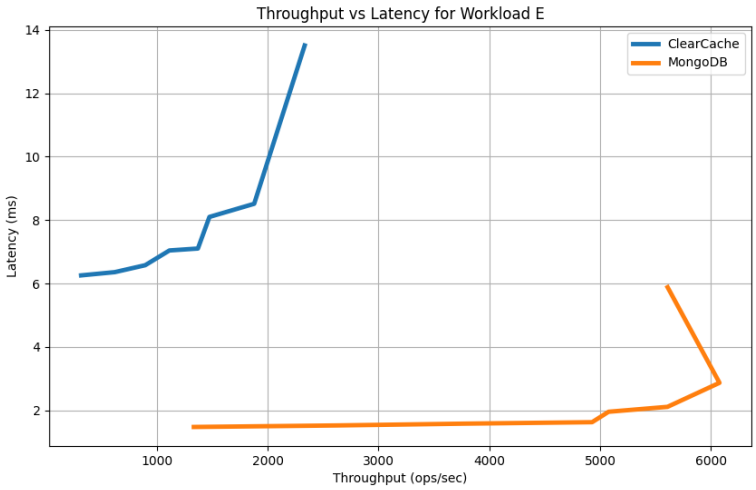

Figure 5.20: Throughput vs. Latency for Workload F with Fanout on Write and Six MongoDB Nodes

5.4.8 Results for Workloads G, H, and I

Workload G (50% post; 25% get timeline; 25% get user posts),H (5% post; 50% get timeline; 45% get user posts) and I (0% post; 50% get timeline; 50% get user posts) combine a mix of the operations seen in the previous workloads.

As depicted Figure 5.21, Figure 5.22, and Figure 5.23, the results generally follow the trends observed in the previous workloads. Since these workloads involve a combination of content posting and timeline retrieval for the read operations, and the proportions for reads and writes stay the same, the performance metrics fall within the expected range.

The only notable deviation is observed in Workload G for MongoDB, where its performance was anticipated to be higher, consistent with prior results. Apart from this exception, the outcomes are aligned with previous findings and do not provide any new insights beyond what has already been discussed.

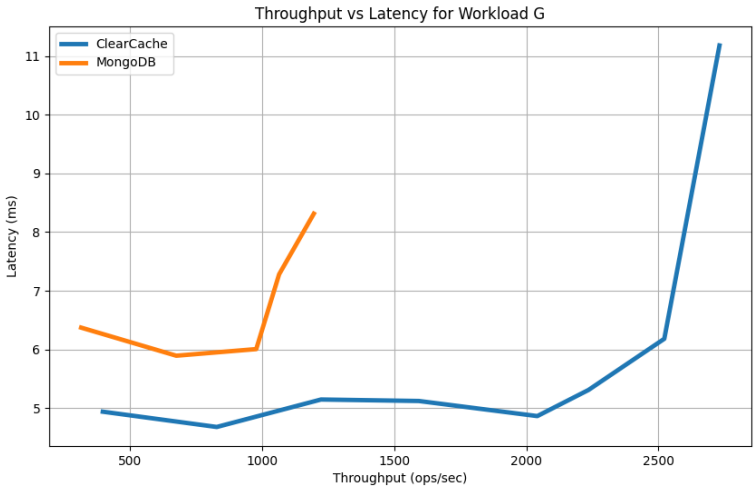

Figure 5.21: Throughput vs. Latency for Workload G

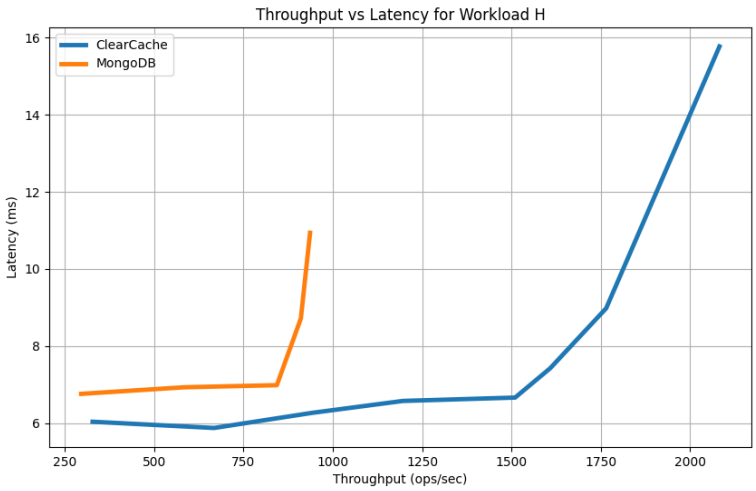

Figure 5.22: Throughput vs. Latency for Workload H

This workloads are particularly interesting as until now MongoDB has been leveraging indexes to improve read performance, but in these workloads, no indexes are used, on top of that we have a group operation that is very costly in MongoDB, as it requires a full collection scan.

5.4.9 Workloads J, K, and L

In the previous results, MongoDB has been leveraging indexes to improve read performance, as both users' posts and feed can be accessed by relying only on indexes. We now

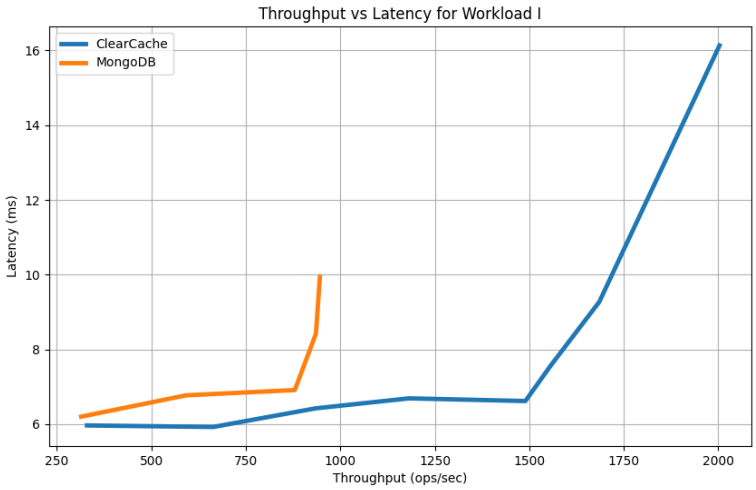

Figure 5.23: Throughput vs. Latency for Workload I

show results when accessing data involves performing a full collection scan, as it show its a lot worse as the documets are iterated one by one. These

5.4.9.1 Workload J

For workload J (50% post; 50% get trending topics), the results, depicted in Figure 5.24 and Figure 5.25, reveal a stark difference between MongoDB and ClearCache. For MongoDB, the full collection iterations to create groups for each query, results in a significant performance bottleneck. This inefficiency leads to a marked increase in latency and a drastic reduction in throughput.

In contrast, ClearCache demonstrates its inherent advantage by utilizing precomputed cached views for read operations, thereby bypassing the need for costly collection scans. Consequently, ClearCache maintains a high throughput and low latency. It is also noteworthy that ClearCache's throughput in this workload, and the next ones, is significantly higher than previously seen. This behavior, as mentioned before, is attributable to the nature of how Redis functions internally, for set retrievals the number of objects in it influences the latency. In this workloads, only a very limited number of topics are generated, what explains the better performance.

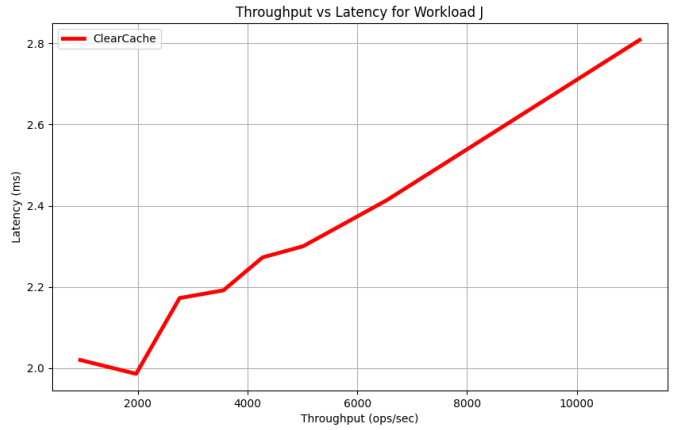

Figure 5.24: Throughput vs Latency for Workload J in ClearCache

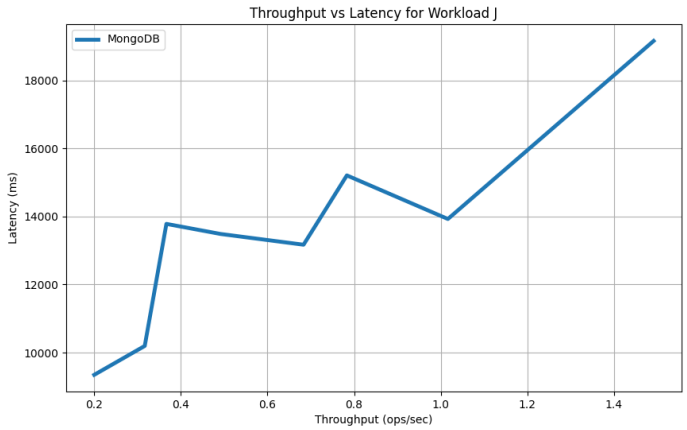

Figure 5.25: Throughput vs Latency for Workload J in MongoDB

5.4.9.2 Workload K

Workload K (5% post; 95% get trending topics) results in Figure 5.26 and Figure 5.27 show, MongoDB's performance deteriorates even more compared to Workload J due to the higher proportion of read operations that involve full collection scans. This results in significantly higher latencies and a lower throughput than in the balanced scenario.

ClearCache, on the other hand, exhibits improved performance in Workload K relative to Workload J. The system benefits from a higher read-to-write ratio, which allows it to serve more requests directly from the cache without the need for recomputation or cache invalidation. Consequently, the throughput remains consistently high, and latency stays low, showcasing ClearCache's efficiency in handling read-heavy workloads.

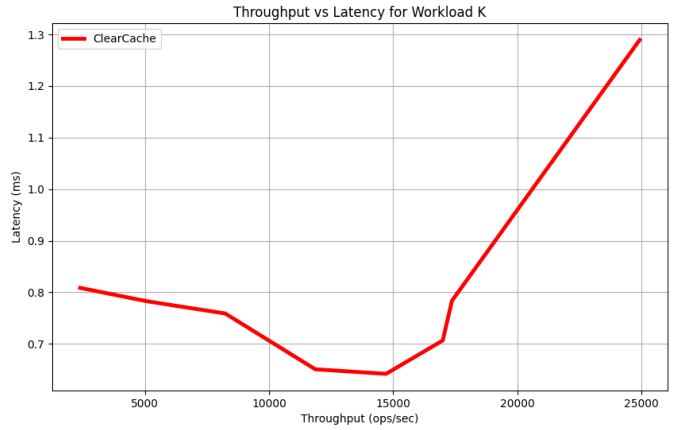

Figure 5.26: Throughput vs Latency for Workload K in ClearCache

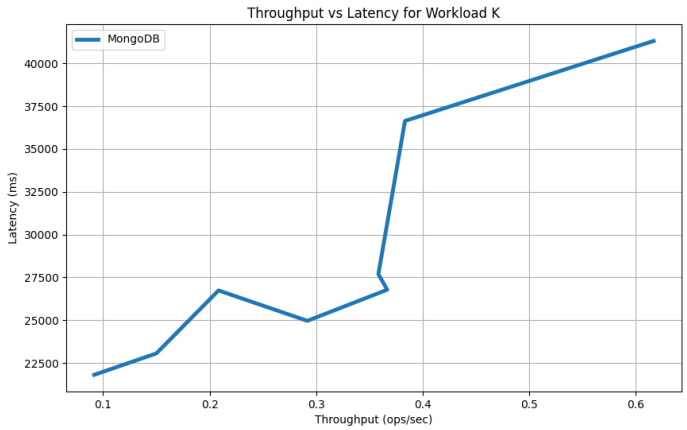

Figure 5.27: Throughput vs Latency for Workload K in MongoDB

5.4.9.3 Workload L

The results for this Workload L (0% post; 100% get trending topics) are shown in Figure 5.28 and Figure 5.29. Due to the complete absence of write operations, MongoDB's inefficiencies become even more pronounced, as each read request triggers a full scan of the collection. The resulting latencies are significantly higher, and throughput is drastically reduced compared to the previous workloads.

Each read operation is served directly from the cache, leading to minimal processing overhead. As a result, ClearCache attains its highest throughput and lowest latency for this workload, clearly demonstrating its capability to handle this type of read-intensive scenarios.

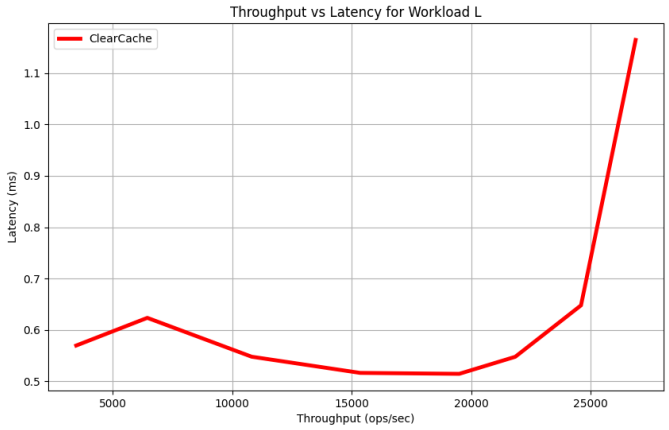

Figure 5.28: Throughput vs Latency for Workload L in ClearCache

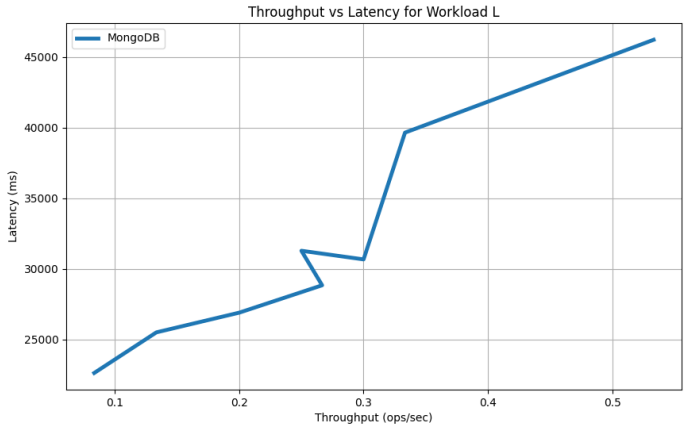

Figure 5.29: Throughput vs Latency for Workload L in MongoDB

5.5 Summary and Discussion

The results indicate that ClearCache either matches or outperforms MongoDB in a handful of scenarios, particularly when handling read-heavy workloads. This outcome aligns with expectations, as ClearCache leverages precomputed views stored in the cache, allowing it to serve read operations more efficiently and reduce the load on the underlying database.

However, as the number of write operations increases, MongoDB's performance becomes comparable and might even surpass ClearCache's for write-heavy workloads. This is because, unlike MongoDB, ClearCache incurs additional overhead by having to update cached views for each write operation, which results in increased latency and reduced throughput.

Nonetheless, ClearCache maintains a notable advantage in situations where the workload involves read operations on non-indexed fields or when views require a full collection

scan in MongoDB, such as with grouped queries. In these cases, MongoDB may experience severe performance degradation, as it needs to iterate through all documents in the collection for each query, which is computationally expensive. ClearCache, on the other hand, can still perform efficiently by serving results directly from the cache without repeatedly accessing the entire collection. This behavior highlights ClearCache's robustness in read-dominant environments, even when MongoDB's native querying mechanisms are less effective.

6

Conclusion

Application caching is a widely adopted method for enhancing the performance of web applications and services. Technologies such as Redis and Memcached efficiently store frequently accessed data in memory, alleviating the burden of repetitive database queries. However, a significant challenge arises in maintaining data consistency between the cache and the database. Concurrent requests may not immediately reflect updates, leading to users encountering outdated information and negatively impacting performance.

To address these challenges, ClearCache was created, designed as a mediator between the application server and the database. ClearCache manages data storage and intercepts user requests, coordinating read and write operations to a Redis cache instance. Specifically crafted for applications utilizing MongoDB, ClearCache leverages Redis to increase performance, always maintaining consistency between the database and cache.

The primary goal of this dissertation was to extend the existing system by introducing a mechanism for caching frequently executed queries, particularly those that can be represented as views. This extension aimed to seamlessly integrate the database and cache by providing a unified interface for data access. The proposed solution enhances the system's ability to handle repetitive query patterns while maintaining the performance and consistency of data across the cache and database layers.

The work focused on implementing a robust method for managing and updating cached views in Redis, leveraging Redis's data structures to efficiently store and query cached information. Additionally, the system was designed to accommodate varying data consistency levels provided by MongoDB, ensuring that the retrieved data adheres to the specified consistency guarantees.

To achieve this integration, the solution introduced several components, both on the client side and cache side. Client-side logic was developed to detect changes in the underlying data and trigger updates accordingly. On the cache side, mechanisms were implemented to support efficient view maintenance and updates, enabling the system to handle data pipelines without compromising performance.

Overall, the work provides a flexible and consistent approach to view caching, allowing developers to optimize query performance and reduce database load without sacrificing

consistency or reliability.

The system's performance was evaluated by testing various workloads, including a comparative analysis of applications with and without ClearCache integration.

6.1 Future Work

Upon reflection, the primary goals set forth for the creation of this system were successfully met. However, there remains potential for further enhancement and optimization of similar systems.

This section outlines additional goals for future iterations, building upon those established in the previous phase[15]. It will address both implementation advancements and opportunities for more thorough evaluation.

- To ensure compatibility with ClearCache, it is essential to implement the remaining write methods offered by the MongoDB interface. Without these methods, the views may not be updated, leading to inconsistencies within the system. By implementing all available methods in the interface, ClearCache can maintain synchronization between the cache and the underlying database, thereby enhancing data integrity and performance.

- The upcoming release of Redis 8[26] introduces several features that could significantly impact the system's functionality. Notably, the integration of Redis Stack and improvements such as JSON querying, more vector search capabilities, and time series data optimization offer new ways to handle complex queries and large data sets more efficiently. While the system currently operates well, Redis 8 could provide opportunities for rethinking certain components, particularly those related to data aggregation and real-time processing. Leveraging these new features may lead to substantial performance gains or even warrant a redesign of specific modules to fully exploit Redis 8's enhanced capabilities.

- The system's performance could be further evaluated through additional experiments across a broader range of use cases and scenarios. Given the numerous variables and cases in which the views can be tested, expanding the evaluation scope will provide a more comprehensive understanding of the system's capabilities and limitations.

Bibliography

[1] A. Adya, B. Liskov, and P. O'Neil. "Generalized isolation level definitions". In: *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. 2000, pp. 67–78. DOI: 10.1109/ICDE.2000.839388 (cit. on p. 6).

[2] H. Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *SIGMOD Rec.* 24.2 (1995-05), pp. 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785. URL: https://doi.org/10.1145/568271.223785 (cit. on pp. 6, 7).

[3] *Best Practices Guide for MongoDB*. https://www.mongodb.com/resources/products/fundamentals/best-practices-guide-for-mongodb. [Online; accessed 1-Aug-2024] (cit. on p. 34).

[4] *BSON Homepage*. https://bsonspec.org/. [Online; accessed 29-Dec-2023] (cit. on p. 19).

[5] L. S. Colby et al. "Algorithms for deferred view maintenance". In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 469–480. ISBN: 0897917944. DOI: 10.1145/233269.233364. URL: https://doi.org/10.1145/233269.233364 (cit. on p. 9).

[6] N. P. Francisco Mendes and J. Leitão. "Cache Aplicacional Consistente e Eficiente". In: (2023) (cit. on p. 24).

[7] A. Gupta and I. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In: *Data Engineering Bulletin* 18 (1999-11) (cit. on p. 8).

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. "Maintaining views incrementally". In: *SIGMOD Rec.* 22.2 (1993-06), pp. 157–166. ISSN: 0163-5808. DOI: 10.1145/170036.170066. URL: https://doi.org/10.1145/170036.170066 (cit. on p. 12).

[9] J. V. Harrison and S. W. Dietrich. "Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach". In: *Workshop on Deductive Databases*. Springer-Verlag. 1992, pp. 56–65 (cit. on p. 12).

[10] J. Hoxmeier, C. Dicesare, and Manager. "System Response Time and User Satisfaction: An Experimental Study of Browser-based Applications". In: *Proceedings of the Association of Information Systems Americas Conference* (2000-01) (cit. on p. 1).

[11] A. Labrinidis and Y. Sismanis. "View Maintenance". In: *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, 2009, pp. 3326–3328. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_852. URL: https://doi.org/10.1007/978-0-387-39940-9_852 (cit. on pp. 7–9).

[12] L. Liu and M. T. Zsu. *Encyclopedia of Database Systems*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 0387355448 (cit. on pp. 4, 5).

[13] 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).

[14] *Memcached Homepage*. https://memcached.org/. [Online; accessed 29-Dec-2023] (cit. on pp. 1, 16).

[15] F. Mendes. "Consistent and Efficient Application Cache". Master's thesis. Lisbon, Portugal: Universidade NOVA de Lisboa, 2023 (cit. on pp. 24, 25, 27, 28, 95).

[16] J. Mertz and I. Nunes. *Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art*. 2020-11 (cit. on p. 2).

[17] *MongoDB Change Streams Documentation*. https://www.mongodb.com/docs/manual/changeStreams/. [Online; accessed 15-Aug-2024] (cit. on p. 22).

[18] *MongoDB Default Majority Write Concerns providing stronger durability guarantees out of the box*. https://www.mongodb.com/blog/post/default-majority-write-concern-providing-stronger-durability-guarantees-out-box. [Online; accessed 15-Jan-2024] (cit. on p. 20).

[19] *MongoDB Homepage*. https://www.mongodb.com. [Online; accessed 29-Dec-2023] (cit. on p. 18).

[20] *MongoDB Materialized Views*. https://www.mongodb.com/docs/manual/core/materialized-views/. [Online; accessed 3-Jan-2024] (cit. on p. 22).

[21] *MongoDB Read Concerns*. https://www.mongodb.com/docs/manual/reference/read-concern/. [Online; accessed 15-Jan-2024] (cit. on p. 20).

[22] *MongoDB Replication*. https://www.mongodb.com/docs/manual/replication/. [Online; accessed 1-Jan-2023] (cit. on p. 19).

[23] *MongoDB Sessions Documentation*. https://www.mongodb.com/docs/manual/reference/method/Session/. [Online; accessed 15-Aug-2024] (cit. on p. 22).

[24] *MongoDB Views*. https://www.mongodb.com/docs/manual/core/views/. [Online; accessed 3-Jan-2024] (cit. on p. 22).

[25] *MongoDB Write Concerns*. https://www.mongodb.com/docs/v2.4/core/write-concern/. [Online; accessed 15-Jan-2024] (cit. on p. 19).

[26] *Redis 8 Release*. https://redis.io/blog/introducing-another-era-of-fast/. [Online; accessed 1-Aug-2024] (cit. on p. 95).

[27] *Redis Cluster*. https://redis.io/docs/management/scaling/. [Online; accessed 29-Dec-2023] (cit. on p. 17).

[28] *Redis Data Types*. https://redis.io/docs/data-types/. [Online; accessed 29-Dec-2023] (cit. on p. 17).

[29] *Redis Homepage*. https://redis.io/. [Online; accessed 29-Dec-2023] (cit. on pp. 1, 17).

[30] *Redis programmability*. https://redis.io/docs/interact/programmability/. [Online; accessed 26-Jan-2024] (cit. on p. 17).

[31] W. Schultz, T. Avitabile, and A. Cabral. "Tunable consistency in MongoDB". In: *Proc. VLDB Endow.* 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. 19).

[32] *Socialite Documentation*. https://github.com/mongodb-labs/socialite/tree/master/docs. [Online; accessed 1-Sep-2024] (cit. on pp. 66, 67).

[33] *Socialite GitHub Repository*. https://github.com/mongodb-labs/socialite/tree/master. [Online; accessed 1-Sep-2024] (cit. on p. 66).

[34] *Synchronize cluster node clocks*. https://redis.io/docs/latest/operate/rs/clusters/configure/sync-clocks/. [Online; accessed 3-Sep-2024] (cit. on p. 64).

[35] Y. Zhuge, H. Garcia-Molina, and J. Wiener. "The Strobe algorithms for multi-source warehouse consistency". In: *Fourth International Conference on Parallel and Distributed Information Systems*. 1996, pp. 146–157. DOI: 10.1109/PDIS.1996.568676 (cit. on p. 12).

[36] Y. Zhuge et al. "View maintenance in a warehousing environment". In: *SIGMOD Rec.* 24.2 (1995-05), pp. 316–327. ISSN: 0163-5808. DOI: 10.1145/568271.223848. URL: https://doi.org/10.1145/568271.223848 (cit. on p. 12).