University of Magdeburg

School of Computer Science

Dissertation

Cloud-based Support for Massively Multiplayer Online Role-Playing Games

Author:

Ziqiang Diao

February 3, 2017

Supervisor:

Prof. Dr. Gunter Saake

Institute of Technical and Business Information Systems

Diao, Ziqiang:
Cloud-based Support for Massively Multiplayer Online Role-Playing Games
Dissertation, University of Magdeburg, 2017.

Cloud-based Support for Massively Multiplayer Online Role-Playing Games

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Ziqiang Diao

geb. am 20.12.1983 in Fushun, China

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Kai-Uwe Sattler
Prof. Dr. Andreas Thor

Magdeburg, den February 3, 2017

Abstract

Massively Multiplayer Online Role-Playing Games (MMORPGs) are very sophisticated applications, which have significantly grown in popularity since their early days in the mid-´90s. Along with growing numbers of users the requirements on these systems have reached a point where technical problems become a severe risk for the commercial success. Within the CloudCraft project we investigate how Cloud-based architectures and data management can help to solve some of the most critical problems regarding scalability and consistency. In this work, we describe an implemented working environment based on the Cassandra DBMS and some of the key findings outlining its advantages and shortcomings (e.g., guarantee of game consistency) for the given application scenario.

As outlined for instance by the CAP theorem [GL02], achieving consistency guarantees within a 100% available and fault-tolerant distributed system is impossible. Nevertheless, in real-life applications actual properties are neither black nor white and the degree of fulfillment of requirements depends on the likelihood of failures and communication parameters of distributed systems. While typical Cloud-based applications weaken consistency in accordance with less strict applications requirements, strong consistency can also be achieved, for instance by tunable consistency. This, however, often comes with a strong degradation of scalability (performance of growing clusters) and availability. Based on a project investigating the usefulness of Cloud DBMS for MMORPGs we describe how strong consistency can be provided for such a scenario, by still proving a high-level of availability and performance suitable for this specific application. For this purpose we implement a lightweight mechanism to detect failures based on timestamps and only react accordingly if required.

Zusammenfassung

Massively Multiplayer Online Role-Playing Games (MMORPGs) sind sehr anspruchsvolle Anwendungen, die seit ihren Anfängen Mitte der '90er Jahre deutlich an Beliebtheit gewonnen haben. Neben den wachsenden Nutzerzahlen haben die Anforderungen an diese Systeme einen Punkt erreicht, an dem technische Probleme ein ernsthaftes Risiko für den kommerziellen Erfolg werden. Im Rahmen des CloudCraft-Projekts untersuchen wir, wie Cloud-basierte Architekturen und Datenmanagement dazu beitragen können, einige der wichtigsten Probleme hinsichtlich Skalierbarkeit und Konsistenz zu lösen. In dieser Arbeit beschreiben wir eine implementierte Arbeitsumgebung basierend auf dem Cassandra DBMS und einige der wichtigsten Erkenntnisse, welche die Vorteile und Mängel (z. B. Garantie der Spielkonsistenz) für das gegebene Anwendungsszenario skizzieren.

Wie zum Beispiel durch das CAP-Theorem [GL02] dargestellt, ist das Erreichen von Konsistenzgarantien innerhalb eines 100% verfügbaren und fehlertoleranten verteilten Systems unmöglich. Dennoch sind in realen Anwendungen tatsächliche Eigenschaften weder schwarz noch weiß, und der Grad der Erfüllung der Anforderungen hängt von der Wahrscheinlichkeit von Ausfällen und Kommunikationsparametern verteilter Systeme ab. Während typische Cloud-basierte Anwendungen Konsistenz in Übereinstimmung mit weniger strengen Anwendungsanforderungen schwächen, kann eine starke Konsistenz auch beispielsweise durch einstellbare Konsistenz erreicht werden. Dies führt jedoch oftmals zu einer signifikanten Verschlechterung der Skalierbarkeit (Leistung wachsender Cluster) und Verfügbarkeit. Basierend auf einem Projekt, das die Nützlichkeit von Cloud DBMS für MMORPGs untersucht, beschreiben wir, in welchem Umfang Konsistenz für ein solches Szenario erreicht werden kann, indem immer noch ein hohes Maß an Verfügbarkeit und Leistung für diesen spezifischen Anwendungsfall nachgewiesen wird. Zu diesem Zweck implementieren wir einen Leichtgewicht-Mechanismus zur Erkennung von Fehlern auf der Basis von Zeitstempeln und reagieren nur dann entsprechend, wenn es erforderlich ist.

Acknowledgments

This work has been completed with the help of many people. Here, I would like to express my sincere gratitude to them.

First of all, I would like to thank my supervisor, Prof. Dr. Gunter Saake. Thank him for giving me the opportunity to pursue further studies in Germany. This work was supported by his careful guidance. From him, I realized the precision and seriousness of Germans. He helped me all the time. Even when his hand was injured and unable to write, or during the holidays, he still insisted on working in order to help me improve my dissertation. Here, I would like to once again express my respect for him. It is my pleasure to study in your working group.

Secondly, I would like to thank Dr. Eike Schallehn, who is always ready to help others. Whether I have any difficulty in my studies or in my life, he will lend a generous hand to help me. Eike also played a crucial role in the completion of this work. Many of my ideas are in the discussion with him. Therefore, he is almost a coauthor of all my publications. He is also a modest man. Once I told him that you are like my second supervisor. However, he replied that he was just a person who shared my research interests.

I would like to thank my family in particular. They care about me throughout my life and study. Especially during my Ph.D. studies, I have always had financial problems. It is my parents' support and encouragement that I could concentrate on my research. My love for you is beyond words. I could only redouble my efforts to speed up the completion of this work in order to repay your kindness as soon as possible.

I really like the atmosphere of the DBSE working group. Each college, whether former or current, such as Ingolf Geist, Syed Saif ur Rahman, Ateeq Khan, Siba Mohammad and so on, is so friendly and helpful. There was even a college, who gave me many suggestions anonymously, when I just joined the group. I still do not know who he/she is, and I do not need to know that. What I should do is to pass this kindness to the other group members, especially the new ones.

Finally, I would like to thank the students that I supervised over these years, they are Shuo Wang, Pengfei Zhao and Meihua Zhao. The three important experimental parts of this work were done with their assistance.

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

3NF	Third Normal Form
ACID	Atomicity, Consistency, Isolation and Durability
AP	Availability and Partition Tolerance
API	Application Programming Interface
BASE	Basically Available, Soft State and Eventual Consistency
C2C	Consumer to Consumer
CA	Consistency and Availability
CAP	Consistency, Availability and Partition Tolerance
CDBMS	Cloud Database Management System
CL	Consistency Level
CP	Consistency and Partition Tolerance
CQL	Cassandra Query Language
DAS	Data Access Server
DBMS	Database Management System
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
Id	Avatar's or Game Object's UUID
MMOFPS	Massively Multiplayer Online First-person Shooter Game
MMOG or MMO	Massively Multiplayer Online Games
MMORG	Massively Multiplayer Online Racing Game
MMORPG	Massively Multiplayer Online Role Playing Games
MMORTS	Massively Multiplayer Online Real-time Strategy Game
MMOSG	Massively Multiplayer Online Social Game

N-TSModel	NodeAwarePolicy Integrating with Timestamp-based Model
NPC	Non Player Character
NTP	Network Time Protocol
OLTP	Online Transaction Processing
P2P	peer-to-peer
PaaS	Platform as a Service
PC	Player Character
RDB	Relational Database
RDBMS	Relational Database Management System
SaaS	Software as a Service
SNS	Social Networking Service
SQL	Structured Query Language
SSH	Secure Shell
T-TSModel	TokenAwarePolicy Integrating with Timestamp-based Model
TS	Timestamp, the Last Checkpointing Time, or Version ID
TSModel	Timestamp-based Model
TST	Timestamp Table
UDT	User-defined Type

1. Introduction

Massively Multiplayer Online Games (MMOs or MMOGs) have become more and more popular over the past decade [FBS07]. In this kind of game, hundreds or thousands of players from all over the world are able to play together simultaneously. In a virtual game world, players are allowed to choose a new identity, establish a new social network, compete or cooperate with other players, or even make some of their dreams that cannot be fulfilled in their real life come true [APS08]. This kind of game is so popular and interesting that in 2013 there are already 628 million MMO players worldwide [She13] and many players are even addicted to it [HWW09]. The game industry is developing rapidly, and the global PC/MMO games market is expected to grow at a compound annual growth rate of 7.9% between 2013 and 2017. In 2014 MMOGs have generated a total of $17 billion in revenues [New14].

As a typical and the most famous type of MMOGs, Massively Multiplayer Online Role-Playing Games (MMORPGs) develop rapidly. Every year at the Electronic Entertainment Expo (E3) (the largest annual commercial exhibition of the global video game industry) many upcoming MMORPGs are revealed by game publishers [IGN15]. There are some examples of the most popular MMORPGs, like *World of Warcraft*, *Aion*, *Guild Wars* and *EVE Online*. The difference between MMORPGs and other MMOGs, such as first-person shooters and real-time strategy games, is that MMORPGs have popularized the term *Persistent World* [ZKD08], which describes a virtual environment that continuously exists and changes, no matter whether millions of users, only few users, or even none at all interact with it [ILJ10]. In addition to the account information, the state data of objects and characters must be recorded on the server side in real-time, so that players can quit and continue the game at any time. Player behaviors in the game will be monitored and backed up in order to maintain the order of the virtual world. Furthermore, MMORPGs usually have more concurrent players than any other MMOGs, (for example, *World of Warcraft* has millions of concurrent players), which also exacerbates the burden of managing data.

From a computer science perspective, these *Persistent Worlds* represent very complex information systems consisting of multi-tiered architectures of game clients [KVK+09], game logic, and game data management which typically implement application-specific patterns of partitioning/sharding, replication, and load-balancing to fulfill the high requirements regarding performance, scalability, and availability [WKG+07, FDI+10]. For this reason, a qualified database system for data persistence in MMORPGs must guarantee the data consistency and also be efficient and scalable [ZKD08]. However, existing conventional Database Management System (DBMS) cannot fully satisfy all these requirements simultaneously [WKG+07, Cat10]. Therefore, with an increasing data volume, the storage system becomes a bottleneck, and solving scalability and availability issues becomes a major cost factor and a development risk.

In the last decade, Cloud data management has become a hot topic. The recently developed Cloud database management systems (CDBMSs), such as Cassandra, HBase and Riak, are designed to support highly concurrent data accesses and huge storage, which can easily meet the challenges mentioned above (e.g., efficiency, scalability and availability), thereby becoming a solution [Cat10]. Nevertheless, CDBMSs are generally designed for Web applications that have different access characteristics than MMORPGs, and require lower or various consistency levels [Aba09]. In other words, features of CDBMSs and RDBMSs are complementary, so they cannot replace each other. For this reason, if we hope to use a CDBMS to solve this issue, the following factors must be considered:

Scalability and performance: the advantages of an easily scalable data management solution for the development and maintenance of an MMORPG are obvious, but some specific requirements like partitioned game logic servers, real-time requirements, and very specific workloads (e.g., write intensive phases of checkpoints) do not easily fit with what these systems were developed for and raise the question of how the systems could be used optimally.

Consistency and availability: some data sets in MMORPGs require a high-level of consistency, such as account and game state data. In accordance with the CAP Theorem [GL02] consistency in Cloud data management is either very loosely defined (e.g., eventual consistency) or must be traded versus availability and/or performance. The latter may not be an option for MMORPGs.

1.1 Goal of this Thesis

Within the *CloudCraft* project we address the question of how to take advantages of Cloud data management solutions while finding ways to address their shortcomings for this class of applications.

For this purpose, we have organized our research as follows: firstly we have analyzed the typical architecture and data management requirements of MMORPGs; secondly,

we have classified data in MMORPGs into four data sets according to the data management requirements (e.g., data consistency, system availability, system scalability, data model, security, and real-time processing), namely account data, game data, state data, and log data; then, we have proposed to apply multiple data management systems (or services) in one MMORPG, and manage diverse data sets accordingly. Data with strong requirements for data consistency and security (e.g., account data) are still managed by an RDBMS (relational database management system), while data (e.g., log data and state data) requiring scalability and availability are stored in a Cloud data storage system; to evaluate and improve the scalability and performance of the new Cloud-based architecture, we have implemented a game prototype based on an open source MMORPG project that we ported to run on Cassandra, one of the most popular CDBMS. Furthermore, we have developed an environment to run simulated, scripted interactions of many clients with many game logic servers as well as Cassandra nodes. Based on observations made within this environment, we illustrate approaches to efficiently use the scaling capabilities; we have also analyzed the data consistency requirements in MMORPGs so as to achieve the required consistency-level for each data set in our game prototype. We found that the guarantee of high-level consistency in Cassandra is not efficient. So we have proposed to use a timestamp-based model as well as a *NodeAwarePolicy* strategy to solve it; furthermore, we have implemented a testbed using a conventional RDBMS (MySQL Cluster) to compare the performance with our new Cloud-based prototype. Experimental results have demonstrated the feasibility of our proposals.

1.2 Structure of the Thesis

This following of this thesis will be structured as follows:

Chapter 2 Background: In this chapter, we will give a brief introduction of MMORPGs, including their typical architecture, characteristics and diverse data sets. Furthermore, the typical databases used in MMORPGs and their limitations will also be discussed.

Chapter 3 Cloud Storage Systems: Using NoSQL DBMSs to manage data is a hot topic recently. In this chapter, we will compare them with the conventional RDBMSs, and introduce Cassandra in detail.

Chapter 4 Cloud Data Management for MMORPGs: In this chapter, we will discuss the feasibility of introducing Cloud-based technologies in MMORPGs.

Chapter 5 Using Cassandra in MMORPGs: Cassandra is not designed for online games, so there are some issues to be addressed. We will analyze its limitations in the game scenario, and propose some solutions for them.

Chapter 6 Evaluation: We have carried out some experiments to prove our proposal of a Cloud-based architecture for MMORPGs. In this chapter we will show some interesting experimental results and discuss them.

Chapter 7 Conclusion: In this chapter we will give a conclusion and summary of our research.

Chapter 8 Future Work: Some limitations and possible improvements of this thesis work are outlined in this chapter.

2. Background

> This chapter shares the material with the DASP article "Cloud Data Management for Online Games: Potentias and Open Issues" [DSWM13].

In this chapter, we will introduce some foundations of MMORPGs, like their typical architectures, various data sets, data management requirements for each data set, and the limitations of RDBMSs in terms of managing data in MMORPGs.

2.1 Massively Multiplayer Online Role-Playing Game

Before we introduce MMORPGs in detail, we will first distinguish it from other types of MMOGs.

2.1.1 Brief Introduction of Massively Multiplayer Online Games

MMOG is a kind of video game, which has a capability to support a large number (thousands) of players simultaneously. MMOGs are usually built on game servers and played over a network, like the Internet. Players only need to download and run a client software (client-server model) or use a browser (browser-server model) to play these games. For this reason, MMOGs can be found on most network-capable platforms, such as the personal computer, smart phone, tablet, video game console, and so on. To enhance the game's interactivity online players are enabled to communicate with each other, which sometimes are from all over the world.

MMOGs are generating huge revenue for game publishers ever year. A report from *NEWZOO* shows that the revenue of PC (Personal Computer)/MMO gaming worldwide in 2014 is $24.4 billion, and it also predicts that the PC/MMO online games market will reach $30.7 billion in 2017, accounting for 31% of total global games market revenues [New14].

MMOGs include a variety of gameplay types, such as:

Massively multiplayer online role-playing game (MMORPG) : is the most common type of MMOGs. The game usually takes place in a fantasy game world. A player assumes the roles of one or more characters (also called avatar) and controls it/them to explore and change the game world. The core purpose of the game is to develop the player's avatar. Therefore, a player needs to pay for the virtual currency and/or complete some tasks assigned by the game system, other players, or the guild in order to earn experience points, reach the level of her/his avatar, upgrade gaming equipment, or buy some virtual items. Some popular MMORPGs include *World of Warcraft*, *Aion*, *Guild Wars* and *EVE Online*.

Massively multiplayer online real-time strategy game (MMORTS) : allows players to set up their own army in the game. In order to maintain and expand their territory, to obtain more resources to strengthen their armies, players need to lead their own armies to go into the battle with enemies (other players' armies) in real-time. This kind of game includes *League of Legends*, *Shattered Galaxy* and *Age of Empires Online*.

Massively multiplayer online first-person shooter game (MMOFPS) : usually simulates a great battle, in which a large-scale of players found a team to fight with enemies. The display diver can simulate the first-person perspective of an avatar. Through this perspective, players observe objects in the game, and react to them accordingly like shooting, jumping, moving, chatting, and so on. Here is a list of MMOFPSs: *MAG*, *Firefall*, *World War II Online* and *Planetside*.

Massively multiplayer online racing game (MMORG) : is a kind of racing-themed game. *Trackmania*, *Kart Rider* and *Crazyracing Kartrider* belong to this kind of game.

Massively multiplayer online social game (MMOSG) : focus on building a new social relationship in a virtual world. Players can even get married and have children there. *Second Life* is an example of MMOSGs.

2.1.2 Characteristics of MMORPGs

In this project (*CloudCraft*), we only focus on MMORPGs for the following reasons:

Most popular MMOGs: MMORPGs have most of the subscribers of MMOGs, thereby occupying most of the Subscription-based MMOG market share (*World of Warcraft* occupies 36% in 2013) [Sup14]. It is not only a subset of MMOGs, but also a symbol of these games. For this reason, although games like MMOSGs have many common features with MMORPGs, we still decided to focus on the latter.

A large number of concurrent players: An MMORPG distinguishes itself from the others by allowing a very large number (thousands) of players to interact with each other in one virtual shared game world. *EVE Online* even supports several

hundred thousand players on the same server, with over 60,000 playing simultaneously at certain times [com10]. In contrast, games like MMOFPSs and MMORGs always divide concurrent players into a large number of groups with only a limited number of participators (e.g., 256 players in *MAG* [1UP08]), and disperse them on separated battlefields hosted by different game servers. Only players in the same group can communicate or combat with each other. This characteristic of MMORPGs requires a high-performance framework of the game server to handle a large number of concurrent requests.

Unpredictable number of active players: The number of active players in MMORPGs changes frequently and strongly with the launch of a new expansion of the game or affected by the gaming experience. An example is that the number of subscribers to *World of Warcraft* peaked in excess of 12 million in October 2010, but it had decreased to 6.8 million in June 2014. Then, in November 2014 coincided with the release of a new expansion the subscriptions has passed 10 million again [IGN14]. Therefore, the MMORPG framework needs to have the ability to flexibly cope with the data management burden caused by the surge in the number of players, and deal with the issue of the waste of resources caused by the plummeted number of players.

Complex gameplay: Different with MMOFPSs and MMORGs, which only simulate a particular scene (shooting or racing) in the life, MMORPGs is trying to create a new virtual world. That means, a variety of scenarios in the real world are presented in the game, such as trading, chatting and combating. In other words, an MMORPG system contains many subsystems, the data management requirements of which are total different. Thus, MMORPGs are more difficult to develop, and the experience gained in the development can be applied to other MMOGs.

Persistent world: Some MMOGs (e.g., MMOFPSs, MMORGs and MMORTSs) usually divide the virtual game world into small sessions. That means, the game world will be reset after reaching a limited time or completing a system specified target. For example, in MMORTSs a match will be terminated after the car driven by a player has reached the final line. Then, this player needs to wait for other players to finish and start a new match. In this case, players can neither save their records during the game, nor continue to complete a task when they log in the game again. Effects of players on the game world will disappear with the end of a session. Therefore, only player's account information must be stored on the server side. An MMORPG distinguishes itself from other MMOGs by keeping the virtual game environment running even when players are offline [APS08]. In other words, they provide a persistent game world. A player can save her/his records at any time, and continue the game as if she/he never leaves. Hence, not only the account information of a player, but also the state information of the game world, player's avatar and non player character (NPC) must be persisted on

the game server. In order to maintain such a virtual world, a complex information system consisting of a multi-tiered architecture is required.

Big data management: As mentioned above, MMOGs host game worlds on the server. Players must log in to a remote server before starting a game. Therefore, different with local area network games and single player games, MMOGs record game logic, game world metadata, players information on the server side. Note that the type and size of information stored by MMOG systems are differently. We find that MMORPGs are facing the management issue of big data, which can be tracked back to two sources:

1) Unlike other types of MMOGs, MMORPGs must also persist avatar's state information on the server, which could contain around one hundred attributes (e.g., avatar's basic information, skills, inventory, social relation, equipment, tasks and position) and lead to a complex database schema. Since an MMORPG could have millions of players, the data size of such information is very large. Furthermore, the state information is modified and retrieved constantly, and the parallel requests from players must be responded in real-time during the game.

2) Moreover, player behavior analysis is essential for game publishers to help them understand the current game status, fix bugs in the current edition of game, guide the development of the future expansion, and detect cheating in the game [SSM11]. For instance, the number of average concurrent users has been monitored to evaluate the popularity of a game; the income and expense of players in the game have been analyzed for holding the economic equilibrium of the game world and curbing inflation; task statistics could help for setting the difficulty of a game. As a result, almost all the behaviors of players as well as system logs need to be persistent on the server, which increase continuously. For example, *EverQuest 2* stores over 20 terabytes (TB) of new log data per year [ILJ10].

Overall, development of an efficient game system to manage and analyze big data is particularly crucial for MMORPGs, which has become a challenge [ILJ10].

2.1.3 Analysis of Current Data Management for MMORPGs

Modern database technologies could be used in MMORPG systems to solve the data management issue mentioned above. But firstly we need to understand the architecture of a game system.

2.1.3.1 A Typical Architecture of MMORPGs

It is not easy to get the knowledge of a practical architecture of an MMORPG in detail because game companies always protect it to avoid their game servers from attacks. Furthermore, the architecture of MMORPGs is significantly influenced by the workload

of the game server, and consequently the implementation of each MMORPG system is individual. However, we can still find some common characteristics and components, namely applying three tiered architecture (see Figure 2.1) [CSKW02, Bur07, WKG+07].

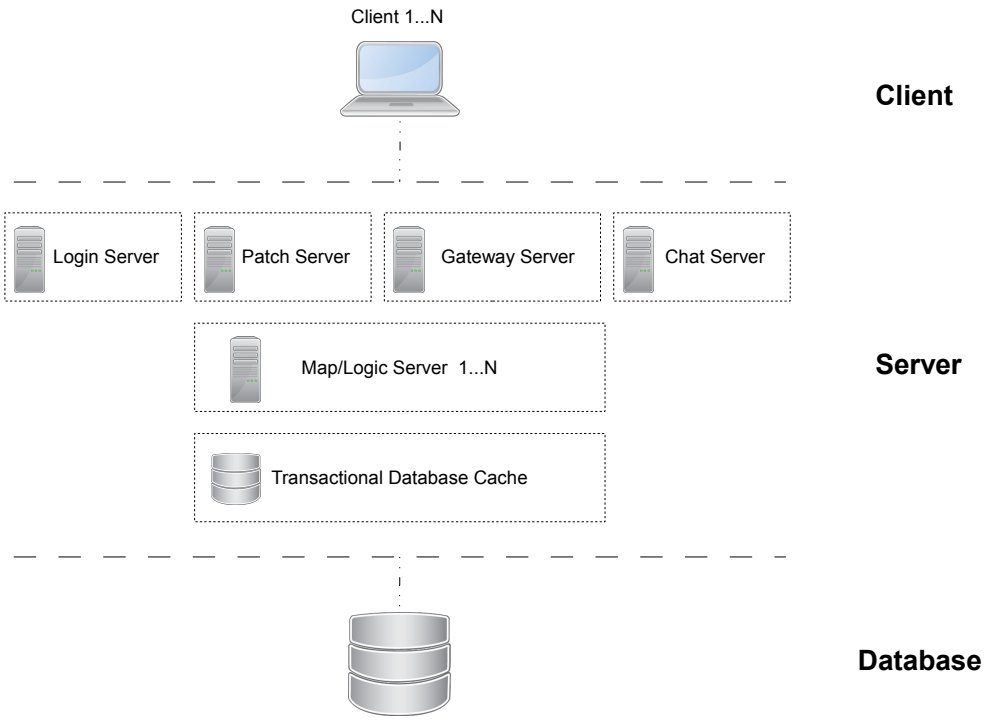


Figure 2.1: Three Tiered Architecture

A game client could be a software/browser on a PC or an application on a smart phone. A client device does not require a strong computing power other than a powerful graphics processing capability. It is an interface of the game, which is responsible for communicating with the game server, computing local game state and displaying the game screen. A game client almost never makes any authoritative decision (e.g., whether a monster is within the valid attacking range of a player) because it could be reverse-engineered, hacked and modified so as to cheat at the game [WKG+07]. Instead, MMORPGs always take a centralized authoritative game server for arbitration.

The game server tier mainly consists of six components, namely login server, patch server, gateway server, chat server, map/logic server and transactional database cache. Map/logic server is a vital component, which runs the simulation engine, maintains the game rules, detects cheating, and spreads the computation result to the corresponding players synchronously [LKPMJP06]. Each map/logic server typically hosts one geographic zone of the game world instance. If a player crosses from one geographic zone to another, the state of her/his character will also be transferred to another server au-

tomatically. Accordingly, players can only communicate, interact and influence others in their geographic vicinity [WKG+07].

Since the game server needs to handle connecting requests from global players, Map/-logic servers are deployed on a large number of machines [AT06](e.g., *World of Warcraft* has over 10,000 servers [NIP+08]), which are usually distributed in several data centers (zone server) around the world to reduce latency.

In this case, a gateway server is required to maintain the game session of a player, besides monitoring requests of the player, forwarding them to a zone server, sending back results to the player, and protecting the map/logic server from attacks. In other words, it is an information transfer station, which is also composed of series of servers.

A single map/logic server usually can only support around 2000 concurrent players [KLXH04]. For the purpose of spreading the load out and extending the number of players on each server, game developers often separate some non-core functions from the map/logic server. Therefore, servers like the login server, patch server and chat server are also necessary in a game server system. A login server is responsible for determining the validity of a player's identity. A patch server checks and updates game data stored at the game client side. A chat server handles the chat messages between players.

Transaction management is a technical challenge in MMORPGs because data corruption cannot be handled just by ending a game session. Additionally, the incomplete or failed transaction must be rolled back. MMORPGs, consequently, manage the game state in an authoritative storage system to provide persistent game worlds. For this reason, the database plays an important role in this kind of game, and MMORPGs are more like traditional database applications comparing with other MMOGs [WKG+07].

From Figure 2.1 we can see that the game server does not retrieve game state data directly from a disk-resident database, but through a transactional database cache (e.g., an in-memory database or a real-time database [Hva99]). There are the following reasons [Ale03]: Firstly, it cannot cope with the heavy I/O workload of an MMORPG with millions of players. Even using some advanced commercial databases like Oracle and SQL server, the transaction rate cannot satisfy the game requirement, and it cannot be improved by simply adding more machines [WKG+07]; secondly, executing transactions in a disk-resident database will inevitably pause the game. This pause may occur at any time, which cannot be tolerated by a real-time system; finally, game providers need to invest large amounts of money to buy hardware in order to achieve the theoretical performance, which unfortunately can only be predicted in the late stage of the game development. However, if the architecture has to be changed at that time, the development of the game will be delayed as well.

The throughput of an in-memory database satisfies the data management requirement of an MMORPG, consequently, it executes real-time operations in the game instead of the disk-resident database. However, an in-memory database cannot guarantee the data persistence when it fails, and its storage costs are high. Therefore, most MMORPGs

have a transactional database cache sited in front of disk-resident database [WKG+07]. That means, the required data needs to be loaded in a cache, which handles players' requests in the form of transactions and then writes data back into the database asynchronously (e.g., every five or ten minutes).

Typically, each data center hosts its own unique state database. A player has to execute a character migration operation before she/he moves to another zone server. This complex data migration process may take a long time [BAT15].

2.1.3.2 Classification of Data Sets in MMORPGs

Not all data in an MMORPG have to be cached on the server side. Actually, MMORPGs always manage diverse data sets accordingly, thereby separating them in different places or databases. We have classified data in the game into four sets so as to understand the process of data accessing in MMORPGs.

Account data: This category of data includes user account information, such as user ID, password, recharge records and account balance. It is usually only used when players log in to or log out of a game for identification and accounting purposes. Compared with transferring game state, the requirement for shortening the response time of retrieving account data is not that urgent, hence the use of a database cache is not necessary.

Game data: Data such as the world geometry and appearance, object and NPC metadata (name, race, appearance, etc.), game animations and sounds, system logs, configuration and game rules/scripts in an MMORPG are generally only modified by game developers. Game client side often keeps a copy of part of game data (e.g., world geometry and appearance, game animations and sounds) to minimize the network traffic for unchangeable data. The game server (patch server) will update the copy when a client connects it.

State data: PC (Player Character) metadata, inventory, position as well as state of characters (PC and NPC) and objects in MMORPGs are modified constantly during gameplay. As motioned above, modifications of state data are currently executed by an in-memory database in real-time and backed up to the disk-resident database periodically.

Log data: Data like player behaviors, chat histories and mails are also persisted in the database, and are rarely modified. This kind of data is not used in a game session, thereby eliminating the need to be preloaded. It is not suggested storing them together with state data in the same database since collecting of log data brings the database unnecessary burden and consumes the database capacity rapidly.

2.1.3.3 A Sample Session in an MMORPG

We integrate all pieces of information and get a typical architecture of MMORPGs (see Figure 2.2). Now we will walk through a sample session in an MMORPG [WKG+07]:

A player connects the game server through a client, and sends a login request to the login server, which is responsible for determining its validity. The login server cooperates with an account database that stores user account information. If the validation is passed, the login server encrypts the user ID, generates a token, and returns it to the player. The client then updates the game data stored on it from a patch server, which gets the data from a game database. When the update is complete, the client uses the token to communicate with a gateway server, which will assign it to a zone server. Note that only one zone server provides services throughout an entire session. The zone server will load the player's state information from the state database into the cache, determine physical location of her/his avatar in the game world, assign it to a logic/map server accordingly, and render a copy of the state information to the client. Once the assignment is successful, the player can start the game, chat (through a chat server) and interact with all other players on the same server. The computation of the interaction occurs on the server or in the database cache, and then the result will be rendered at the client. The updated state information will be backed up to the state database periodically. The log database records all player actions (including the reset actions) and the chat history. When the player quits the game, her/his final state information will be persisted in the state database and then removed from the database cache (see Figure 2.3).

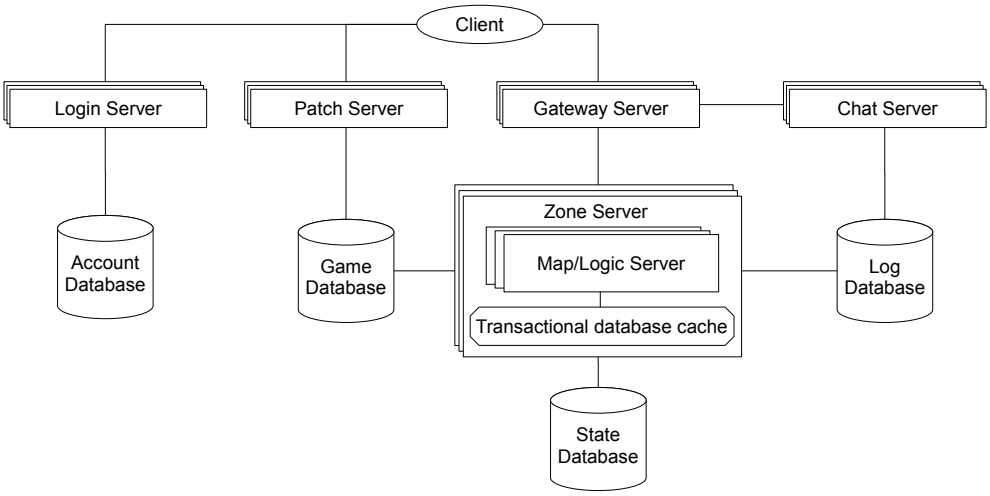


Figure 2.2: Principle Architecture of MMORPGs [WKG+07]

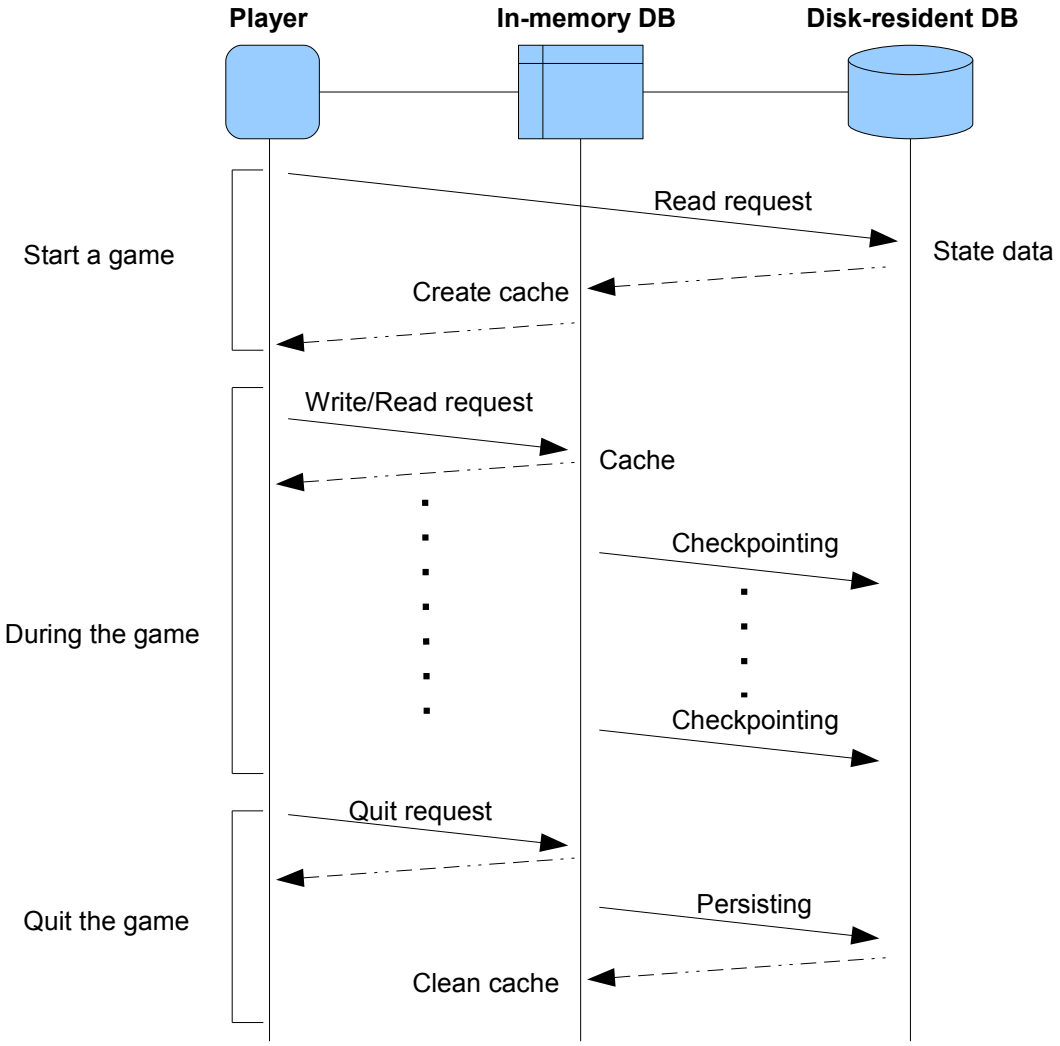


Figure 2.3: Management of State Information of an Avatar in MMORPGs

2.2 A Typical Database System in MMORPGs

Currently, popular MMOPRGs mainly apply a distributed RDBMS to persist data (e.g., Second Life and World of Warcraft both run on MySQL [DBM09], Guild Wars runs on Microsoft SQL Server [DBM07]), which can commit complex transactions and are proven to be stable.

A relational database (RDB) implements the relational model. It maintains varying amounts of tables, which are related by primary keys and foreign keys. Using SQL statements, database developers and administrators maintain a database, and manipulate/-query data items. There are many tools to maintain and optimize a relational database system (RDBS). For instance, SQL Server Management Studio and Visual Studio support Microsoft SQL Server. Overall, an RDB meet all requirements of MMORPGs in terms of data manipulations.

With increasing business requirements, centralized systems can hardly meet the needs such as run-time performance and reliability. Distributed database systems can solve these problems by increasing data redundancy and the number of computers. Distributed databases is distributed in the physical layer, but organized in the logical layer. Each computer has a complete copy of the DBMS, and a local database. For users, they still consider themselves operating a single database system.

As an example, consider MySQL Cluster [Orab] and its characteristics. MySQL Cluster adopts a shared-nothing architecture to ensure scalability. In order to balance the workload among nodes, it automatically partitions data within a table based on the primary key across all nodes. Each node is able to help clients to access correct shards to satisfy a query or commit a transaction. For the purpose of guaranteeing availability, data are replicated to multiple nodes. MySQL Cluster applies a two-phase commit mechanism to propagate data changes to the primary replica and a secondary replica synchronously, and then modifies other replicas asynchronously. In this case, at least one secondary replica has the consistent and redundant data, which can be used as a fail-over server when the primary server fails. MySQL Cluster also writes redo logs and checkpoint data to the disk asynchronously, which can be used for failure recovery. When MySQL Cluster maintains tables in memory, it can support real-time responses.

However, as we mentioned in the introduction, RDBMS cannot fulfill some requirements (e.g., scalability) of MMORPGs well, and considerable efforts have to be spent on fulfilling these requirements on other levels [Cat10]. In the following sections, we will discuss this in detail.

2.3 Data Management Requirements of MMORPGs

MMORPG applies a complex game server to manage various data sets, which have distinct data management requirements. We have summarized them in Table 2.1 and explained them in the following.

Support for different levels of consistency:

In a collaborative game, players interact with each other. Changes of state data must be synchronously propagated to the relevant players within an acceptable period of time. For this purpose, we need a continuous consistency model in MMORPGs [LLL04]. Changes of the state data and account data must be recorded in the database. It is intolerable that players find that their last game records are lost when they log in to the game again. As a result, a strong or at least Read-Your-Writes consistency [Vog09] is required for such data. However, strong consistency is not necessary for log data and game data. For example, the existence of a tree in the map, the synchronization of a bird animation, or the clothing style of a game character is allowed to be different among client sides. Log data are generally not analyzed immediately. Hence, eventual consistency [Vog09] is sufficient for these two classes of data.

	Account Data	Game Data	State Data	Log Data
Consistency	★ ★ ★	★ ★	★ ★ ★	★ ★
Runtime performance	★ ★	★	★ ★ ★	★ ★
Availability	★ ★	★ ★ ★	★ ★ ★	★ ★
Scalability	★	★	★ ★ ★	★ ★ ★
Partitioning	★ ★	★ ★	★ ★ ★	★ ★
Flexible model	★	★	★ ★ ★	★
Simplified processing	★	★ ★	★ ★ ★	★ ★ ★
Security	★ ★ ★	★	★ ★	★ ★ ★
Re-usability	★ ★ ★	★ ★ ★	★ ★ ★	★ ★ ★

Table 2.1: Data classification and analysis of their requirements

Performance/real-time:

State data are modified constantly by millions of concurrent players, which brings a significant traffic to game servers (one player sends an average of six packets per second [CHHL06]), thereby generating thousands of concurrent database connections. These commands must be executed in real-time (within 200ms [CHHL06]) and persisted on the disk efficiently, which have become a challenge to the database performance.

Availability:

As an Internet/Web application, an MMORPG system should be able to respond to the request of each user within a certain period of time. If lags or complete denial of services appear frequently, this will significantly decrease the acceptance of the game and will result in sinking revenues. Availability can be achieved by increasing data redundancy and setting up fail-over servers.

Scalability:

Typically, online games start with a small or medium number of users. If the game is successful, this number can grow extremely. To avoid problems of a system laid out for too few users or its costs when initially laid out for too many users, data management needs to be extremely scalable [GDG08]. Furthermore, log data will be appended continually and retained in the database statically for a long time [WKG+07]. The expansion of data scale should not affect the database performance. Hence, the database should have the ability to accommodate the growth by adding new hardware [IHK04].

Data sharding:

Performing all operations on one node can simplify the integrity control, but that may cause a system bottleneck. Therefore, data must be divided into multiple nodes/shards

in order to balance the workload, process operations in parallel, and reduce processing costs. Current sharding schemes are most often based on application logic, such as partial maps (map servers). This does not easily integrate with the requirement of scalability, i.e., re-partitioning is not trivial when new servers are added. Accordingly, suitable sharding schemes are a major research issue.

Flexible data model:

A part of data in the game like state data do not have a fixed schema, for example, PCs have varying abilities, tasks, and inventory. Additionally, MMORPGs are typically bug-fixed and extended during their run-time. Therefore, it is difficult to adopt the relational model to manage such data. A flexible data model without a fixed schema is more suitable.

Simplified processing:

In MMORPGs, only updates of account data and part of state data must be executed in the form of transactions. In addition, transaction processing in online game databases is different from that in business databases. For example, in MMORPGs, there are many transactions, but most of them are of small size. Parallel operations with conflicts occur rarely, especially in the state and log database, which are responsible for data backup. There are no write conflicts among players in these two databases because they have already been resolved in the transactional database cache. There might be a write conflict from one player, which is usually caused by network latency or server failure, and can be addressed by comparing the timestamp of the operation. Using locks as in a traditional database increases the response time. Additionally, deadlock detection in a distributed system is not easy. Hence, a simplified data processing mechanism is required.

Security:

Game providers have to be concerned about data security because data breach may lead to economic risks or legal disputes. For this reason, user-specific data, such as account data and chat logs, must be strongly protected. Furthermore, it must be possible to recover data after being maliciously modified.

Ease of use, composability and re-usability:

The data management system should be easy to use by developers and apply for various MMORPGs. Companies developing and maintaining MMORPGs should be able to re-use or easily adapt existing data management solutions to new games, similar to the idea separating the *game engine* from the *game content* currently widely applied. An interesting solution would be game data management provided as a service to various providers of MMORPGs, but this strongly depends on building up trustworthy services.

2.4 Limitations of RDBMS in MMORPGs

Traditional RDBMSs are normally easy to use, powerful, stable, efficient, and have already gained a large number of successful practical experiences. When the traffic of a web application is not too heavy, a single database can easily cope with it. In recent years, with the rapid development of the Internet, blog, SNS (social networking service) and Microblogging gradually become popular. With the increase of web traffic, almost the majority of web sites using the RDBMS architecture began to face database performance problems. For this reason, developers began to extensively use database sharding, replication and caching technique, and optimize the table structure and index. However, when the data size grows to a certain stage, scalability problems have emerged. In the following we discuss limitations of RDBMS in MMORPGs in detail:

Hard to scale out:

When the workload of a node in a database cluster is too heavy, it will become a bottleneck of the system. Typically, there are two solutions, namely vertical (scale up) and horizontal scaling (scale out). Vertical scaling refers to add resources to a single node in the system, such as upgrading CPUs and extending memories and disks. This method can effectively improve the performance of a single node. However, it consumes a large amount of money to buy advanced equipment. If the game becomes unfortunately less popular and correspondingly the workload of this node goes down, the new added resources will be idle. Moreover, sometimes even if the most powerful equipment has been used, the issue could still exist. For these reasons, in MMORPGs we typically use the horizontal scaling method, which means adding more nodes to the cluster in order to spread the loads. Accordingly, data in the database will be partitioned (vertically/horizontally) into several logical fragments, replicated several times (to avoid single node failure), and distributed across multiple nodes. While applying the relational model, a database has to take into account many things like maintaining data integrity (e.g., primary key constraint, foreign key constraint or value range constraint) and managing distributed transactions by locking (e.g., two-phase commit protocol) [SMA+07, Vol10]. All of them would block the concurrent execution of transactions, thereby decreasing the rate of transactions per second. And the more nodes and replicas exist in the cluster, the greater the impact would be. For this reason, even though some RDBMSs (e.g., MySQL Cluster and Oracle RAC) have used a shared-nothing architecture, the scalability of them has still been criticized for a long time [FMC+11, Cat10].

Complex database schema:

An RDBMS implements the relational database model and complies with Codd's 12 rules [Cod85b, Cod85a], which results in a complex database schema and a low system performance. Data in the database is structured as a set of relations (tables in SQL), which are connected by references (foreign keys). Hence, there are many integrity constraints to follow when modifying data. For example, we cannot delete the value of an attribute, if another table refers to it. That means, modification of data may

comply with multiple tables, which is not efficient; to reduce data redundancy RDB normally complies 3NF (third normal form). In this case, to obtain the result of a query the system has to perform a join operation across several tables, which might be distributed in different nodes, or compute values of multiple attributes, both of which will increase the response time.

Fixed data structure:

The structure of a table, the data type as well as the value range of each column must be predefined in an RDBMS. The change of data structure (e.g., adding a new attribute or changing the type of an attribute) leads to modifications of all previous records in a table, and probably is not transparent to the application level. However, the life cycle of an MMORPG is bug-fixed. As a result, a game company would have to pause its game service for a while frequently in order to upgrade and maintain the game system, which would affect the gaming experience of players.

Strict consistency:

The RDBMS follows a transaction mechanism, thereby providing ACID guarantees (Atomicity, Consistency, Isolation and Durability) [HR83]. Furthermore, modification of data object needs to be synchronized to all replicas, or all requests for a data object need to be processed on a master/primary replica. As a result, the workload at the master/primary replica or the network traffic between nodes will become a bottleneck of the system. However, as we discussed above, strict consistency is not necessary for all data sets in MMORPGs. For some use cases (e.g., backup of state data and log data), the overhead of this ill-suited consistency mechanism becomes a drawback.

Low availability:

The RDBMS, nowadays, provides solutions to improve availability by replication and fail-over within the context of distributed databases. Nevertheless, these properties are not trivial to implement and maintain for a given application, especially considering the large scale of MMORPGs.

High cost:

The number of mature and commercial RDB products is not too much (e.g., Oracle, SQL Server, DB2, MySQL and Teradata). Furthermore, many of them require to pay for an expensive license (except for MySQL). For example, there are three kinds of licensing models for using SQL Server 2014, namely per core ($14,256 for enterprise edition), Server + CAL ($8,908 for business intelligence edition) and per user ($38 for developer edition)[1]. The database cluster of an MMORPG is supported by a large number of computers with multi-core processors, and is maintained by numerous game

[1] http://www.microsoft.com/en-us/server-cloud/products/sql-server/purchasing.aspx (accessed 04.11.2015)

developers. Purchase of the license increases the cost of game development and operation. Furthermore, it is difficult to predict the scale of a game database cluster, which depends on the popularity of the game. Excessive purchase of the license is a serious waste. Additionally, maintenance costs of RDBMSs are high, which make many enterprises unbearable.

2.5 Summary

In summary, the RDBMS is powerful, but it cannot fit all application scenarios in MMORPGs. Issues like system scalability, big data management and change of table structure are challenges for web application developers using RDBMSs. In this context, the concept of NoSQL databases has been proposed in 2009. Web sites in the pursuit of high performance and high scalability, have chosen NoSQL technology as a priority option. NoSQL database has various types, but a common feature of them is that they have removed relational characteristic. There is no relationship between data, so it is very easy to extend. Hence, it also brings the scalability to the architecture level. In the next chapter we will highlight this kind of database.

3. Cloud Storage Systems

> This chapter also shares the material with the DASP article "Cloud Data Management for Online Games: Potentials and Open Issues" [DSWM13].

In this chapter, we will introduce Cloud storage systems, highlight the new challenges from managing big data, compare NoSQL stores and RDBs with their application scenarios, features and data models, and at last take Cassandra as an example to show the implementation of a NoSQL store in detail.

3.1 Cloud Computing

With the rapid development of the Internet industry, release, exchange, collection and processing of information more and more depend on the network. Information on website is no longer static as in earlier years. The Internet functionality and the network speed have been improved greatly. Now, the Internet has become an indispensable part of people's life.

Given this background, Internet companies are facing unprecedented opportunities and challenges. Some websites have to deal with millions of concurrent visitors and thereby processing petabyte data [SLBA11], which could not be handled by conventional system architecture and DBMS efficiently. For instance, the social network website, Facebook, has over 1.71 billion monthly active users, who totally share 4.75 billion pieces of content and upload 300 million photos daily [Zep16]. As a result, a new advanced information technology, Cloud computing, has been proposed.

Cloud computing combines IT and Internet technology, so as to obtain the super computing and storage capacity [AFG+10]. By using it, shared hardware and software resources as well as information could be provided to computers and other devices on demand. Accordingly, IT companies can improve the utilization rate of their existing resources (e.g., storage space and computing power) by using Cloud technologies,

and even rent such resources from a Cloud provider, which reduces their operational costs [KKL10, KK10].

Cloud computing offers three kinds of service models [Kav14], namely infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS)(see Figure 3.1).

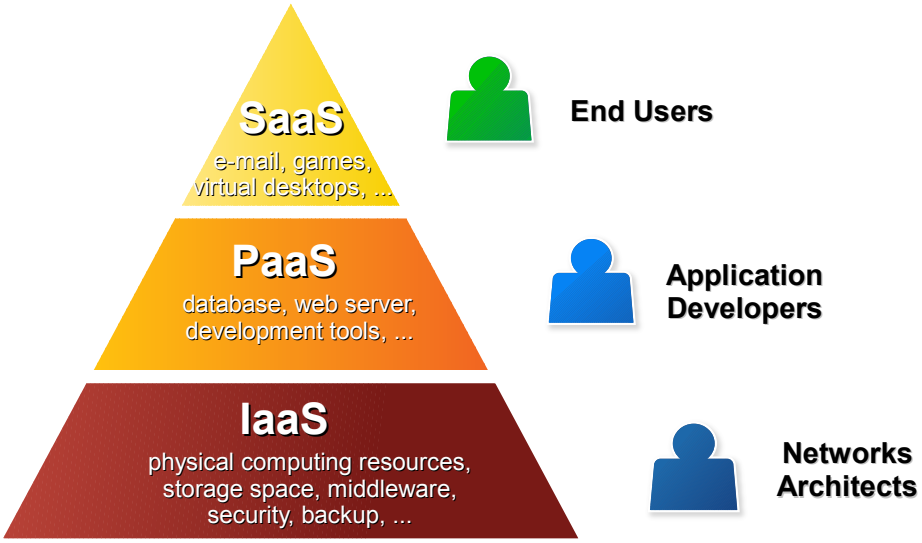


Figure 3.1: Three Service Models [Vah14]

IaaS: over the Internet, consumers (e.g., networks architects) obtain infrastructure services like physical computing resources, storage space, middleware, security, backup, etc.

PaaS: consumers (like application developers) get a development environment service (e.g., database, web server, development tools) from a Cloud provider.

SaaS: consumers through the Internet obtain software services, such as e-mail, virtual desktops or games. Typically, users do not need to buy the software, but lease the right to use a web-based software from the providers by getting an account name and a password.

This project aims at building a suitable game data management system. For this reason, in the next section we focus on introducing Cloud data management in detail.

3.2 Cloud Data Management

We can classify data into three types, namely structured data, unstructured data and semi-structured data. Structured data refer to any data that could reside in a fixed structure such as a two-dimensional table structure, like a row of a table in an RDB. Unstructured data are all data objects that could not fit into any fixed two-dimensional table, such as texts, photos, graphics, videos and webpages. Semi-structured data are between these two types of data. They have generally a self-describing structure, which mixes tags/markers and information together. XML and HTML are forms of semi-structured data. These three kinds of data are managed separately by different services in the Cloud. Normally, we use Cloud storage service to persist unstructured data, and use Cloud database to store structured data. Semi-structured data could be managed by both of them.

3.2.1 Cloud Storage and Cloud Database

Cloud storage: is an online storage service (belongs to IaaS model), which is a new concept extending from the concept of Cloud computing. Specifically, it refers to the data storage on virtual servers offered by a third party [Roe11].

While applying a private storage system, we must clearly know its interfaces, transmission protocols, number of hard disks and their capacity. To ensure data security and business continuity, we also need to establish the appropriate data backup systems and disaster recovery systems. In addition, periodically status monitoring, maintenance, hardware and software updates and upgrades for the storage devices are also necessary. While using the Cloud storage, everything mentioned above is no longer necessary for the user. All devices in a Cloud storage system are completely transparent to the user. An authorized user can access it anywhere via the Internet. Data are stored in the form of files in a distributed storage system like Hadoop Distributed File System [SKRC10].

Web 2.0 technology helps Cloud storage users to store, share, back up and recover data like texts, pictures as well as video and audio files easily via PCs, mobile phones and other mobile equipment. The software Dropbox[1] is a good example for a Cloud storage service.

Cloud database: [2] refers to a database running on a Cloud infrastructure (belongs to PaaS model). There are two development models: users could either deploy their own databases in a virtual machine image, or obtain a database service maintained by a Cloud database vendor. With a professional company for database maintenance and other benefits like storage integration, Cloud databases are economical (pay-on-demand, no hardware, software or licensing fees), safe, efficient, available, reliable, easy to use and so on. Therefore, recently Cloud databases have been

[1] https://www.dropbox.com/ (accessed 19.12.2015)
[2] https://en.wikipedia.org/wiki/Cloud_database (accessed 19.12.2015)

more and more applied to the domains of Web application, data analysis and data management.

There are two models using in Cloud databases, namely the SQL- and the NoSQL-based model. That means, users can get both database services from conventional RDBMSs (e.g., Oracle, MySQL, SQL Server and IBM DB2) and NoSQL DBMSs like Cassandra, HBase and MongoDB. In the next section, we will introduce the inevitability of the appearance of NoSQL stores.

3.2.2 Big Data Management

With the emergence of Web 2.0[3], instead of employees of a website, users become the major producers of web information. On the one hand, it greatly increases the richness and interactivity of web information. On the other hand, it brings the challenge of processing big data, for example, the big transaction data from C2C (Consumer to Consumer) websites like eBay, and the big interaction data from SNS websites like Facebook.

Big data usually have following properties: they have diverse types (Variety), such as structured, unstructured, semi-structured; the amount of these data is massive (Volume), which normally reaches the terabyte or petabyte level; the generation speed of these data is particularly fast (Velocity), so they must be processed in real-time. (It is called 3Vs properties.) Therefore, to manage the big data, a DBMS must have scalability, efficiency and high availability.

While using conventional RDBs, website developers have to face many insurmountable issues (large data processing, multiple data types management, system scalability, etc.). In the last chapter, we have analyzed the reason, which is due to the restrictions from the relational model. As a result, website developers have to explore new storage architectures to meet these challenges. Accordingly, a movement called NoSQL (non SQL) was launched in 2009[4] [SLBA11].

3.3 NoSQL Stores

In the late 1990s, Eric Brewer has described the CAP theorem, BASE and eventual consistency, which became later the theoretical foundation of NoSQL. In the following, we will introduce them in detail, and discuss NoSQL stores by comparing it with conventional RDBs.

3.3.1 CAP Theorem

The famous CAP Theorem is first presented as a conjecture by Eric Brewer, which was later proven to be true [GL02]. Although some researchers raise an objection to this

[3] https://en.wikipedia.org/wiki/Web_2.0 (accessed 03.12.2015)
[4] http://nosql-database.org/ (accessed 03.12.2015)

theorem, it is still followed by distributed databases like NoSQL stores. This theorem states that consistency (C), availability (A), and partition tolerance (P), at most two of them can be guaranteed simultaneously for a distributed computer system.

Consistency: all replicas in a distributed system keep the same value at any time.

Availability: each request can be responded within a period of time. (Even if the value is not consistent in all replicas, or just sends back a message saying the system is down.)

Partition tolerance: in the case of network partitioning (such as network interrupt or node failure), the system can continue to operate as it is complete.

The choice of CA could only be made when the system is deployed in a single data center, where partition occurs rarely. However, even if the probability of occurrence of the partition is not high, it is entirely possible to occur, which shakes the CA-oriented design. In the case of node failure, developers have to go back, and make a trade-off between C and A.

Current network hardware cannot avoid message delay, packet loss, and so on. So in practice, partition tolerance must be achieved in a cross regional system. For this reason, developers have to make a difficult choice on data consistency and availability.

Conventional RDBMSs are designed and optimized for OLTP (Online Transaction Processing) systems like banking systems, where inconsistent data may lead to erroneous computing results or customer's economic losses. Consequently, this kind of DBMS chooses to sacrifice system availability (CP type). When there is a network partition, a write request would be blocked due to the continuous attempt of connecting with the lost node.

Web 2.0 websites have many significant differences with OLTP systems:

Requirement for data consistency: many real-time Web systems do not require a strict database transaction. The requirement for read consistency is low, and in some cases the requirement for write consistency is also not high. Eventual consistency is allowed.

Requirement for write and read in real-time: the RDB ensures that a read request could immediately fetch the data after a successful insertion of a data item. However, for many web applications, such a high real-time feature is not required. For example, it is totally acceptable on Twitter that after posting a new Tweet, subscribers see it in a few seconds or even ten seconds.

Requirement for complex SQL queries, especially multi-table queries: any web system dealing with big data avoids joining multiple large tables and creating complex data analysis type of reports. Especially, SNS websites have avoided that

from the requirements of system functionality as well as the design of database schema. Usually there are only retrievals of primary key and queries with simple conditions within a single table. So complex SQL queries are not required.

Moreover, users of Web 2.0 websites expect to get 7*24 uninterrupted service [BFG+08], which unfortunately cannot be fulfilled by RDBs guaranteeing strong consistency. For these reasons, website developers have abandoned the SQL model and designed alternative DBs. Some NoSQL stores are developed to provide a variety of solutions to ensure the priority of system availability (support AP).

It is noticed that NoSQL DBMSs are typically designed to deal with the scaling and performance issues of conventional RDBs. In addition, their functionality highly depends on their specific application scenarios (not only Web 2.0 websites). Therefore, it does not mean all NoSQL stores (e.g., HBase) have dropped data consistency in favour of availability.

3.3.2 ACID vs. BASE

Conventional RDBs chose to stick ACID properties (Atomicity, Consistency, Isolation, Durability) of transactions [HR83].

Atomicity: a transaction is an indivisible work unit. Operations within one transaction are performed either all or none.

Consistency: at the beginning or the end of a transaction, the database must be in a consistent state. It is noteworthy that the concept of C in ACID is different with that in CAP and later in BASE. Here, C means all rules (data integrity) of a database, such as unique keys. In contrast, C in CAP only refers the state of a single replica, which is only a subset of ACID consistency [Bre12].

Isolation: a transaction cannot be interfered by other transactions. A data item cannot be accessed by other transactions until it is modified completely.

Durability: once a transaction is committed, changes of data should be persistent. Even a system failure will not affect them.

In contrast, in the NoSQL movement, developers have made a variety of programs giving priority to availability and following BASE, which is an acronym of Basically Available (BA), Soft state (S), and Eventual consistency (E) [FGC+97].

Basically available: NoSQL DBMS typically does not concern isolation, but system availability. In other words, multiple operations can simultaneously modify the same data. Hence, the system is able to respond any request. However, the response could be an inconsistent or changing state.

Soft state: data state can be regenerated through additional computation or file I/O. It is exploited to improve performance and failure tolerance. Data are not durable in disk.

Eventual consistency: the change of a data item will be propagated to all replicas asynchronously at a more convenient time. Hence, there will be a time lag, during which the stale data would be seen by users. In this project, three kinds of eventual consistency will be mentioned, namely causal consistency [Lam78, Vog09], read-your-writes consistency [Vog09] and timed consistency [TRAR99, LBC12]:

1) **causal consistency:** in this paper, it means when player A uses a client software or a browser to access a game server, the server will then transmit the latest game data in the form of data packets to the client side of player A. In this case, the subsequent local access by player A is able to return the updated value. Player B who has not contacted the game server will still retain the stale data.
2) **read-your-writes consistency:** in this paper, it describes that once state data of player A have been persisted in the database, the subsequent read request from player A will fetch the up-to-date data, yet others may only obtain a stale version of them.
3) **timed consistency:** in this paper, it specifically means that update operations are performed on a quorum of replicas instantaneously at the time t, and then the updated values will be propagated to all the other replicas within a time bounded by $t + \triangle$ [LBC12].

BASE and ACID are actually at opposite ends of the consistency-availability spectrum [Bre12]. Most NoSQL stores limit ACID support [GS11]. Some of them use a mix of both approaches. For example, Apache Cassandra introduces lightweight transactions in a single partition.

3.3.3 RDBMSs vs. NoSQL DBMSs

RDBs and NoSQL stores are built on different data models, namely the relational/SQL model and the NoSQL model.

Relational/SQL model: in the relational/SQL model, data are represented in a relation (table) of attributes (columns) and tuples (rows). The database schema is fixed. The attribute name and its data type must be predefined. Each tuple contains the same attributes. Even though not every tuple needs all these attributes, the database will still assign all of them to it and insert a NULL (in SQL model) into the appropriate field (as an attribute value). The integrity constraints (e.g., primary key, foreign key and value range constraint) describe valid tuples of a relation. In addition, the transactional integrity constraints (ACID properties)

describe valid changes to a database. This structure is suitable for join or complex query operations across relations (tables). Figure 3.2 illustrates a sample of tables in an RDB. Three tables are connected by foreign keys. All tables have fixed schema.

NoSQL model: NoSQL stores have simplified the relational/SQL model. Their data typically are represented as a collection of key-value pairs. And they provide a flexible/soft schema. Each key-value pair could contain divers types/numbers of value. Each tuple (row) support to increase or decrease the number of the key-value pairs as needed. They typically do not place constraints on values, so values can be comprised of arbitrary format. Each tuple is identified by a primary key or composite keys. Many integrity constraints have been canceled (e.g., foreign key constraint) or weakened (e.g., transactional integrity constraint). For this reason, data partition is easy to reach, and the system can scale out arbitrarily. The flexible data model makes it possible to use denormalization in place of join operation across entities, so the system performance has been significantly improved. We have mapped tables in Figure 3.2 to a NoSQL store showed in Figure 3.3. Data in the RDB have been denormalized in one table, which has a dynamic schema. If a character has more than one item, accordingly more key-value pairs/columns will be appended to the corresponding row.

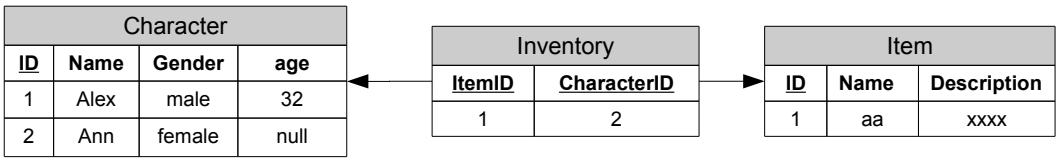


Figure 3.2: A Sample of Tables in an RDB

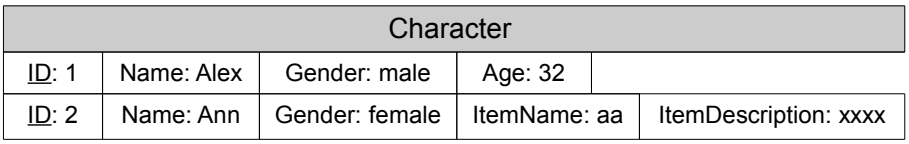


Figure 3.3: A Sample of a Table in a NoSQL Store

NoSQL stores are implemented in significantly different ways, but they still have some common characteristics. Based on the research result of Ben Scofield we have rated different categories of RDBMSs and NoSQL DBMSs in Table 3.1 [Sco10].

NoSQL DBMSs are often excellent in the aspects of partition tolerance, performance, availability, scalability and development costs. However, their drawbacks are also obvious. For example, they are limited to the functionality (e.g., ad-hoc query, data analysis and transaction management) due to the lack of support of a SQL-like query language

	RDBMSs	NoSQL DBMSs
Data Consistency	high	variable (low)
Functionality	high	low
Reliability	high	variable (moderate)
Complexity	moderate	low
Partition Tolerance	high	high
Flexibility	low	high
Runtime Performance	variable	high
Availability	moderate	variable
Scalability	variable	high
Cost	high	low

Table 3.1: A General Rating of Different Categories of Two Kinds of DBMS [Sco10]

and the limitation of their underlying structures. We can even state that all functions that NoSQL DBMSs support could be achieved by RDBMSs; they are less mature than RDBMSs because they do not have the decades of experience of application and development. Particularly, they tend to be open-source, with normally just one or two companies/communities handling the support angle; additionally, the simple key-value pair structure is failed to support values with schemes of arbitrary complexity.

In fact, NoSQL DBMSs are complementary to RDBMSs in some aspects. These two kinds of DBMSs have their own characteristics and application scenarios. Hence, they will not replace the other. In the rapidly developing Web 2.0 era, we should choose the right DBMS according to the business scenario, or even combine various DBMSs in order to get their advantages. That means, we use RDBMSs to concern in the functionality (e.g., ad-hoc query) of the system, and use NoSQL DBMS to persist data (e.g., fast backup and recovery of data). In this project, we have adopted this approach to manage data in MMORPGs.

3.3.4 Classification of NoSQL DBMSs

Based on their data models we can mainly classify NoSQL DBMSs into the following groups[5]:

Key-value store: the data structure of it is similar to the Hashtable, in which a key corresponds to one value. Each key appears at most once (unique) in the collection. Redis, Riak and Dynamo employ this data model.

Wide Column Store: it also uses key-value pairs to store data, in contrast one key corresponds to multiple columns (key-value pairs). Wide column stores often employ a structure like tables, rows and columns to store structured and semi-structured data. Unlike in the relational model, the number of columns is not

[5]http://nosql-database.org/ (accessed 03.12.2015)

fixed, and the column names and their data types can vary from row to row in one table. It has the ability to hold a large number (billions) of columns in one row. Timestamp is recorded in each column to determine the valid content. Google BigTable, Apache Cassandra and HBase use this data model.

Document Store: it generally uses a format as JSON to store data. Its content is in the document type. Hence, it has the opportunity to build an index on certain fields to achieve some of the features of an RDBMS. CouchDB and MongoDB are based on this data model.

Others: there are still many other types of NoSQL stores, such as graph databases, multimodel databases, object databases, XML databases and so on.

3.4 Apache Cassandra

In the *Cloudcraft* project, we have taken Cassandra as an example to persist data in MMORPGs. Thus in this section, we will give an introduction of Cassandra.

Cassandra is a distributed NoSQL DBMS, which was initially developed at Facebook to power their Inbox Search feature [LM10]. It was released as an open source project in 2008. Now the enterprise edition of Cassandra is supported by DataStax, Inc.[6]. Cassandra inherits the data model of Google BigTable [CDG+08] and the completely decentralized architecture of Amazon Dynamo [DHJ+07]. Due to its linear scalability and high availability Cassandra is widely in use at many Web 2.0 websites like Digg and Twitter, as well as some commercial websites like eBay, GitHub and Netflix.

Cassandra has following main features[7]:

Fault tolerant: data in Cassandra are automatically replicated to multiple nodes (could be across multiple data centers) for fault-tolerance. A failed node can be replaced by another node with no downtime.

Decentralized: Cassandra has a peer-to-peer structure. Every node in the cluster is identical. That means, there is no master/primary node. Accordingly, there are no single points of failure or network bottlenecks.

Durable: even when an entire data center goes down, there is still no data lose.

Scalability: read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications [KKR14].

Tunable consistency: consistency level of writes and reads is offered to be tunable. Users are allowed to specify the consistency level for each write/read independently from “writes never fail” to “block until all replicas to be available”, with the quorum level in the middle.

[6] http://www.datastax.com/ (accessed 15.02.2016)
[7] http://cassandra.apache.org/ (accessed 23.01.2016)

Query language: from the release of 0.8, Cassandra Query Language (CQL) has been added. The CQL syntax is similar to that of SQL, so developers, who are familiar with SQL, do not need to spend much time to learn it. The function of CQL is, however, not as strong as SQL. For example, for retrieving data, only the columns that are part of the primary key and/or have a secondary index defined on them could be used as query criteria [Casb].

Lightweight transaction: from the release of 2.0, Cassandra starts to support lightweight transactions, which are restricted to a single partition. This feature aims at supporting a linearizable consistency or the isolation in ACID terms.

In the following subsections, we will provide an overview of Cassandra [Hew10, LM10].

3.4.1 Data Model

Cassandra is a partitioned wide-column store. Its data model brought from Google BigTable. To understand the data model, we need to know the following terminologies:

Column: column is the basic unit of data in Cassandra. It consists of three components, namely name (key), value and timestamp (Table 3.2). Values of these components are supplied by the client, including the timestamp (represents when the column was last updated). Timestamp is used in Cassandra for conflict resolution. If there is a conflict with the column value between two replicas, the column with the highest timestamp will replace another one. Timestamp cannot be used in the client application. Therefore, we can consider the column as a name/value pair.

A column value has two properties/markers, *TTL* (time to live) and tombstone. *TTl* is an optional expiration period (set by the client application), after which the data will be marked with a *tombstone*. Data with a tombstone will be then automatically removed during the *compaction* and repair processes (we will discuss it later in detail).

Additionally, from Cassandra 1.2, collection types (e.g., set, list and map type) are supported. That means, we could store multiple elements in a single column value. Elements in the set type are sorted, and there is no duplicate values. List type keeps the insertion order, and allows duplicate values. A map type contains a name and a pair of typed values (user defined), and elements are sorted. Each element has an individual TTL. Furthermore, in Cassandra 2.1 and later, we can create a user-defined type to attach multiple data fields to a column or even an element in a collection type.

Row: row is a container of columns. Each row is uniquely identified by a row key that supplied by the client, and consists of an ordered collection of columns related to that key (Table 3.3). In RDBs it is only allowed to store column names as

Column		
name: byte[]	**value: byte[]**	**timestamp: Int64**
"user"	"Mila"	2015-10-10 02:22

Table 3.2: Structure of a Column

Row				
Row Key: byte[]	column 1	column 2	...	column N
	1——————>2 billion columns			

Table 3.3: Structure of a Row [McF13]

	Column Family					
sorted by row key	**row key 1**	column 1	column 2	column 3	column 4	column 5
	row key 2	column 3	column 6			
	row key 3	column 1				
	row key 4	column 2	column 6	column 7	column 8	
	row key 5	column 1	column 3	column 5	column 7	column 9
	row key 6	column 3	column 6	column 7		
	sorted by column name					

Table 3.4: Structure of a Column Family

strings, but in Cassandra both row keys and column names can be any kind of byte array, like strings, integers and UUIDs. Cassandra supports a maximum number of columns in a single row up to 2 billion [Casa], and is consequently called wide-column store.

Column family: rows are ordered by their keys in column families (Table 3.4). Each row does not have to share the same set of columns. Column families are predefined, but the columns are not. Users can add any new column to any column family at any time. Hence, column family has a flexible schema.

Keyspace: a keyspace contains several column families. It is the outermost container for data in Cassandra. When we create a keyspace, attributes like data replication strategy (replica placement strategy), number of replicas (*replication factor*) and column families must be defined.

Cluster: Cassandra is typically distributed over several machines that operate together. Cluster (sometime also called ring) is the outermost container of the system. Usually there is only one keyspace in a cluster. Data in Cassandra are distributed in nodes, each of which contains at most one replica for a row. Cassandra arranges nodes in the cluster in a ring format.

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family
Primary key	Row key
Column name	Column name
Column value	Column value

Table 3.5: Analogy of Relational Model and Cassandra Model [Pat12]

Table 3.5 helps us transform from the relational world to Cassandra world. However, we cannot use this analogy while designing Cassandra column families. Instead, we need to consider column family as a map of a map. The key of an outer map is row key, and similarly the key of an inner map is column name. Both maps are sorted by their keys [Pat12]. Because of that we can do efficient lookups and range scans on row keys and column keys. Furthermore, a key can itself hold a value, consequently, a valueless column is supported.

3.4.2 Architecture

From this subsection, we start to discuss the internal design of Cassandra.

3.4.2.1 Peer-to-Peer architecture

Nodes in many traditional database clusters (e.g., MySQL Cluster) and even in some advanced data stores (e.g., Google BigTable), are playing different roles. For instance, MySQL cluster divides its nodes into three groups, namely management nodes, data nodes and SQL nodes. Additionally, not all nodes in a cluster are equal in the data processing. For instance, by applying a master/slave model, only master nodes are responsible for updating data, and the updates will be then synchronized to slave nodes. This model is optimized for reading because clients can fetch data from each node. Furthermore, it is beneficial to data consistency in a distributed environment. However, there is a potentially single point of failure. When a master node is offline, the relevant update service will block until a slave node takes over it.

By contrast, Cassandra adopts a peer-to-peer (P2P) model, where all nodes are identical. This design makes Cassandra overall available and easy of scaling. Firstly, removing or taking offline of any node will not interrupt the read/write service. Secondly, in order to add a new server, we simply need to add it to the cluster without complicated configuration. The new added node will automatically learn the topology of the ring and get the data that it should be responsible for from other nodes. And then it starts to accept requests from clients.

A precondition for supporting this decentralized architecture is that nodes are aware of each other's state. Cassandra applies a gossip protocol for the intra-ring communication. Gossip service (Gossiper class) starts with the starting of Cassandra on a machine, and

runs every second on a timer to communicate other nodes. The gossiper class on each node holds a list of the state information of all nodes (alive and dead). Gossiper sends a message to a random node periodically in order to synchronize the state information and detect failure. State information in the gossiper class includes load-information, migration and node status, such as bootstrapping (the node is booting), normal (the node has been added into the ring and is ready for accepting reads), leaving (the node is leaving the ring) and left (the node dies or leaves, or its token has been changed manually).

3.4.2.2 Ring

Cassandra assigns data to nodes in the cluster by arranging them in a logical ring. Token (a hash value) is used in Cassandra for data distribution. Each node holds a unique token to determine its position in the ring (from small to large in the order of clockwise, see Figure 3.4) and identify the portion of data it hosts. Each node is assigned all data whose token is smaller than its token, but not smaller than that of the previous node in the ring (see Figure 3.5). The range of token values is determined by a partitioner. Cassandra uses *Murmur3Partitioner* as default for generating tokens, consequently, the range of token values is from -2^{63} to $2^{63} - 1$. Cassandra partitions data based on the partition key, which is computed to a token value by a token function. Cassandra uses the primary key (row key) as the partition key. When a row is identified by a compound key (multiple columns), the first column declared in the definition is treated as the partition key.

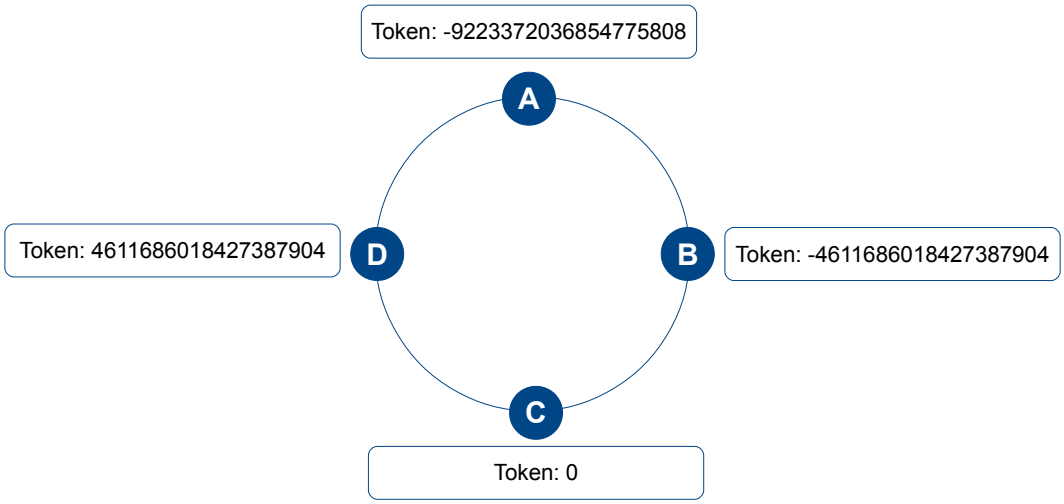


Figure 3.4: A Four-node Cassandra Ring using Murmur3Partitioner

Data replication is typically used in Cassandra to ensure reliability and fault tolerance. The number of copies for each row of data is specified while creating a keyspace by specifying the *replication factor* (an attribute of keyspace). A typical setting of that is *THREE*. That means, in the ring/cluster there are three nodes hosting copies of each row. There is no primary or master replica. This replication is transparent to clients.

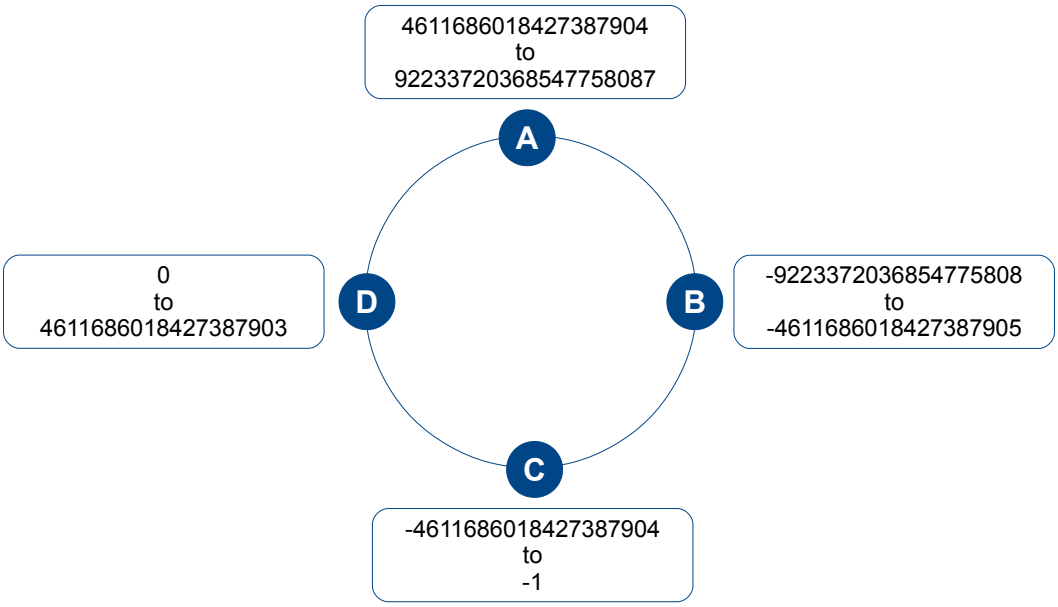


Figure 3.5: Token Range of each Node

Replica placement strategy is another attribute of the keyspace, which refers to how replicas will be placed in the ring. *SimpleStrategy* is used for a single data center, which places the first replica on a node according to its token value, and places additional replicas on the next nodes clockwise in the ring (see Figure 3.6). There is another strategy named *NetworkTopologyStrategy*, which is recommended for multiple data centers. Replicas will be placed on distinct racks across data centers.

Adding or moving a node in the ring will trigger the rearrangement of token values on relevant nodes (not all) automatically. The new added node starts to provide read services only after obtaining all required replicas.

3.4.2.3 Data Storage Mechanism

The storage mechanism of Cassandra borrows ideas from BigTable, which uses *Memtable* and *SSTable*. Before writing data, Cassandra firstly writes the operation in a log, which is called *CommitLog* (there are three kinds of commit log in the database, namely undo-log, redo-log and undo-redo-log. Cassandra uses timestamp to recognize the data version, hence *CommitLog* belongs to redo-log.). And then data are written to a column family structure called a *Memtable*, which is a cache of data rows. Data in a *Memtable* are sorted by keys. When a *Memtable* is full, it is flushed to disk as an *SSTable*. Once flushed, a *SSTable* file is immutable. That means no further writes can be done, but only reads. The *Memtable* will be then flushed to a new *SSTable*. Thus, we can consider that there is only sequential writes and no random writes in Cassandra, which is the primary reason that its write performance is so well.

SSTable cannot be modified, so that normally a column family corresponds to multiple *SSTables*. While performing a key lookup, it would increase the workload greatly if all

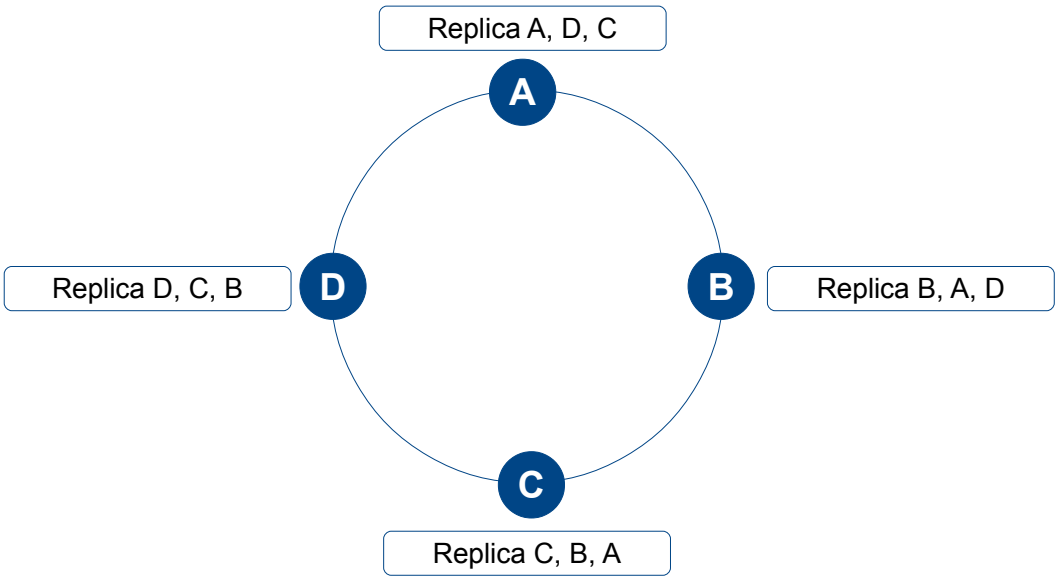


Figure 3.6: Data Distribution with SimpleStrategy (*replication factor* is 3)

SSTables are scanned. To avoid scanning the unnecessary *SSTables* Cassandra applies *Bloom filters*, which map all keys containing in *SSTables* to a bit array in memory. Only when the filter indicates that the required key exists in a *SSTable* file, the disk is accessed to get it.

To bound the number of *SSTable* files Cassandra performs *compaction* regularly. *Compaction* refers to merging multiple old *SSTables* containing the same column family into a new *SSTable*. The main tasks of *compaction* are:

Garbage Collection: Cassandra does not delete data directly, thereby consuming more and more disk space. *Compaction* moves the data with tombstone marker from disk.

Merger of *SSTables*: *compaction* merges multiple *SSTable* files (including index, data and filter) into one to improve the read performance.

Generation of *MerkleTree*: In the process of the merger, a new *MerkleTree* of the column family is generated. A *MerkleTree* is a hash tree to represent the data in a column family. It is used to compare with that on other nodes to reconcile data.

3.4.3 Guarantee of Eventual Consistency

The method of guaranteeing eventual consistency is to regularly check whether all replicas are consistent. If not, take some synchronization measures to repair it. Cassandra adopts three built-in repair utilities for that, namely *Anti-Entropy*, *Hinted Handoff* and *Read Repair*.

***Anti-Entropy*:** it is a replica synchronization mechanism. During a major *compaction*, the node exchanges *MerkleTree* with neighboring nodes to detect whether a conflict exists. If any difference is found, it launches a repair for that.

***Hinted Handoff*:** during the processing of a write operation, if the write consistency can be met, the coordinator (a node in the cluster handling for this write request) creates a hint in the local system for a replica node that is offline due to network partitioning, or some other reasons. A hint is like a small reminder, which contains the information of the write operation. However, the written data are not readable on the node holding the hint. When the replica node has recovered from the failure, the node holding the hint sends a message immediately to it in order to replay the write request. This mechanism makes Cassandra always available for writes, and reduces the time that a recovered node gets ready for providing read services.

***Read Repair*:** during/after responding a read operation, Cassandra may (10% probability as default) check data consistency on all replicas. If data are inconsistent, the repair work will be launched (a detailed introduction of *Read Repair* combining with consistency levels will be given later in Section 3.4.4.2). The probability for *Read Repair* is configured while creating a column family by changing *read_repair_chance*.

3.4.4 Data Processing

In this subsection, we will give a short summary of the procedure of writing, reading and deleting data, as well as consistency levels in Cassandra.

3.4.4.1 Writing Data

A write request from the client is firstly handled by an arbitrary node called coordinator in Cassandra cluster, which does not have to hold the row being written. The coordinator forwards then the request to *all* nodes holding the relevant replica. When the write is complete (it has been written in the *CommitLog* and *Memtable*) on a replica node, it sends back a success acknowledgment to the coordinator. Once the coordinator gets enough success acknowledgments depending on the write consistency level, the request is considered successful. The coordinator will then respond back to the client. Otherwise, the coordinator will inform the client that there are not enough available replicas to perform the write operation.

Figure 3.7 shows the procedure of performing a write in a single data center, which hosts an eight-node cluster with the *replication factor* three adopting *SimpleStrategy*. The write consistency level is *ONE*. When the coordinator receives the success acknowledgment from the first replica node (node C), the write operation is considered a success. If a replica is not available at this moment, thereby missing the write. Cassandra will make it eventual consistent using one of the synchronization measures mentioned in the previous subsection.

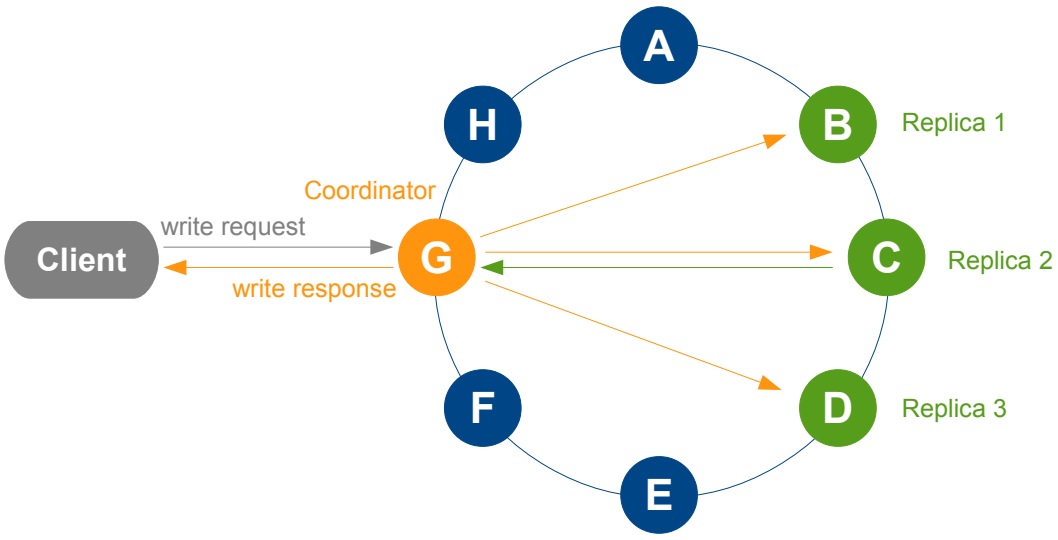


Figure 3.7: Procedure of Writing Data (Write Consistency Level is *ONE*)

In Cassandra, there is no update operation as in an RDBMS, where an old value of a column is replaced by a new one. Instead, while writing an existing partition, the new value is written with the modification time (timestamp). When there is a read against that column, the value with higher timestamp will be returned. Hence, "update" in Cassandra is already simplified as an insert, which does not need to get and modify the previous value. This feature makes the "update" efficiently.

3.4.4.2 Reading Data

Similar with writing data, a read request is also first sent to an arbitrary node in the cluster. And then reading data is divided into two steps.

Step one: the coordinator forwards a direct read request to the closest replica node, and a digest request to a number of replica nodes determined by the read consistency level. Accordingly, these nodes respond back with the row/a digest of the requested data. If multiple nodes are contacted, the coordinator compares the rows in memory, and sends the most recent data (based on the timestamp included in each column) back to the client. If the read consistency level cannot be fulfilled at the moment, the coordinator has to respond back the client that reading data is failed.

Step two: after that, the coordinator may also contact the remaining replica nodes in the background. The rows from all replicas will be compared to detect the inconsistent data. If the replicas are not consistent, the up-to-date data will be pushed to the out-of-date replicas. As we introduced above, this process is called *Read Repair*.

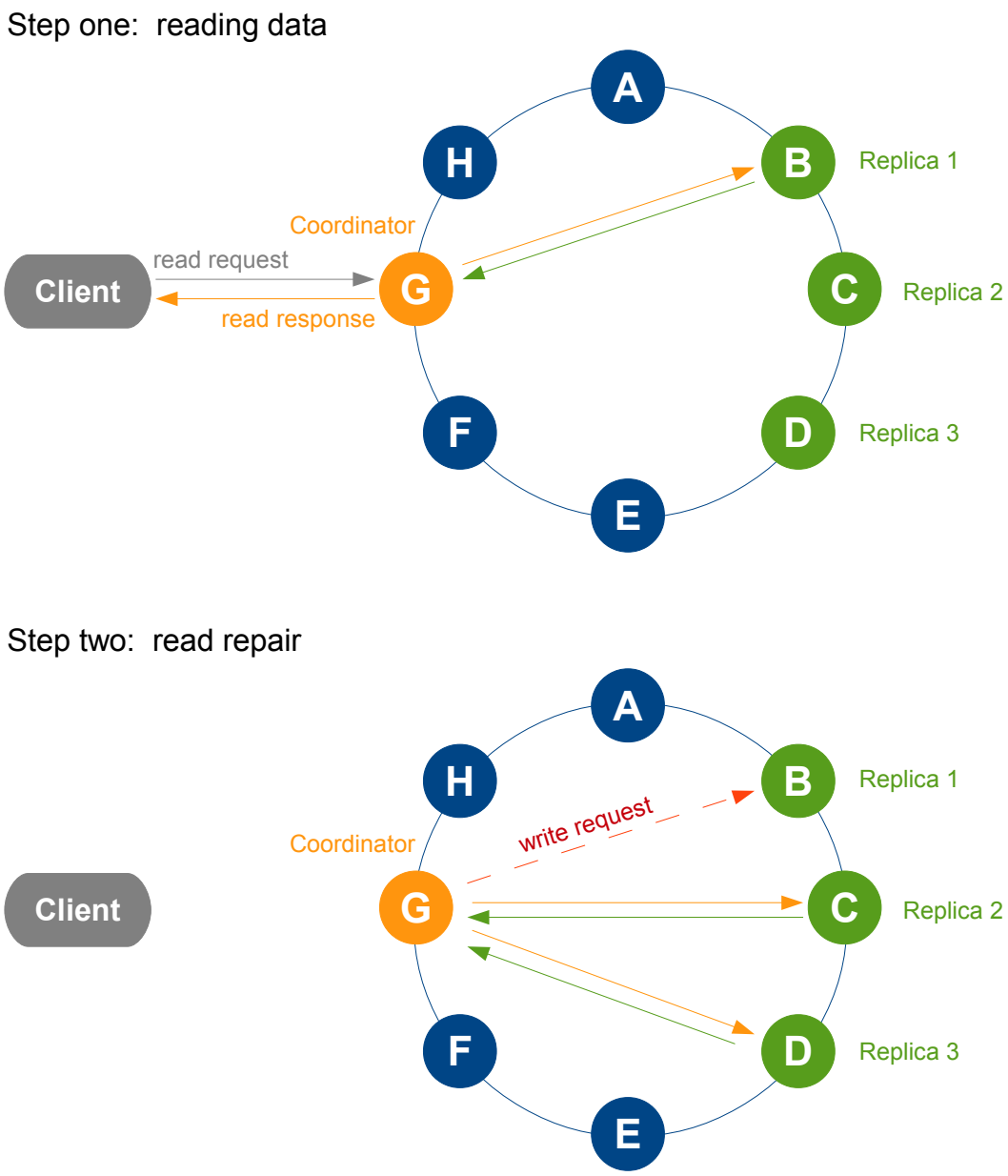


Figure 3.8: Procedure of Reading Data (read consistency level is *ONE*)

Figure 3.8 shows an example, where the read consistency level is specified to *ONE*, and the up-to-date rows are hold on replica node C and D. In the first step, only replica node B has been contacted because it is the closest replica to the coordinator (that means replica node B responds node G the fastest). The data fetched from node B is responded back to the client. In the second step, the remaining two replica nodes have been contacted. All rows from the replicas have been compared. The coordinator has found that the replica holding on the node B is out-of-date, so the coordinator issues a write to it.

Cassandra improves its read performance by holding a partition key cache and a row cache, which helps to avoid reading from disk. The partition key cache is enabled by default. It caches the partition index to help Cassandra know where a partition is located on disk so as to decrease seeking times and save CPU time as well as memory. The row cache is similar to a traditional cache like memcached in MySQL, which stores the entire contents of a partition that is frequently accessed. This feature, however, consumes large amounts of memory. Thus, on the official website it is suggested enabling this function unless it is in demand, and typically only one of these two kinds of caches should be enabled for a column family[8].

3.4.4.3 Consistency Levels

Cassandra provides a tunable consistency, which means client can specify how much consistency is required for each query. The level of consistency refers to how many replica nodes are involved in a query. The higher the level is, the more likely the fetched data are to be up-to-date, or the more replica nodes are synchronized, and consequently, the lower availability the query will be.

There are several write/read consistency levels that a client can specify. Particularly, the *QUORUM* level in Cassandra is calculated as follows:

$$(\textit{replication factor}/2) + 1 \tag{3.1}$$

For example, if the *replication factor* is specified to *THREE*, two replica nodes must respond the write/read request.

Write consistency levels: Table 3.6 shows the possible write consistency levels and their implications for a write request. It is noteworthy that the coordinator forwards a write request to all available replica nodes in all data centers, even if a low consistency level is specified.

Read consistency levels: read consistency levels are declared in Table 3.7. Read consistency level states the number of replicas must respond to a read request, so not all replica nodes would be contacted. In addition, *ANY* level is not supported here.

3.4.4.4 Deleting Data

Difference with RDBMS, Cassandra does not remove data from disk immediately when they are deleted. Instead, it adds a *tombstone* marker on that data, and removes them later in the background during the *compaction*. (So the delete in Cassandra is actually a write.) The reason is that Cassandra is designed to be distributed, durable and eventual consistent. If a node is down, it cannot receive as well as perform a delete request. When

[8] https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_configuring_caches_c.html (accessed 19.02.2016)

Level	Description
ONE/TWO/THREE/QUORUM/ALL	A write must be success on at least one/two/three/a quorum of/all replica nodes. If there are not enough available replica nodes, the write will fail.
LOCAL_ONE/LOCAL_QUORUM	A write must be success on at least one/a quorum of replica nodes in the local data center. It is used in multiple data center clusters with the replica placement strategy *NetworkTopologyStrategy*.
ANY	A write must be written on at least one node. If all replica nodes are down at write time, the write can still succeed after a *Hinted Handoff* is written. However, reading of this write is only available until the replica nodes for that partition have recovered.

Table 3.6: Write Consistency Levels [Dat16]

Level	Description
ONE/TWO/THREE/QUORUM/ALL	Coordinator returns the data after one/two/three/a quorum of/all replica nodes have responded.
LOCAL_ONE/LOCAL_QUORUM	Coordinator returns the data after one/a quorum of replica nodes in the local data center have responded.

Table 3.7: Read Consistency Levels [Dat16]

this node becomes available again, it will compare its data with other nodes. This node will mistakenly think all replicas nodes that received the delete request have missed a write request, thereby launching a repair. As a result, the deleted data would reappear.

Cassandra uses *compaction* to collect garbage regularly (The default setting for that is 10 days.). Not only the data with a tombstone marker, but also the out-of-date data generated by the "update" will be removed from disk.

3.4.5 CQL

CQL is the primary language for communicating with Cassandra. Cassandra develops rapidly in recent years. Accordingly, many new features are continually added in CQL, hence there are many differences between distinct CQL versions. This subsection focuses on the CQL v3 [Casb], which is the latest version of this language at the moment.

The Syntax of CQL is close to SQL. Numerous keywords (e.g., CREATE, INSERT, UPDATE, DROP, BATCH, COUNT, TABLE and PRIMARY KEY) from SQL are

reused here so that users can grasp this language quickly. As a result, it makes a sense that data are stored in tables containing rows of columns like in an RDB. However, as we know, Cassandra has a different data model with an RDB. We will give some samples of using CQL v3 in the following.

In Listing 3.1, we have created a keyspace called playerInfo. As introduced in the previous subsection, we can also specify a replica placement strategy (e.g., *SimpleStrategy*) and the number of replicas (e.g., *THREE*) while creating a keyspace.

```
CREATE KEYSPACE playerInfo
    WITH replication = {'class':'SimpleStrategy',
    'replication_factor': 3};
```

Listing 3.1: Creation of a Keyspace

In Listing 3.2, we have showed the statement of creating a column family. The syntax is similar to that in SQL to create a table contains a number of columns and a primary key for each row. Alternatively, we can even use the CREATE TABLE statement to create a column family. Furthermore, some operations like the chance of performing *Read Repair* can be specified here.

```
CREATE COLUMNFAMILY player (
    player_id int PRIMARY KEY,
    player_name text,
    player_race text,
    player_level int
) WITH read_repair_chance = 1.0;
```

Listing 3.2: Creation of a Column Family

Support of Dynamic Columns/Wide Rows:

CQL supports dynamic columns/wide rows by using a compound (composite) primary key in the column family. For example, in Listing 3.3 on the facing page, we have created a column family called userLogs, which uses user_id and took_at as a compound primary key. Besides user_id is the partition key and took_at is the clustering column in userLogs. Data on disk are ordered in ascending order of took_at values. In Listing 3.4 on the next page, we have inserted some rows in this column family. If we execute a command showed in Listing 3.5 on the facing page in cqlsh (a python-based command line client for executing CQL), we will get the results presented in Listing 3.6 on page 44, which shows how data are actually structured in Cassandra. Data with the same partition key are stored together in a same *SSTable*, and sorted by the value of the compound primary key.

```
CREATE TABLE userLogs (
    user_id uuid,
    took_at timestamp,
    actions text,
    PRIMARY KEY (user_id, took_at)
)
WITH CLUSTERING ORDER BY (took_at ASC);
```

Listing 3.3: Creation of a Column Family with a Compound Primary Key

```
INSERT INTO userLogs (user_id, took_at, actions)
        VALUES (1, '2016-02-16 20:19:00-0500', 'log in');
INSERT INTO userLogs (user_id, took_at, actions)
        VALUES (2, '2016-02-16 20:20:00-0500', 'upload a photo');
INSERT INTO userLogs (user_id, took_at, actions)
        VALUES (1, '2016-02-16 20:22:00-0500', 'change password');
INSERT INTO userLogs (user_id, took_at, actions)
        VALUES (1, '2016-02-16 20:23:00-0500', 'log out');
```

Listing 3.4: Insertion of Data

Support of Lightweight Transactions:

We can use IF clause in the INSERT and UPDATE statements to support lightweight transactions (see Listing 3.7 on the following page and Listing 3.8 on the next page). Similarly, we can also use IF NOT EXISTS/IF EXISTS clause in the CREATE/DROP and DELETE statements.

Drawbacks of CQL:

Although CQL is developing rapidly and getting more and more strong, it is still not yet SQL. Currently there are still some restrictions comparing with SQL. For instance, when we query data, only the columns that are part of the primary key and/or have a secondary index defined on them are allowed in the WHERE clause; the IN relation is only allowed on the last column of the partition key [Casb].

3.4.6 Client Tools

Users can use the CQL shell, cqlsh, and DataStax DevCenter (a graphical tool) to interact with Cassandra. Furthermore, developers can also use a number of drivers developed by DataStax for different programming languages (e.g., C/C++ Driver, C#

```
SELECT * FROM userLogs;
```

Listing 3.5: Selection of Data

```
[cqlsh]

 user_id | took_at                       | actions
---------+-------------------------------+-----------------------
       1 | 2016-02-16 20:19:00-0500 | log in
       1 | 2016-02-16 20:22:00-0500 | change password
       1 | 2016-02-16 20:23:00-0500 | log out
       2 | 2016-02-16 20:23:00-0500 | upload a photo
```

Listing 3.6: Results of the Query

```
INSERT INTO userLogs (user_id, took_at, actions)
        VALUES (1, '2016-02-16 20:20:00-0500', 'log in');
        IF NOT EXISTS;
```

Listing 3.7: Insertion of Data with Lightweight Transaction

Driver, Java Driver, PHP Driver, and Python Driver) in production applications to pass CQL statements from the client to Cassandra cluster. Since Cassandra and our client applications are written in Java, in the following, we will give a brief introduction for Java Driver[9].

Java Driver has a fully asynchronous architecture, which is based on layers. There are three modules in the driver, namely driver-core, driver-mapping (an object mapper) and driver-examples. Driver-core (also as the core layer) handles things like connection pool, discovering new nodes and automatically reconnecting, which are related to the connections to a Cassandra cluster. Moreover, the higher level layer can be built on top of API provided by it.

Java Driver is built on Netty, which helps to provide non-blocking I/O with Cassandra. Furthermore, it has the following main features:

Synchronous and asynchronous API: except for performing a simple synchronous query, Java Drive provides an asynchronous API to allow the client to get query

[9]http://docs.datastax.com/en/developer/java-driver/2.0/common/drivers/introduction/introArchOverview_c.html (accessed 19.02.2016)

```
UPDATE userLogs
    SET actions = 'post a status'
    WHERE took_at = '2016-02-16 20:20:00-0500'
    IF actions = 'log in';
```

Listing 3.8: Update of Data with Lightweight Transaction

results in a non-blocking way. This concept makes it possible to maintain a relatively low number of connections per nodes to achieve good performance.

Connection pooling: by using a connection pool, a request is allowed to use an opened connection to access Cassandra cluster, which eliminates the process of creating and destroying a connection. It is noteworthy that for each session, there is only one connection pool per connected host. The number of connections per pool can be changed at run-time, which depends on the current load and the configuration from users.

Automatic nodes discovery: the driver automatically obtains information like status of all nodes in the cluster.

Transparent failure: the driver will automatically and transparently connects another node if a Cassandra node is down and the client has not specified that node as a coordinator. Java Driver will automatically perform a reconnection to the unavailable node in the background.

Cassandra trace handling: the client can trace a query by using a convenient API of Java Driver. Information of a query like the IP address of the coordinator as well as nodes performing the query and the number of involved *SSTable* is included in the tracing result.

Configurable load balancing: users can configure a *LoadBalancingPolicy* for each new query, which determines the node will be picked as a coordinator, and hosts will be communicated with by the driver. There are several implementations provided by Java Driver, such as *RoundRobinPolicy*, *DCAwareRoundRobinPolicy*, *TokenAwarePolicy* and *LatencyAwarePolicy*. From the release 2.0.2 of the driver, *TokenAwarePolicy* is used by default. By using this policy, the driver first communicates with a replica node (as coordinator) instead of a random node in the cluster for the purpose of reducing internal network communications.

3.5 Summary

In this chapter, we have introduced foundations of the rapid developed Cloud storage systems, compared NoSQL DBMSs and RDBMSs in detail, and highlighted Cassandra. In the next chapter, we will propose a Cloud-based architecture for MMORPGs.

4. Cloud Data Management for MMORPGs

> Some ideas in this chapter are originally published in the DASP article "Cloud Data Management for Online Games: Potentials and Open Issues" [DSWM13] and the CLOSER paper "Towards Cloud Data Management for MMORPGs" [DS13].

In this chapter, we will discuss the feasibility of introducing Cloud-based technologies (e.g., Cloud database and Cloud storage) to manage data in MMORPGs.

4.1 Feasibility Analysis

Currently, data in MMORPGs are managed by distributed RDBMSs and file systems. From the technical perspective, all functions of the above storage systems can be obtained from Cloud services. Furthermore, through using Cloud-based technologies, some data management requirements of online games (e.g., availability, scalability and high performance) that used to be difficult to meet could also be satisfied. However, some features are mutually exclusive, such as data consistency and availability (according to the CAP theorem). For this reason, firstly before we apply a Cloud-based technology in online games, we need to know whether the new features that this technology brings are necessary or not; and secondly, we must find out the right balance for the opposite features [KHA09, KK10]. It is noteworthy that there are various data sets in a game system, which are managed in different ways and accordingly have different management requirements (see Table 2.1). In the following, we will analyze them separately. Since one of the most important motivations that we apply Cloud-based technologies is scalability, and the biggest conflict they bring is between data consistency and availability, we will use them as criteria to determine whether to adopt a Cloud-based technology. The requirements and our recommendations have been summarized in Table 4.1.

	Stored in	Consistency	Availability	Scalability	Recommendation
Account data	Database	Strong	Low	Low	RDBMS/NoSQL DBMS
Game data	Client	Causal	High	Low	Local file system & database
	Server	Eventual	High	Low	Distributed file system & database
State data	Server	Strong	High	Low	In-memory RDBMS
	Database	Read-your-writes	High	High	NoSQL DBMS
Log data	Database	Timed	High	High	NoSQL DBMS

Table 4.1: Data Management Requirements and Recommendations for Data Storage

4.1.1 Requirements of Account Data

Account data, such as players' identity information and some other sensitive data (e.g., password and recharge records), are stored in the account database in online games. The scalability and data consistency requirements are listed in the following:

Consistency vs. Availability: the inconsistency of account data might bring troubles to a player as well as the game provider, or even lead to an economic or legal dispute. Imagine the following two scenarios: a player has changed the password successfully. However, when this player logs in to the game again, the new password is still not valid; a player has transferred to the game account, but the account balance is not properly presented in the game system. Both cases would influence the player's experience, and might result in the customer or the economic loss of a game company. Hence, we need to access account data under strong consistency guarantees, and manage them with transactions. Availability is less important here.

Scalability: an account database manages data from millions of players, but with a small size. Hence, system scalability is not required. Processing of a large number of concurrent requests might become a challenge for the database, which could be addressed by database sharding.

In general, an RDBMS can already fulfill all management requirements of account data. On the other side, if there is no transaction between rows, a NoSQL DBMS (CP type, see Section 3.3.1) supporting lightweight transactions can handle them better because the user account and password (as well as further fields) could be regarded as key-value pairs to process.

4.1.2 Requirements of Game Data

Game data refer to all data/files that are generated/created and can only be changed by game developers, such as game world geometry and appearance, object and NPC's metadata (name, race, appearance, etc.), game animations and sounds, and so on. They are distributed at both client and server side, and managed by databases (e.g., object's metadata) and file systems (e.g., game animations). In the following, we will discuss the consistency, availability and scalability requirements of game data.

Consistency vs. Availability: players are not as sensitive to game data as to account data. For example, the change of an NPC's appearance or name, the duration of a bird animation, and the game interface may not catch the attention of a player and have no influence on players' operations. On the other hand, some changes of the game data must be propagated to all online players synchronously, for instance, the change of the game world's appearance, game rules as well as scripts, and the occurrence frequency of an object during the game. The inconsistency of these data will lead to errors on the game display and logic, unfair competition among players, or even a server failure. For this reason, we also need to treat data consistency of game data seriously.

Game data are updated or loaded from a game server to the client side when a player logs in to or starts a game. Therefore, from a player's point of view, a causal consistency (see Section 3.3.2) is required. From a game server's point of view, as long as players in the same game world hold the same version of game data, the game is fair (players across game worlds are not able to communicate with each other, consequently, data inconsistency among game worlds will not be detected or affect the gameplay). It is noted that a game world is typically hosted by one game server. Hence, eventual consistency among game servers is acceptable. That means, both conventional and Cloud DBMSs/file systems can manage them.

Scalability: game data size changes with upgrading the game or launching a new game edition. The growth of them typically will not pose a challenge to the hard disk space. Hence, there is no scalability requirement.

Generally, the management of game data do not pose any challenge to the file system/-database. Both conventional and Cloud storage system can handle them.

4.1.3 Requirements of State Data

In MMORPGs, state data (e.g., Player Character's metadata and position information) are cached in an in-memory database at a game server to provide real-time executions, and duplicated (backed up) in a disk-resident database to ensure the data persistence. The management requirements for these two databases in an online game are distinct. We analyze them as follows.

Consistency vs. Availability: state data are modified by players frequently during gameplay. The modification must be perceived by all relevant players synchronously, so that players and NPCs can respond correctly and timely. An example for the necessity of data synchronization is that players cannot tolerate that a dead monster continues to attack their characters. Note that players only access the in-memory database during gameplay. Hence, this database must ensure strong consistency, and support the transaction management as well as

complex queries. Moreover, the request for data processing must be responded in real-time, so it has high requirement for availability.

Another point about managing state data is that modified values must be backed up to a disk-resident database asynchronously. Similarly, game developers also need to take care of data consistency and durability in a disk-resident database. For instance, it is intolerable for a player to find that her/his last game record is lost when she/he starts the game again. In contrast to that in the in-memory database, we do not recommend ensuring strong consistency to state data. The reason is as follows: according to the CAP theorem, we have to sacrifice either data consistency or availability in the case of network partitioning or high network latency. Obviously, it is unacceptable that all backup operations are blocked until the system recovers, which may lead to data loss. Consequently, the level of data consistency has to be decreased. We propose to ensure a read-your-writes consistency (see Section 3.3.2) for state data on disk.

Scalability: we have already discussed the scalability requirement of state data in Section 2.3. MMORPGs are having trouble dealing with the surge in the size of state data. However, it is noted that state data in the in-memory database do not have a large scale, which is limited by the total number of concurrent players that a game server supports.

It is reasonable to manage state data in the in-memory database and the disk-resident database by using different technologies. An in-memory RDB is more suitable for caching state data because it can perform complex queries. We propose to use a NoSQL store (AP type, see Section 3.3.1) in our project to persist/back up state data on disk because it can solve the scalability issue and guarantee high availability. The challenge is that there is no such a NoSQL DBMS exactly designed for MMORPGs. We must find a relatively appropriate one, and then improve it to fulfill all requirements (e.g., supporting read-your-writes consistency) of managing state data.

4.1.4 Requirements of Log Data

In our project, log data refer to a player's behaviors, chat histories and mails. These data are structured persisted in the database. In the following, we discuss their requirements for data consistency, availability and scalability.

Consistency vs. Availability: log data are mainly used by game developers for fixing bugs of the game program or by data analysts for data mining purpose. These data are firstly sorted and cached on the server side during the game, and then bulk stored into a disk-resident database, thereby reducing the conflict rate as well as the I/O workload, and increasing the total simultaneous throughput [BBC+11].

It is noteworthy that propagation of log data among replicas will significantly increase the network traffic and even block the network. Moreover, log data are

generally organized and analyzed after a long time. Data analysts are only concerned about the continuous sequence of log data, rather than the timeliness of them. Hence, data inconsistency is acceptable in a period of time. For these reasons, a deadline-based consistency model, such as timed consistency (see Section 3.3.2), is more suitable for log data.

Scalability: log data are derived from millions of players, and are appended continually. Accordingly, the database managing them must be scalable.

The general format of log contents is a specific time followed by an operation information, which is a natural key-value pair. Moreover, they have high requirements for scalability and availability, but low requirements for data consistency. Therefore, management of log data using a NoSQL DBMS (AP type, see Section 3.3.1) is more sensible than using an RDBMS. We can use some tools like Apache Spark[1] for data analysis then.

4.2 A Cloud-based Architecture for MMORPGs

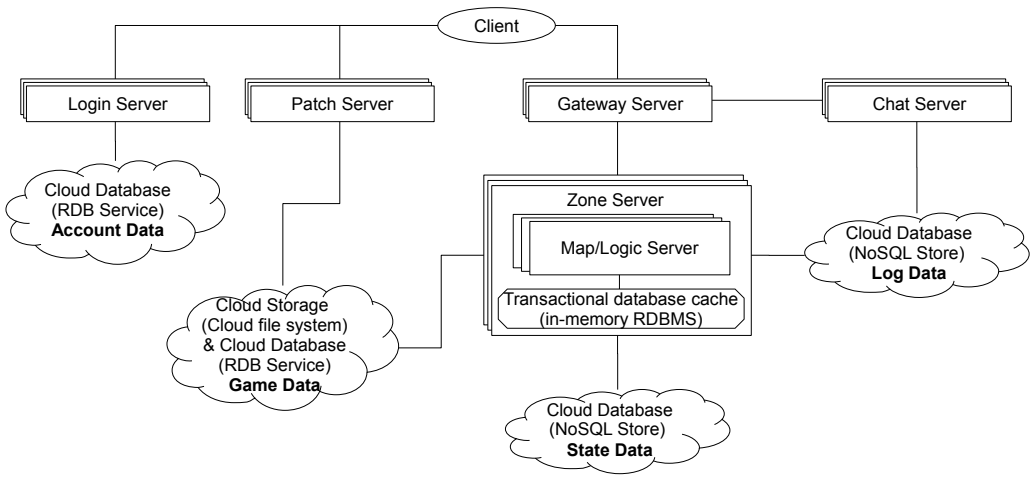


Figure 4.1: A Cloud-based architecture for MMORPGs

Based on the analysis above, we have proposed a set of cooperating and composable Cloud services to improve the existing game architecture [DSWM13].

We suggested managing game data with Cloud storage (Cloud file system) as well as Cloud database (RDB service), and manage other data sets with Cloud databases. Account data must be processed with ACID transactions, so they are managed by an

[1] http://spark.apache.org/ (accessed 12.01.2016)

RDBMS service. In contrast, log data and state data are recommended persisting in a NoSQL store.

Storing data in a public Cloud can make a game company focus on the game development rather than maintaining the data storage system. Data are stored in the Cloud provider's data center. The provider is responsible for managing the data, which helps a game company to reduce the time of developing new game editions and costs for buying new drivers. However, the drawback is that the data could be stored in an insecure environment, which cannot be controlled by the game company. Furthermore, the latency of data transmission highly depends on the network traffic between game servers and the Cloud provider's data center. For this reason, this hosting solution is suitable for a small game company or a company in its infancy. However, it is better to host the account data by the game company itself.

For a game company already having expensive data centers, building a private Cloud environment could be the best option. Data are transmitted across the company's intranet, hosted at internal data centers, and protected behind a firewall. The drawbacks are the costs for maintenance/upgrading infrastructures and buying software licenses. The game company can use Hadoop Distributed File System (HDFS) to store game data. However, choosing a suitable NoSQL DBMS for state data and log data is a challenge. We will discuss it in the next section.

4.3 Criteria of Choosing a NoSQL DBMS

In contrast to RDBMSs, there are many types of NoSQL DBMSs, each of which is designed for a specific application scenario and differs from each other. Although NoSQL DBMSs are easy to use, choosing a suitable one from a wide range of products for a user is troublesome. We often need to consider the following factors for the determination:

Data structure characteristic: includes whether data are structured or semi-structured, whether a field can be changed, whether there is a large text field, whether data can be modified and so on.

State data and log data in MMORPGs are represented in multiple tables. Consider that we usually comply with the denormalization principle to design the schema of a NoSQL store, the new data structure must be complex; a field could be changed/added for the purpose of improving a game; a large text field is typically not contained in both data sets; state data are frequently modified, but log data are only appended.

A NoSQL store implementing a simple key-value pair model (key-value store) cannot handle such a complex and dynamic data structure. For this reason, a NoSQL DBMS implementing a wide column model (one key corresponds to multiple columns) or a document model is more suitable.

Query characteristic: includes the condition in queries and the range of hotspot queries. For example, the retrieval of user information is a random query, and the retrieval of news is in accordance with time (the latest news is queried normally more frequently).

In our case, there are two kinds of queries, namely the random query (retrieving the state of a player) based on the primary key (user ID) and the range query (retrieving all operations of a player in a certain period of time) based on composite keys (user ID and time).

A document store (implementing a document model) is designed to load the whole document, when data stored in one document are retrieved or modified. This feature ensures high-level consistency. However, in our case, we only need to read/write parts of the whole row. For this reason, a NoSQL store implementing a wide column model is a better choice because only the columns we need will be loaded.

Write/read characteristic: includes the proportion of insert/update, whether a data field is updated/read frequently, whether an atomic update is required.

State and log databases are mainly used for the backup/persistence purpose. Hence, there are far more insert/update operations than read operations; Atomicity is important for a write operation.

Therefore, a write-intensive type of NoSQL DBMS is more suitable for our application scenario.

In general, we need to choose a wide column-based NoSQL store, which is good at writing. There are many NoSQL DBMSs meeting these features. In our project, we decide to use Cassandra as an example because it is more mature and under an active development. In the next section, we will explain how Cassandra match the above criteria.

4.4 Possibility of Using Cassandra in MMORPGs

In Section 4.3 we have analyzed the criteria of choosing a right NoSQL store for MMORPGs. In this section we will explain how Cassandra complies these criteria in detail.

Data structure: Cassandra is a partitioned row store. Data are structured in the column family, where each row is identified by a primary key or composite keys. This structure is similar with that in an RDBMS. Therefore, it facilitates data mapping between two kinds of database systems.

Cassandra offers a collection column, which is declared using the collection type (e.g., list, set and map type), followed by another type like int or text. This feature helps for denormalizating state data from numerous relations in an RDB.

Cassandra supports a flexible schema. The set of columns in a row is dynamic. Users can add or remove a field freely at run-time, which is convenient for program debugging.

Traditional RDBMSs limit the number of columns in each table. For instance, in each table Oracle 11g supports maximum 1000 columns [Oraa] and SQL Server 2014 supports maximum 1024 columns [Mic]. When we create a wide table with hundreds of columns, RDBMSs often fail to manage it efficiently. If we do a column-wise decomposition, the high number of join operations and locking would greatly affect on the system performance. In contrast, Cassandra, which implements a wide-column data model, supports a maximum number of cells (rows * columns) in a single partition up to two billion [Casa]. This feature makes it possible to store all log information of one player in a single row so as to fetch it efficiently.

Query: although the functionality of CQL is less powerful than SQL, it still supports random/range queries on (compound) primary key.

Write/read operation: as introduced in Section 3.4 Cassandra is designed and optimized to perform writes efficiently: writes can be initiated by any node in the cluster; there is no single points of failure for insert/update. Even if all replica nodes are down, a write operation could still not be blocked (if write consistency level is *ANY*); data are first written in memory, and then are persisted on disk; update does not check the prior existence of the row. For these reason, Cassandra differing from other NoSQL stores is write-intensive.

Cassandra cannot only comply with these criteria, but also meet the requirements like system scalability, availability and data consistency. Therefore, it becomes a good choice to manage state and log data in MMORPGs.

4.5 Related Work

Commercial MMOG systems typically apply a client/server architecture for maintaining data security and synchronization [SKS+08, WKG+07]. A centralized server could, however, become a bottleneck of the system, thereby effecting on the system scalability. Hence, the design of an alternative scalable architecture for the MMOG system, has received more and more attentions. However, most of the research aim at making the system to support more concurrent players [SKS+08, KVK+09, YV05, GDG08, YK13, MGG06]. In other words, they focus on the scalability of the game server. Our research focuses on MMORPGs, which offers a persistent game world. That means, a large volumes of data need to be stored in the backend database, which becomes a challenge. The new Cloud-based architecture proposed in our CloudCraft project is aimed at improving the scalability and efficiency of the game database, rather than improving the scalability of the game server. Certainly, the improvement of the efficiency of backing up game state allows game servers to deal with more concurrent players.

In [Muh11], the author proposed to use another NoSQL store Riak[2], whose database model is similar to that of Cassandra, to substitute the RDBMS in MMOGs. However, after analyzing the use of databases in MMORPGs and features of NoSQL stores, we come to the conclusion that NoSQL DBMSs cannot substitute RDBMSs in the game scenario because of their limitations on the support of complex queries and the real-time processing, etc. [ZKD08]. We propose to make them coordinate in a new architecture, and deal with different data processing requirements.

In Section 4.1, we have analyzed game consistency requirements of each data set from the storage system's perspective. Although some studies have also focused on the classification of game consistency, they generally discussed it from players' or servers' point of view [ZK11, LLL04, PGH06, FDI+10]. In other words, they have actually analyzed data synchronization among players. Another existing research work did not discuss diverse data sets accordingly [DAA10, SSR08], or just handled this issue based on a rough classification of game data [ZKD08].

4.6 Summary

In this chapter, we have proposed a Cloud-based architecture for MMORPGs, and discussed the possibility of use Cassandra to substitute an RDBMS for managing state and log data. Cassandra is, however, not designed for online games, so it has some drawbacks while using in this application scenario. We will point out these shortcomings, and propose some solutions in the next chapter.

[2]Riak: http://basho.com/riak/ (accessed 05.08.2015)

5. Using Cassandra in MMORPGs

> Some ideas in this chapter are originally published in the DASP article "Cloud Data Management for Online Games: Potentials and Open Issues" [DSWM13], the GvD paper "Consistency Models for Cloud-based Online Games: the Storage System's Perspective" [Dia13], an article "CloudCraft: Cloud-based Data Management for MMORPGs" [DWSS14], the DB&IS paper "Cloud-based Persistence Services for MMORPGs" [DWS14], and the IDEAS paper "Achieving Consistent Storage for Scalable MMORPG Environments" [DZSM15].

In the last chapter, we have proposed to take Cassandra as an example for data persistence in the Cloud-based game architecture. However, there are some issues to address when we use it. In this chapter, we will analyze shortcomings of Cassandra using in the game scenario, and offer a viable solution for each of them. Issues like the guarantee of read-your-writes consistency efficiently, optimization of read performance, mapping database schema between an RDBMS and Cassandra will be discussed.

5.1 Guarantee of Read-your-writes Consistency

Cassandra is typically used in an eventually consistent environment. However, in our application scenario, we must guarantee a high-level consistency. Cassandra can ensure that, but in an inefficient way.

5.1.1 Issues Caused by Guaranteeing High-level Consistency

In a Cassandra cluster each node is identical, and there is no master/primary replica for a given data object. On the one hand, the failure of one node will not affect the system availability. On the other hand, the guarantee of high-level consistency is expensive. A query could be executed by any replica node in the cluster, thus to guarantee high-level

consistency (e.g., strong consistency or read-your-writes consistency for state data), the consistency level of a write operation and its subsequent read operation must meet the following prerequisite:

$$W + R > N, with\ W, R \in \{1, 2, \ldots, N\} \tag{5.1}$$

In this formula, N refers to the total number of replicas (*replication factor*) for a row, while W and R represent the consistency level of write and read, respectively. This formula states that only if the total number of replicas responded write and its subsequent read exceeds the *replication factor*, Cassandra could ensure data consistency. This is because only in this case at least one replica responding to the query contains the up-to-date data. However, this mechanism causes following issues.

Issues:

1) More than half of the replicas have to participate in the process of updating and getting data, which increases the response time for both write and read operations. Particularly in multiple data centers, where replicas are distributed in several locations, the influence is significant.

2) In the case of multiple nodes failure, only one replica available (*replication factor* is larger than one), write or/and read will fail because the prerequisite is not met, thereby undermining system availability. In fact, if only a fixed replica node performs write and its subsequent read (just like a master replica), the result must be up-to-date. Other replicas will be eventually consistent through the built-in repair utilities (see Section 3.4.3).

As a result, guarantee of the required read-your-writes consistency for state data in Cloud-based MMORPGs is problematic.

5.1.1.1 Efficiency of Data Propagation in Cassandra Cluster

Could we ensure data consistency by just specifying write and read consistency level to *ONE*? The answer is no! The reason is that data are asynchronously propagated to all replicas in Cassandra. However, the time gap between the write and a following read is a vital parameter, which affects the accuracy [BVF+12]. We have carried out some experiments on Cassandra to evaluate the efficiency of data propagation. The experimental setup here is the same with that described in Section 6.4.

Detection of Inconsistent Data

In the experiment we quantified the effect of eventual consistency, when the consistency level of both write and read is specified to *ONE* (*replication factor* is three). The result is shown in Table 5.1.

During test I, all five nodes in the cluster are available. Clients send a total of 10000 write requests to Cassandra. As soon as a write request is successful, a read request

Operation	Number of nodes	Write requests	Read requests	Detection method	Inconsistent data
WORO	5	10000	10000	Eager	10.43%
WORO	5	10000	10000	Lazy	0%
WORO	$3 \rightarrow 5$	10000	10000	Lazy	4.09%
WORO	$3 \rightarrow 5$	200000	10000	Lazy	22.16%

Table 5.1: Detection of Inconsistent Data

against this row is sent (so called eager detection). In theory, 66.7% of the results could be stale. In our practical experiment, however, only an average of 10.43% of them is stale. If the read requests are sent after all (10000) write requests are successful (so called lazy detection), no stale data have been detected (see test II).

The reason is that, no matter which consistency level is specified by the client, Cassandra actually forwards the write request to all available replicas at the same time. In other words, if all replicas are available, the modification will be synchronized to all of them. Since the write performance of Cassandra is high, in our experiment environment (a single data center) stale data are rarely detected.

In test III we have simulated a bad case: two nodes are failed when updating data, but all five nodes are alive when reading data (if more than two nodes fail, for some data objects all three replicas will be unavailable. Thus, write operations against these data could not be performed.). In this case, reading from the two temporarily unavailable nodes should fetch stale data. However, the result (see test III) shows that only an average of 4.09% of data is stale.

We increase the total number of write requests to 200000 so that there could be more inconsistent data (about 300 MB) in the cluster, and Cassandra needs more time to replay writes. The experimental result shows that although the number of stale data is enlarged (about 22.16%), it is not directly proportional with the increase of the number of write requests (see test IV). Furthermore, the inconsistency window in this situation is about 942296 ms (about 15 minutes 42 seconds), which is short.

5.1.1.2 Inspiration Obtained from the Experiment Results

The Cassandra cluster in our experiment is deployed in an Intranet, which reduces network latency. However, from the experiment result, we can still conclude that Cassandra actually offers a higher data consistency level than it promises (just like some other Cloud storage systems, such as Amazon SimpleDB, Amazon S3 and Azure [WFZ+11]). Although there could be some inconsistent data in the case of a node failure, they only exist for a short period of time after the node is restarted. In MMORPGs a player typically restarts the game after a long time. During this time gap, modifications of state data have a sufficient time to be propagated to all available replicas. Therefore, in most cases the up-to-date data could actually be fetched by specifying the consistency level of writes and reads to *ONE*. Unfortunately, a client could not specify the consistency level like that only because there might be an unpredictable node failure during the update, and consequently stale data would be returned later.

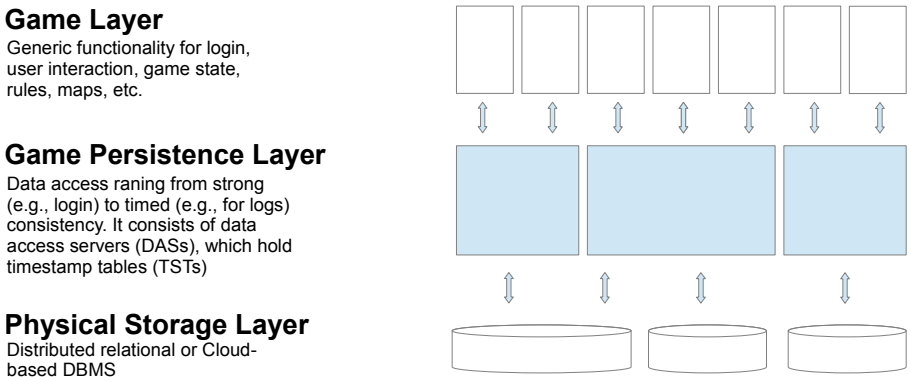


Figure 5.1: *CloudCraft* Architecture: Reusable Services for MMORPGs

Proposal:

To address this issue, the system should be able to detect stale data automatically. Only when stale data are returned, the system increases the consistency level of read and executes it again. To achieve this goal, the existing Cloud storage system needs to be extended [APV07, GBS11, WPC09, Dia13].

5.1.2 A Timestamp-based Solution

In our work, we have proposed a timestamp-based model (TSModel) to solve the issue discussed above [Dia13, DZSM15]. The basic idea is that a timestamp is applied as a version ID for each checkpoint of state data, which will be then used as a query criterion to fetch the current data [APV07]. In this subsection we will explain our solution in detail.

5.1.2.1 Integration with MMORPG Application Scenario

In order to guarantee game-specific consistency, a game persistence layer is applied between the game layer and the physical storage layer in the new Cloud-based game architecture (see Figure 5.1). This layer consists of data access servers (DASs) holding timestamp tables (TSTs). A DAS is responsible for creating consistent checkpoints from the in-memory database, flushing them to Cassandra, fetching data from Cassandra regarding game-specific consistency, and playing the role of a counter (generation of monotonically increasing timestamps). The structure of a TST is very simple, containing only four attributes, namely an avatar's or game object's ID (Id), the last checkpointing time (TS), the host's Internet protocol (IP) address of the last checkpoint in the Cassandra cluster, and a player's log status for avatar data.

A time asynchronisation among DASs less than the frequency of checkpointing is acceptable. In a typical client-server-based MMORPG, unless a player has changed the zone server or a DAS has failed, checkpointing of game state data is handled by a fixed DAS. For this reason, an accurate global time synchronization is not necessary. The system clock on each DAS could be synchronized by applying the network time protocol (NTP).

Data: snapshot of state data
Result: a checkpoint in Cassandra and a record in a timestamp table (TST)
begin
  $Id \longleftarrow$ *avatar's/game object's UUID*
  $TS \longleftarrow$ *system current time*
  $data \longleftarrow$ *snapshot of the avatar/game object's state data*

  //set write consistency level to ONE
  $CL \longleftarrow$ *ONE*

  //back up state data into Cassandra (checkpointing)
  if $Cassandra.put(Id, TS, data)$ *with* CL
  then
    //if succeeds (true)
    //save the checkpointing time as a version ID in TST
    $TST.put(Id, TS)$
  end
end

Algorithm 1: The Process of Checkpointing

5.1.2.2 Checkpointing and Data Recovery with the TSModel

To describe the timestamp-based detection model, we need to first outline the process of checkpointing game state data.

The Process of Checkpointing

The DAS creates a consistent snapshot of game state from the in-memory database periodically. The system's current time of the DAS will be used as a unique monotonically increasing version ID (also called TS) for each checkpoint. The DAS executes a bulk write to Cassandra with consistency level (CL) *ONE*. Cassandra divides the message into several write requests based on Id. The current state of an object and the TS are persisted together in one row. When the DAS receives a success acknowledgment, it will use the same TS to update the TST accordingly (see Algorithm 1).

When a player has quit the game and the state data of her/his avatar have been backed up to Cassandra, the log status will be modified to "Logout". Then, the DAS sends a delete request to the in-memory database to remove the state data of the avatar. Both situations are showed in Figure 5.2.

The Process of Data Recovery

When a player restarts the game, the DAS first checks the player's log status in the TST.

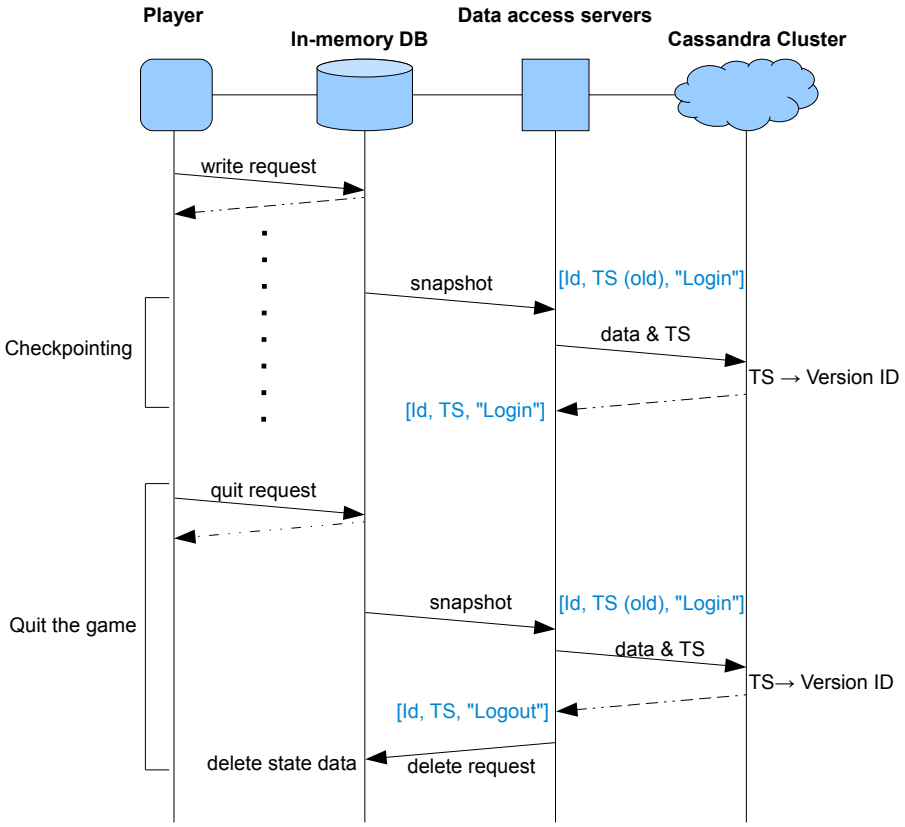


Figure 5.2: Flow Diagram of Checkpointing

Read (1): If the value is "Login", that means the previous checkpointing is not yet completed, so the up-to-date state data of her/his avatar is still hosted in the in-memory database. In this case, the state data do not need to be recovered, and data will be directly fetched from the in-memory database (see Figure 5.3 Read 1).

Read (2): If the value is "Logout", the DAS gets the timestamp from the TST, and then uses TS and Id as query criteria to retrieve the up-to-date checkpoint with CL *ONE*. When a replica in Cassandra receives the request, it compares the TS with its own TS. If they match, the state data will be returned. Otherwise, a null value will be sent back. In this case, the DAS has to increase the CL and send the read request again until the up-to-date checkpoint is found or all available replicas have been retrieved. If the expected version still has not been found, the latest version (but stale) in Cassandra has to be used for recovery. At last, the player's log status in the TST will be modified from "Logout" to "Login" (see Algorithm 2 and Figure 5.3 Read 2).

```
Input: an avatar/game object's UUID
Output: state data of the anvatar/game object
begin
    Id ⟵ avatar's/game object's UUID

    //get the version ID of the avatar/game object from the timestamp table
    (TST)
    TS ⟵ TST.getTS(Id)

    //set read consistency level to ONE
    CL ⟵ ONE
    data ⟵ null

    //get state data from Cassandra
    while data == null and CL ≤ number of available replicas do
        //get state data based on the UUID and version ID
        data ⟵ Cassandra.get(Id, TS) with CL

        //Check whether the result returned is null
        if data == null then
            //if did not get any result meeting the retrieval conditions

            //increase the read consistency level to check more replicas
            CL++
        end
    end

    //Check whether the result returned is null
    if data == null then
        //if all available replicas have been checked, but still not got any result

        //get state data from all available replicas
        CL ⟵ number of available replicas
        dataSet ⟵ Cassandra.get(Id) with CL

        //get the state data with the highest timestamp
        for d ∈ dataSet do
            if data.TS < d.TS then
                data ⟵ d
            end
        end
    end

    return(data)
end
```

Algorithm 2: The Process of Data Recovery

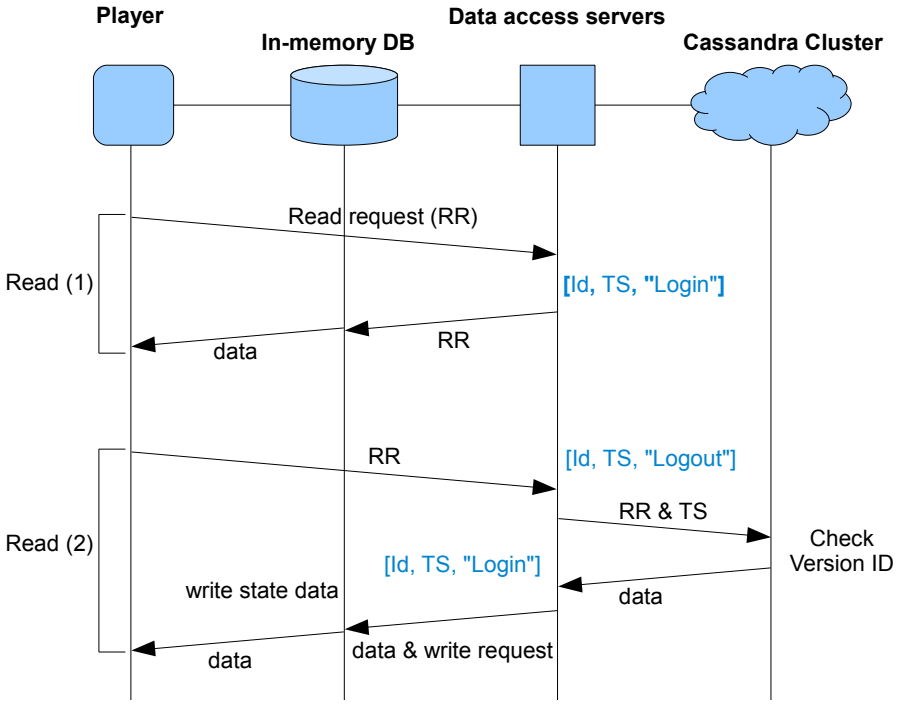


Figure 5.3: Flow Diagram of Data Recovery

5.1.2.3 Optimisation using a Node-aware Policy

The timestamp-based solution can obviously improve the efficiency of data recovery, but from the description above, there is an issue:

New issue description: if the first attempt of retrieval fails, the read operation has to be executed again with a higher read consistency level, which increases the response time. Therefore, we can conclude that the success rate determines the read performance.

The reason is that the read request is executed by a replica node, which does not host the up-to-date checkpoint. For instance, Figure 5.4 shows the process of writing a checkpoint and its subsequent operation of reading the checkpoint. The up-to-date checkpoint is hosted by node B (replica 1). Unfortunately, the coordinator has forwarded the read request to node C (replica 2), which hosts a stale checkpoint. In this case, a null value would be returned to the client, and the read operation has to be executed again.

To optimize our timestamp-based solution, we propose to sacrifice a part of database transparency in exchange for the success rate. In other words, the IP address of the replica node that has performed the last checkpointing, will also be recorded in the TST. For subsequent read requests on this checkpoint, the DAS will connect to that

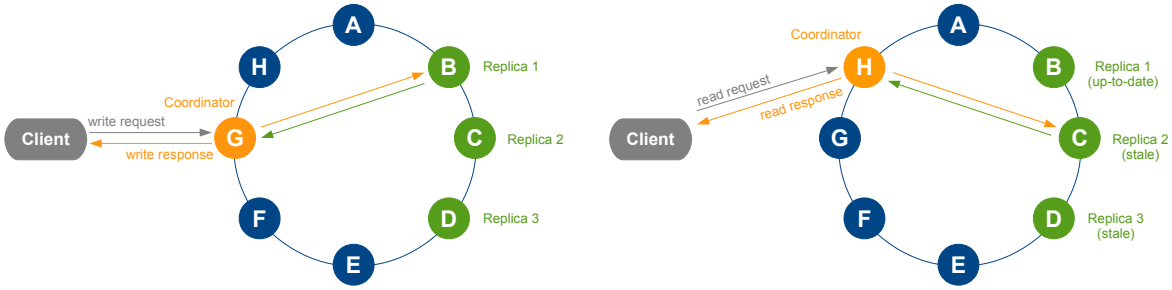


Figure 5.4: Process of Executing Write and Read Operations in Cassandra Cluster

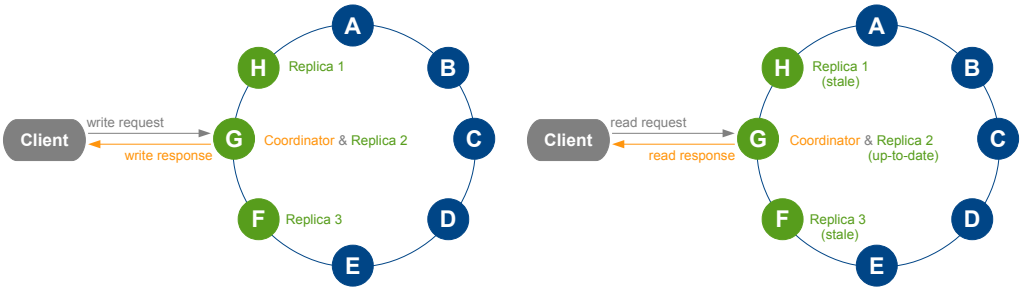


Figure 5.5: Process of Executing Write and Read Operations Using *NodeAwarePolicy*

node (as a coordinator) directly. In this case, the success rate will be increased if that node is still available (see Figure 5.5).

We can understand this strategy like this for write operations each replica is still identical as before, but for read operations there is a "primary" replica. For this reason, our proposal will not affect the system availability. The checkpoint could still be flushed to any replica as before; if that replica node fails, a read request could be executed by the other replica nodes. In our project, we name this strategy as *NodeAwarePolicy*.

Comparison with the *TokenAwarePolicy* in Java Drive

It is noteworthy that the Cassandra tool, Java Drive, provides a *TokenAwarePolicy* for load balancing (see Section 3.4.6), which has a similar function with our *NodeAwarePolicy*. However, there are the following differences:

1) For each write/read using the *TokenAwarePolicy*, only the node hosting the first replica (determined by the token value of a data object's partition key) will be used as a coordinator. In other words, each replica node is not identical any more, instead there is a "primary" replica. So the system performance will be affected if the workload of the "primary" replica node is heavy, or the physical distance between that node and the Cassandra client initiating a request.

2) By using the *TokenAwarePolicy*, the Cassandra client only maintains a list of IP addresses of all nodes in the ring, which can be easy to obtain. In contrast, the

NodeAwarePolicy needs to record the host IP address for each data object on the server side, so we have to consider the persistence of these information (we will discuss that in the next subsection.).

3) The *TokenAwarePolicy* is more suitable for a changing environment, where the IP address of a node change frequently. It is because that Java Drive updates the information of the ring regularly. However, it could becomes a problem for *NodeAwarePolicy*, which only updates the IP address after a write operation is successfully performed.

4) The *TokenAwarePolicy* cannot get information of all replica nodes of a data object because it has no idea about the *replication factor* and replica placement strategy of the ring. As a result, if the "primary" replica node is unavailable, a random node will take its place as a coordinator. In contrast, *NodeAwarePolicy* considers the node failure, thus all these information will be collected. If a replica node fails, the other replica nodes will be first used as an alternative.

In summary, our *NodeAwarePolicy* has a better write/read performance than the *TokenAwarePolicy*, especially in an unstable environment, where node failure occurs occasionally. We will give an experimental proof later in Section 6.4.

5.1.2.4 System Reliability

To prevent a single TST from becoming a bottleneck of the system, several TSTs on different nodes need to work in parallel. The checkpoint information (e.g., TS) is assigned to different TSTs depending on the location (game world/map) of avatars/objects. If one of the servers hosting TSTs goes down, a new/another TST takes the place of its work immediately. A TST failure will not affect the gameplay. The lost data (e.g., TS) could be regained by accepting new checkpoints (for active avatars), or recovered by fetching relevant information (such as getting the biggest version ID of the checkpoints of one avatar/object) from Cassandra cluster (for inactive avatars). However, during the recovery, the client has to perform a read *ALL* to get the up-to-date checkpoint.

5.1.3 Related Work

Some researchers proposed to extend the existing CDBMSs or design a new Cloud data store to provide ACID transactions [CBKN11, GBS11, WPC09, DEAA09, PD10, VCO10, GS11, LLMZ11]. That means the new systems could provide ACID updates to multiple rows at a time just like in RDBMSs. The new systems could certainly help to guarantee the game consistency, which is lower than strong consistency. However, comparing with our proposal they are too general and not specific to the game scenario. As a result, the new systems have to sacrifice performance and availability to achieve some unnecessary features for online games.

Similarly, in [DAA10], authors have although analyzed the game consistency, they suggested supporting strong consistency by extending an existing CDBMS, which will affect

the system performance. Our timestamp-based solution could guarantee data currency in an eventually consistent environment.

The possibility of using the timestamp as a version ID to identify current data in a replicated database has already been studied. However, these solutions either focus on some other storage systems or are designed for other application scenarios. For instance, in [APV07], authors have proposed a new key-based timestamping service to generate monotonically increasing timestamps, which is designed for distributed hash tables with a P2P structure. The main objective is to synchronize timestamps in a P2P environment and to scale out to large numbers of peers. The testbed in this work takes a client/server structure, where timestamp synchronization is not problematic. Improvement of the efficiency and accuracy of queries in a NoSQL DBMS (e.g., Cassandra) is our research focus.

In [GBS11], authors also record the version ID got from a Cloud storage system for data currency. However, they suggested that if a read request fetched an old version ID from the Cloud, the system should wait for some milliseconds and try it again. Consequently, this approach blocks the read operation and increases unnecessary response time, which is not suitable for MMORPGs.

5.2 Read Performance

Cassandra is designed to provide a high performance of writes, rather than reads. Hence, its read performance is not as high as its write performance, especially in our application scenario.

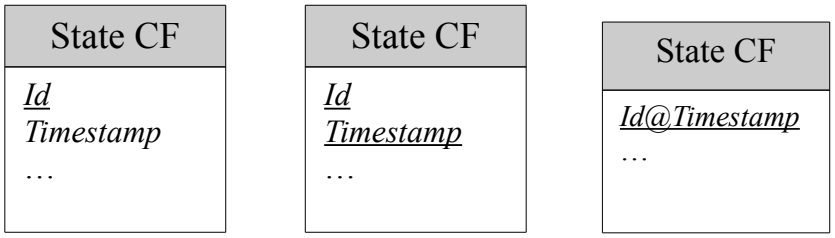
5.2.1 Issue Description

In the following, we discuss this issue from internal and external aspects.

Issues:

1) We proposed to persist the state information of an avatar/object in a single row in Cassandra. As introduced in Section 3.4.2.3 Cassandra stores data in *SSTables* and *Memtables*. When a read request against a row comes in to a replica node, all *SSTables* and any *Memtables* that contain columns from that row must be combined. For this reason, the more fragments the row has, the more CPU and disk I/O will be consumed. Unfortunately, Cassandra is used in our project to persist checkpoints. That means data in one row are frequently updated. If data are not well structured in a column family, it is easy to produce large amounts of fragments.

2) The row cache can be used in Cassandra to avoid fetching data from disk (see Section 3.4.4.2), which works when a partition/row is frequently accessed. However, in our case, a checkpoint is at most used once. Therefore, this built-in mechanism is useless to us.



(a) Option I (b) Option II (c) Option III

Figure 5.6: Different Designs of State Column Family (State CF)

5.2.2 Proposal of the Data Structure in Column Family

Since we cannot cache all checkpoints in memory, we have to read them from disk. Cassandra works best when it executes a single sequential operation, so we must keep this best practice in mind while designing the data structure in Cassandra.

We have proposed to store a timestamp with the checkpoint. There are three possible data structures to implement it (see Figure 5.6). We will explain and discuss their advantages and disadvantages in the following.

Option I

We store the timestamp as a "regular" column (see Figure 5.6a).

Advantages: the old values will be deleted when storing a new checkpoint. That means, if a replica node has executed an update successfully, there are no stale data on this node any more. Hence, the database size will not grow unlimitedly.

Disadvantages: columns of a row are fragmented in several *SSTable* files because new values will be stored in a new *SSTable* (see Section 3.4.4.1). As a result, data will be then collected from multiple *SSTable* files, which affects the read performance; furthermore, a secondary index on the timestamp column needs to be created, if we want to use the timestamp as a criterion to look up data. However, the timestamp is changed too frequently in our application scenario. Maintenance of such an index affects the write performance.

Option II

The second approach is to use the avatar's/object's ID and the timestamp as compound primary key (see Figure 5.6b). In this case, the timestamp is used as a clustering column.

Advantages: the backup of a checkpoint is simplified to an insert operation (not update). Accordingly, an entire checkpoint will be flushed to a simple *SSTable* file. Therefore, fetching of a checkpoint only reads from disk sequentially. Moreover, we do not need to create a secondary index on the timestamp column so as to use it as a query criterion.

Disadvantages: the stale data could not be removed automatically by compacting SSTables because Cassandra does not know they are out of date. The database size may grow unlimitedly, which will affect the system performance. Hence, an additional process is required to delete these data.

Option III

We can also combine the avatar's/object's ID and the timestamp (separated by a special symbol like @), and use it as a primary key (see Figure 5.6c).

This design has all the advantages and disadvantages of the second option because each checkpoint is also stored in a single *SSTable*. The difference between them is the partition key. In the option II the avatar's/object's ID is used as the partition key, so all checkpoints of one avatar/object could be fetched from one node. However, in the option III the partition key is the primary key, the value of witch is unique. That means, checkpoints of one avatar/object are distributed on different nodes (probably all nodes in the cluster), which brings some new advantages and disadvantages.

Advantages: it offers higher fault tolerance. By using the option II (as well as the option I), if all replica nodes hosting the state data of one avatar/object are down, checkpointing of this avatar's/object's state data has to be blocked until one of the replica nodes recovers. By using the option III, the replica nodes are not fixed because the value of each checkpoint's partition key is different. So if a latest checkpoint has been successfully flushed to disk, the backup of the blocked checkpoint could be ignored.

Disadvantages: frequent changes (adding and removing) of row keys cause a heavy overhead for maintaining the primary index. Furthermore, when the timestamp table on the server side fails, fetching the latest checkpoint is problematic. CQL does not support a 'like' query as in SQL. That means, currently, it is impossible to get the checkpoint by only using an avatar's/object's ID.

From the above discussion, we can reasonably reach a conclusion that the option II is more appropriate for our application scenario. In addition, this schema could also be applied by the log column family, so that all log information of a player could be ordered by the timestamp and stored in the same replica node.

5.3 Mapping Database Schema

In our Cloud-based architecture, a checkpoint is frequently transferred between an RDB (the in-memory database) and a NoSQL store (Cassandra). Hence, there are some new requirements for the design of the database schema in Cassandra.

5.3.1 Issue Description

We explain the new issue in the following:

Issue:

The design concept of these two kinds of database is different. For example, tables in the RDB are often normalized in order to minimize data redundancy, so the result of a query is always gotten from multiple tables. In contrast, tables (column families) in Cassandra are denormalized to improve the query performance, so the result is gotten from a single row. For this reason, we need a middleware on the server side to transfer data between a multi-table schema and a single-column family schema. Consequently, the structure of a column family must be well designed, so that the middleware can work efficiently and the change of the structure of a table will not cause a wide range of modification of the middleware program.

5.3.2 Proposal of the Structure of a Column Family

Figure 5.7 illustrates an example. In the following, we will provide two possible solutions to map these tables to a single column family in Cassandra.

Solution I:

In our previous work, limited by the capability of the early CQL and Cassandra, we simply proposed to design the structure of a column family like in Figure 5.8 [DSWM13]. Since the column name in Cassandra must be unique, we have to rename the columns in the Inventory table. Although this is feasible, it brings a number of problems:

1) The middleware must be aware of the naming rules in Cassandra. As long as the rules have been changed, the middleware program has to be modified to adapt to it either. Otherwise, it will inevitably lead to system errors. Furthermore, the program does not have the versatility, which is hard to be used for other games.

2) Columns in a column family are ordered by their names. After we map multiple tables in an RDB to a single column family, all columns will be reordered. That means, columns once belonged to the same table will not be stored sequentially, which increases the processing time of the middleware to map results/checkpoints back to the in-memory DB.

Solution II:

With the help of Cassandra 2.1 and later, we can use collection types and a user-defined type (UDT) to improve the structure of a column family (see Section 3.4.1). Figure 5.9 shows a sample of the new data structure of character column family. *Inventory* is the name of a map column, and <*quantity:10, location:1*> is the value of a UDT (The implementation of this structure will be illustrated later in Section 6.3). Through using this structure, we can order all columns of a table together, and use their own names. Accordingly, all problems mentioned above have been solved.

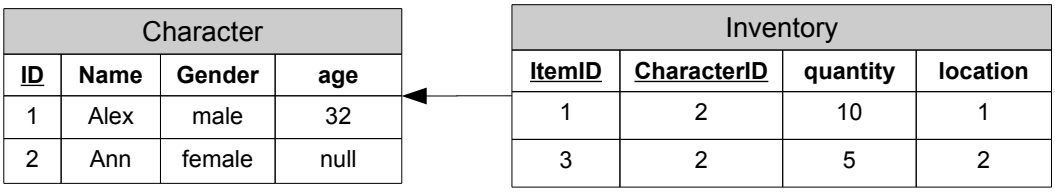


Figure 5.7: An Example of an RDB Schema

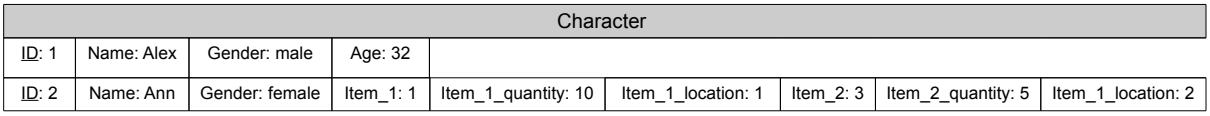


Figure 5.8: Solution I: A Possible Data Structure in Cassandra

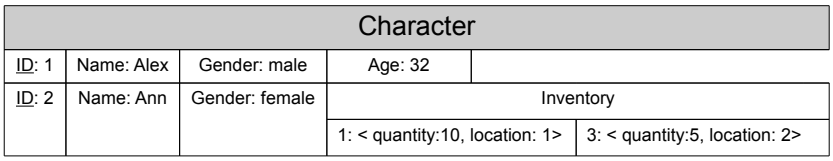


Figure 5.9: Solution II: A Possible Data Structure in Cassandra

5.4 Other Issues

Cassandra is not designed for online games, so there are some inherent mechanisms or characteristics, which may bring some troubles.

Read Repair:

Read Repair is used to guarantee eventual consistency (see Section 3.4.3). However, in our use case, we do not mind whether the data in the cluster are consistent or not after fetching the checkpoint. Performing this request consumes unnecessary system resources, which affects the throughput of the cluster. Although the default probability of performing it is reduced to 0.1 in Cassandra 1.0 and later, we still propose to disable it.

Query capability:

CQL and Cassandra are developing rapidly, but their query capability is not as strong as RDBMSs. As a result, game developers have to consider all possible queries before they design a column family. That will increase the difficulty of developing games and limit the expansion of the game functionality.

5.5 Summary

In this chapter, we have discussed the potential issues we have to face when using Cassandra for backing up checkpoints. Some possible solutions like a timestamp-based model have been proposed to address these issues. However, for some issues we can only rely on the further development of Cassandra. In the next chapter, we will introduce the implementation of the Cloud-based game testbed and some other testbeds in detail, and then evaluate and compare them.

6. Evaluation

> Part of the experiment results in this chapter are originally published in an article "CloudCraft: Cloud-based Data Management for MMORPGs" [DWSS14], the DB&IS paper "Cloud-based Persistence Services for MMORPGs" [DWS14], and the IDEAS paper "Achieving Consistent Storage for Scalable MMORPG Environments" [DZSM15].

Under the *CloudCraft* project, we have run a number of sub-projects in order to verify our proposal of a Cloud-based MMORPGs. In this chapter, we classify these sub-projects into three groups, namely the potential scalability of a Cloud-based game system, the performance comparison of MySQL Cluster and Cassandra in MMORPGs, and the efficiency of guaranteeing the Read-Your-Writes consistency. Next, we will introduce them one by one.

6.1 Experimental Infrastructure

We have built various game prototypes for different evaluation purposes. These prototypes differ in the system architecture, the database schema, the Cassandra configuration, and so on. However, they have the same experimental infrastructure.

For carrying out experiments, our faculty provides eight virtual machines with the Ubuntu operating system, each of which configures 2.40 GHz CPU, 8 GB memory and 91 GB hard disk (see Table 6.1). For security reasons, these virtual machines cannot be visited from outside directly. A client needs firstly to connect a stepping stone server through the secure shell (SSH) protocol, and then get access to the virtual machines via it indirectly (see an example in Figure 6.3).

6.2 Experimental Proof of the System Scalability

In this sub-project, we aim at providing a proof of concept for Cloud-based online game, as well as evaluating the scalability and performance of it. For this purpose, we have

Computer System	8 virtual machines
CPU	Intel(R) Xeon(R) E5620 2.40 GHz
RAM	8 GB
Disk	90.18 GB, 7200RPM
Network	100MBit/s
Operating System	Ubuntu 13.04 (64 bit)
Java version	1.7.0_25
Programming language	Java

Table 6.1: Experimental Infrastructure

designed and implemented a prototypical game platform, which borrowed the design from an open source MMORPG test environment and ported it to Cassandra [Wan13].

We have to point out that physical resources for the experiments were limited as described below, so the focus is mostly on scaling the number of clients versus a small set of up to five Cassandra servers. Nevertheless, we got some interesting results.

6.2.1 Prototype Architecture

Figure 6.1 shows the architecture of our game prototype, which consists of a client side and a server side. The client side can be scripted to support experimental setups of thousands of players; the server side is responsible for handling requests from game clients and managing the various data sets in the game. There are four layers at the server side, namely, the communication layer, the game logic layer, the data access layer, and the physical storage layer. The game client and the game server communicate via a socket server, which we named the communication layer; the game logic layer is responsible for handling commands sent by players and dealing with game logic; the data access layer is used for communication between the logic layer and the storage layer; the physical storage layer performs data accessing operations and hosts data in the game. As we have in the previous chapter proposed, Cassandra cluster is applied at the physical storage layer.

6.2.2 Implementation of the MMORPG Environment

Our research focuses on analyzing the influence of using a Cloud storage system for MMORPGs rather than designing a real and complex online game. Therefore, a simplified but robust game client and game server supporting basic game logic suffice to fulfill our experimental requirements.

6.2.2.1 Implementation of the Game Client

We have implemented a game prototype based on an open source project JMMORPG[1], which consists of a simple Java game client and a game server running on an RDBMS. We have used the architecture and the client GUI (Graphical User Interface) of it, such as avatar figures and maps (see Figure 6.2).

[1]JMMORPG project:http://sourceforge.net/projects/jmmorpg/ (accessed 20.02.2014).

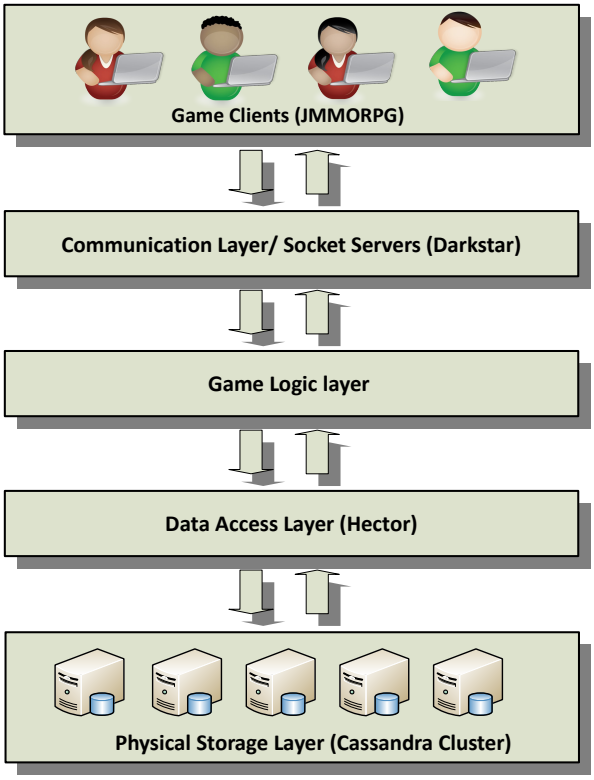


Figure 6.1: Architecture of the Game Prototype

Figure 6.2: GUI of the Game Client

6.2.2.2 Implementation of the Game Server

We have rebuilt the game server based on the JMMORPG project, and persisted data in the Cassandra cluster.

The communication layer in the prototype is built based on the Darkstar project[2], which provides a convenient functions library to help developers to deal with the challenging aspects of networked game development [Bur07, BW09]. There are three core components in the Darkstar, namely DataManager, TaskManager and ChannelManager. All of the game state objects are represented by Java objects in Darkstar. Therefore, the management of concurrent data access becomes a challenge. Darkstar provides a DataManager interface to handle concurrency; each client communication (read/manipulate data) generates a task, which is transactional, independent and short-lived (100 ms). It is the smallest executable unit in Darkstar. A TaskManager is used for scheduling and creating a single task; a ChannelManager is applied in Darkstar to create and manage the channel, which is a communication group consisting of multiple client sessions and the server.

In the logic layer, we have simulated some basic game logic, such as responding to commands ordered by clients (e.g., players' login requests and avatars' movements) and supporting interactions among players (e.g., chatting and trading), all of which involve querying the database.

We have applied a high-level Java API (Hector[3]) for the data access layer, which makes it possible to access Cassandra through an RPC (Remote Procedure Call) serialization mechanism.

Furthermore, in the Cassandra cluster, we have implemented several column families for accounts, avatars, NPCs, logs, maps, inventories and items, which have the structure like in Figure 5.8.

6.2.3 Experimental Setup

The game prototype was running on Cassandra 1.2.5, which was the latest stable version when we carried out the experiment. At most three virtual machines were used to deploy the game server. The number of nodes in the Cassandra cluster was set from three to five. Figure 6.3 shows the infrastructure of the prototype.

We also have implemented a simplified command-line game client for the experiments because it consumed less system resources and works like the GUI client. Our benchmark was a player's normal behavior, such as moving and trading. From data management perspective, the essence of these operations is performing writes/reads to the database. We have created one row for each avatar in the avatar column family to host its state data, each of which consists of 20 columns and has 540 bytes (row size). The

[2]DarkStar website: http://sourceforge.net/apps/trac/reddwarf/ (accessed 20.02.2014).

[3]Hector website: http://hector-client.github.io/hector/build/html/index.html (accessed 20.02.2014).

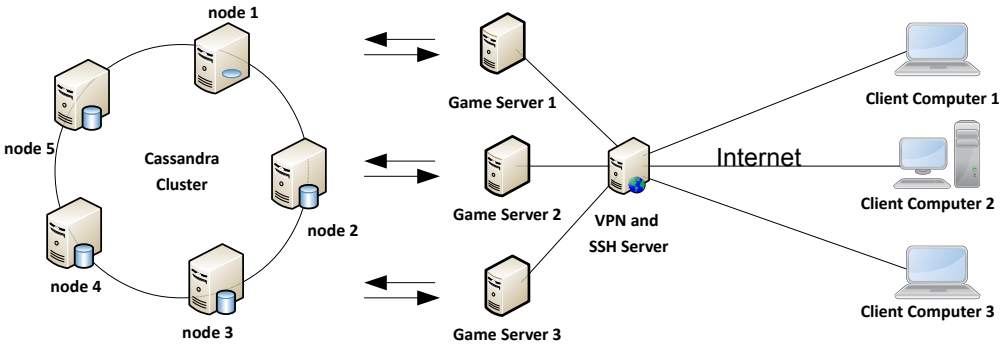


Figure 6.3: Infrastructure of the Game Prototype [DWSS14]

game client randomly ordered a write/read command regarding one of those columns, and then sent it to the game server. Meanwhile, the response time of each command has been recorded.

The evaluation focuses on the potential scalability and performance of our prototype in the case of multi-player concurrent accesses. In the experiment, we will change the number of nodes in the Cassandra cluster from one to five, so we keep the *replication factor* of the cluster to one (That means, the cluster has only one copy for each row, and a write/read command will succeed once one replica node responds to it.). Otherwise, if the *replication factor* is larger than the number of nodes in the Cluster (for example, in a single node Cassandra, the *replication factor* is specified to three), the system throws an exception when executing a command.

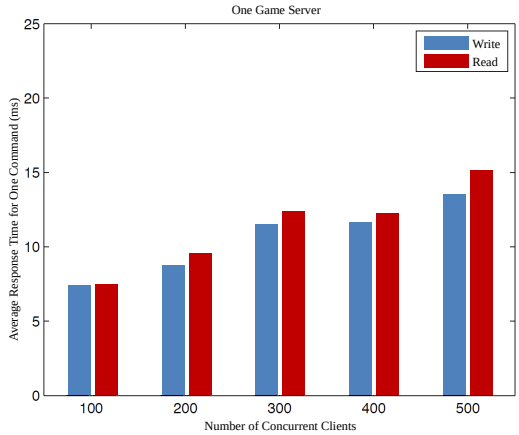
6.2.4 Experiments

We have evaluated the scalability of the game server and Cassandra cluster in an online game scenario separately.

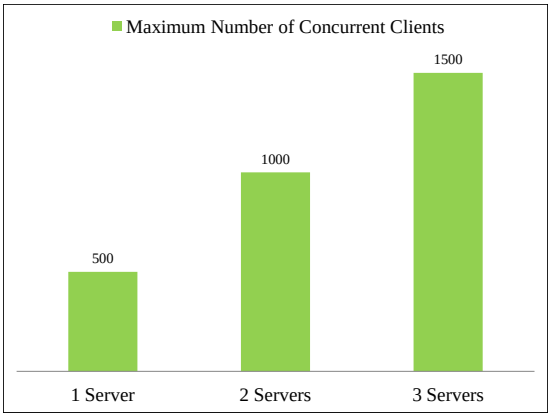
6.2.4.1 Scalability of the Game Server

Through this experiment we wanted to get the maximum number of concurrent clients that our game server can support. Therefore, we have fixed the number of nodes in the Cassandra cluster to five, and added up to three game servers during the experiment. The number of concurrent clients connecting to the server was increased from 100 to 1500. Each client randomly sends 500 write/read commands. We calculated then the average response time (total run-time of all concurrent clients/(500*number of concurrent clients)) for one write/read command.

We present the experimental result with a single game server in Figure 6.4a. When the client number is not more than 500, the average response time for each read/write command is under 15 ms. That means, 500 concurrent clients put little pressure on the game server as well as the 5-node Cassandra cluster. However, when the client number is up to 600, the game server throws many “time-out” exceptions, which block



(a) Average Response Time (Calculated from (500*Number of Concurrent Clients) Commands) of a single Game Server Connected by Different Number of Concurrent Clients



(b) Maximum Number of Concurrent Clients Supported by Different Number of Game servers

Figure 6.4: Scalability of the Game Server Connecting with Five-node Cassandra

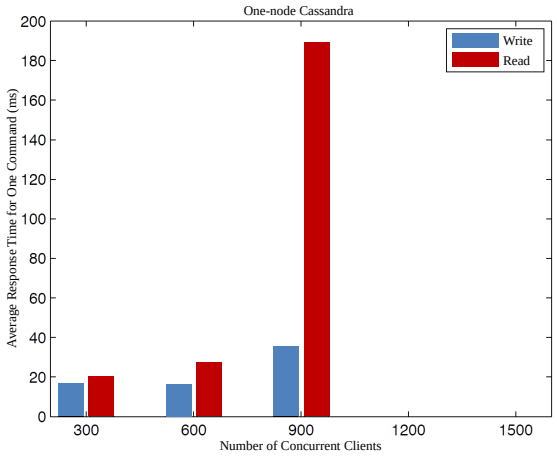
the acceptance of subsequent commands. (The default maximum amount of time that a transaction will be permitted to run before being aborted is 100 milliseconds[4].) So the maximum number of concurrent clients in the case of single game server is around 500. Similarly, we found that the client number is directly proportional to the growth of the number of game servers (see Figure 6.4b). Therefore, we came to the conclusion that the total amount of clients is limited by the concurrent processing capability of the game server, whereas it could be raised easily by adding more servers.

6.2.4.2 Potential Scalability of Cassandra in an MMORPG

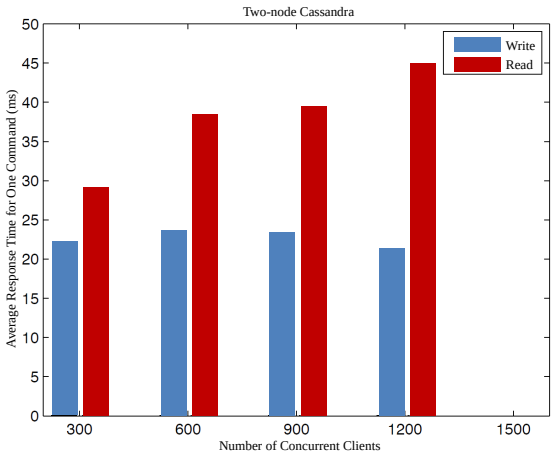
Scalability of a database is reflected by its ability that by increasing the number of database nodes to improve database performance. Hence, this time we have fixed the number of game servers to three, and set the node number in the Cassandra cluster from one to five. Each game server is connected by 100, 200, 300, 400, and 500 clients in turn. That means, the Cassandra cluster handles 300, 600, 900, 1200, 1500 clients separately. Every client sends 500 read or write commands. The corresponding response time of each command is recorded and afterwards the average response time is calculated.

From Figure 6.5a we can find that, a high performance of one-node Cassandra is achieved for less than 600 clients. When the number of clients reaches 900, the response time of read operation increases sharply over 180 ms, which is unexpected. If we start 1200 clients, the Cassandra cluster will not respond to the write and read request normally. Many clients report a connection time out exception because of limitation of Cassandra I/O. Thus, we terminate the first-round experiment and conclude that one-node Cassandra can only support up to 600 clients in our experimental environment.

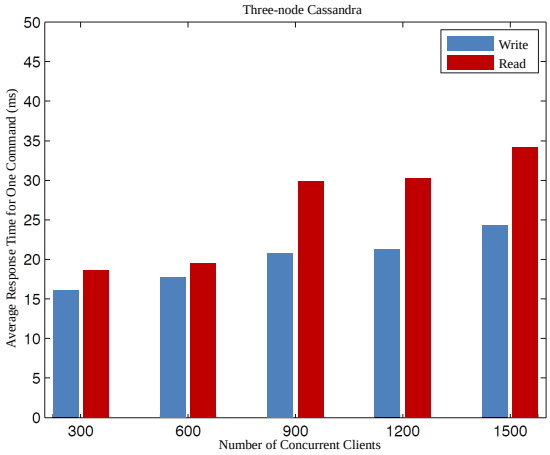
[4]Darkstar configuration file:http://grepcode.com/file/repo1.maven.org/maven2/com.projectdarkstar.server/sgs-server/0.9.8.10/com/sun/sgs/app/doc-files/config-properties.html (accessed 20.02.2014).



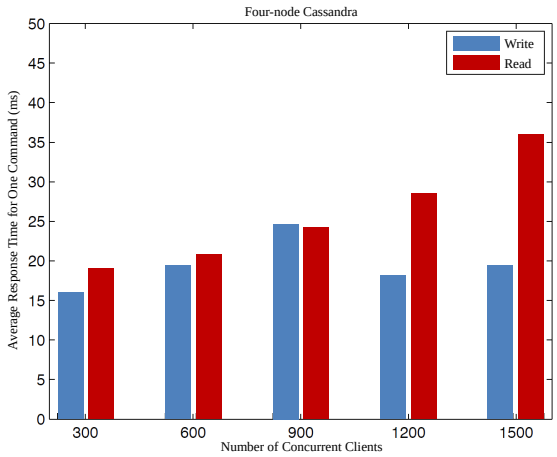
(a) Performance of One-node Cassandra



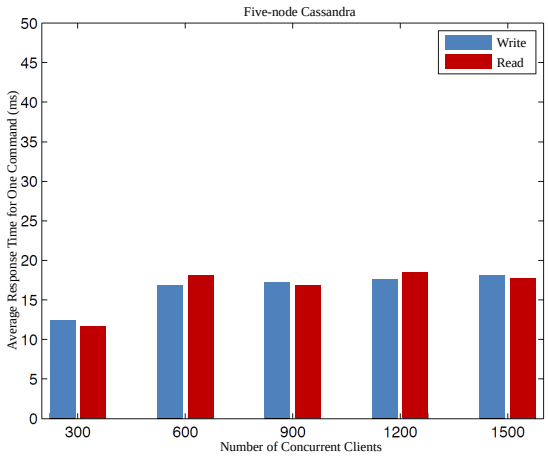
(b) Performance of Two-node Cassandra



(c) Performance of Three-node Cassandra

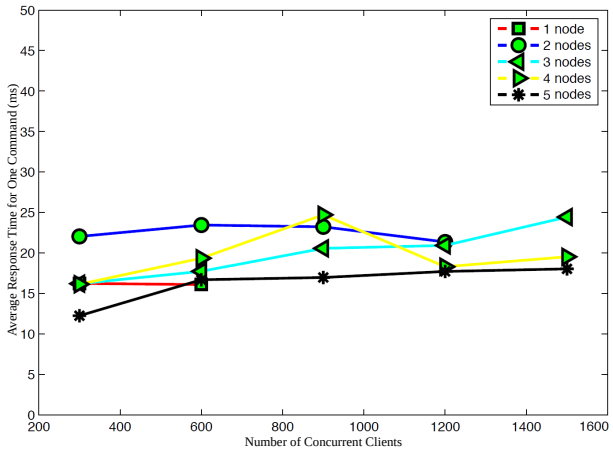


(d) Performance of Four-node Cassandra

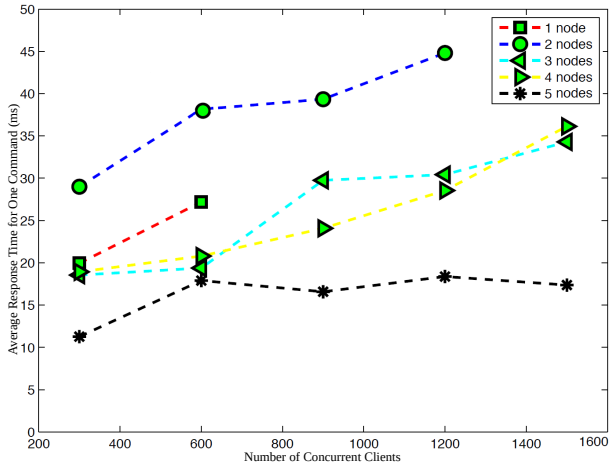


(e) Performance of Five-node Cassandra

Figure 6.5: Average Response Time (Calculated from (500*Number of Concurrent Clients) Commands) of Cassandra Cluster Connecting with Three Game Servers



(a) Comparision of Write Performance



(b) Comparision of Read Performance

Figure 6.6: Comparison of Write and Read Performance of Different Cassandra Clusters (Node Number from One to Five) Connected by Various Number of Concurrent Clients (from 300 to 1500)

Figure 6.5b shows that the maximum number of clients reaches 1200 when there are two nodes in the Cassandra cluster. In the case of 1500 concurrent connections, the issue of timeout appears again. Therefore, we conclude that a two-node Cassandra cluster can support about 1200 clients by using our prototype.

Figure 6.5c, 6.5d and 6.5e present that, when the number of nodes in Cassandra cluster is more than three, our prototype can support at least 1500 concurrent players.

In order to observe the different results in Figure 6.5, we plot the writing and reading response time in Figure 6.6a and 6.6b.

According to the experimental result, we can observe the following tendency:

1. The number of concurrent players supported by our prototype can be increased (from 600 to 1500) by adding more nodes into the Cassandra cluster.

2. Cassandra achieves a satisfactory writing performance (around 20 ms), which is relatively better than the reading performance. Furthermore, the change in the number of nodes has little influence on writing performance (concentrated between 15 ms and 25 ms). In contrast, reading performance is obviously improved by adding nodes. In the case of five-node Cassandra cluster, reading and writing performance tends to become similar.

3. The five-node Cassandra cluster exhibits the best and most stable performance in all range of clients' number. With increasing number of clients, there is no obvious variation of reading and writing response time. Both of them fluctuate around 15 ms.

4. Generally, the system performance has been improved by scaling out the Cassandra cluster. For example, five-node Cassandra has the best performance; three-node and four-node Cassandra are observably better than two-node cluster. However, there are still some exceptions. An example is that the performance of three-node and four-node Cassandra is similar. Theoretically, four-node Cassandra should be better. However, our experiment shows some contrary results, such as reading response time at 1500 clients and writing response time at 900 clients. It may be caused by network latency, system configurations, or even some internal processing mechanism of Cassandra. Unfortunately, our prototype cannot reveal the reason.

5. One-node Cassandra shows a better performance in the case of 300 or 600 clients. The reason could be that the advantage of a multi-node Cassandra cluster is not outstanding when the number of concurrent players is relatively small. In addition, the communication between nodes also consumes some time since data are distributed on different nodes.

Based on the analysis above, we can conclude that a NoSQL DBMS like Cassandra exhibits a satisfactory scalability for typical MMORPG requirements. With increasing numbers of clients, the database performance encounters a bottleneck. However, the database throughput as well as response time can be improved easily by scaling out the cluster; Cassandra shows a high performance in the experiment. The response time of writing and reading typically fluctuates between 10 ms and 40 ms, which fulfills the requirement of an MMOG [CHHL06]; Cassandra is a write-intensive database. The experimental results show that its writing performance is stable and excellent. This feature makes it suitable to perform a backend database of a multi-player online game, which needs to handle more write requirements.

Figure 6.7: A Screenshot of PlaneShift[7]

6.3 Comparative Experiments of System Performance

After building a scalable game prototype, our research focus switches to compare the system performance of a NoSQL DBMS (Cassandra cluster) and a conventional RDBMS (MySQL Cluster in this experiment) when checkpointing and recovering state data of MMORPGs. The database schema using in the JMMORPG project is in this case relatively simple, which cannot satisfy our new experimental requirements. For this reason, we have changed our experimental subject to another open source project, PlaneShift[5], which is closer to a practical commercial MMORPG.

6.3.1 A Practical Game Database Case Study: PlaneShift Project

PlaneShift is a 3D MMORPG under heavy development [Pan]. Figure 6.7 presents a screenshot of PlaneShift. Its latest source code could be downloaded from the website SourceForge[6]. Since this game is in the beta stage of the development, it still has many bugs and missing features, and its game server is not reliable. However, this game is created to meet commercial quality standards, so it has all the functionality that a practical MMORPG should have, including combats, magics, crafts and quests. Furthermore, state data of character and the virtual world are persisted on the server, so a user can reconnect at any time to continue the game. As a result, this game has a complete game architecture as well as a complex database schema like in a commercial MMORPG.

PlaneShift uses MySQL to manage all game data. The database design and even the scripts[8] to access the database are provided on the website. The database schema of

[5]PlaneShift project:http://www.planeshift.it/ (accessed 10.10.2015).

[6]PlaneShift source code:https://sourceforge.net/projects/planeshift/ (accessed 10.10.2015).

[7]A Screenshot of PlaneShift: http://www.planeshift.it/element/%5B%5D/Picture%20of%20the%20day/planeshift_333.jpg (accessed 20.12.2015).

[8]PlaneShift source code:https://github.com/baoboa/planeshift/tree/master/src/server/database/mysql (accessed 27.11.2015).

PlaneShift is presented in Chapter A. There are totally 92 tables in the database, which could be mainly divided into eight groups based on the game scenario, namely character tables, guild tables, NPC movement tables, NPC dialog tables, crafting tables, spells tables, items tables and Mini-game Tables[9]. These tables are designed to mange various data sets (see Section 2.1.3.2). For example, the accounts table is used to store players' account data; the gm_command_log table is applied to back up players' log data; state data in PlaneShift are respectively managed in character tables, NPC Movement tables, items tables, and so on.

In our experiment, we only focus on accessing the state data of character entities. For this reason, only nine tables are involved (see Figure 6.8):

Characters table : has total 61 attributes, such as id, name, account_id, loc_x (location coordinate) and bank_money_circles. It is the core table among these tables. Other tables refer to the *id* attribute of it.

Item_instances table : records the information of an item instance, like its owner (char_id_owner), guardian (char_id_guardian), creator (creator_mark_id), location coordinate and so on. It has 32 attributes.

Character_relationships table : is used to represent the relationship (e.g., family, buddy and spouse) between two characters, which has four attributes.

Other tables : including *character_events*, *character_traits*, *player_spells*, *character_quests*, *trainer_skills* and *character_skills* are associative tables (bridge tables). These table are used to resolve many-to-many relationships between *characters table* and another table in the database. For example, *character_skills table* maps *characters table* and *skills table* together by referencing the primary keys of them to present all skills that a character has.

6.3.2 Implementation of Testbeds

We have only borrowed the database schema of PlaneShift, rather than the entire project. Two testbeds using different kinds of databases have been implemented. Based on the experimental requirements, they only support a limited functionality. Since we will compare the database performance of adding, checkpointing and recovering state data, both testbeds support to insert, update and read data to/from the database.

6.3.2.1 Implementation of the Database using MySQL Cluster

In the testbed-MySQL, we have used MySQL Cluster 7.4.4 to manage data, and used JDBC to access the database. MySQL Cluster is deployed on five virtual machines. There should be at least one management server in the cluster to manage and monitor

[9]PlaneShift source code:http://planeshift.top-ix.org/pswiki/index.php?title=DatabaseDesign#Character_Tables (accessed 20.12.2015).

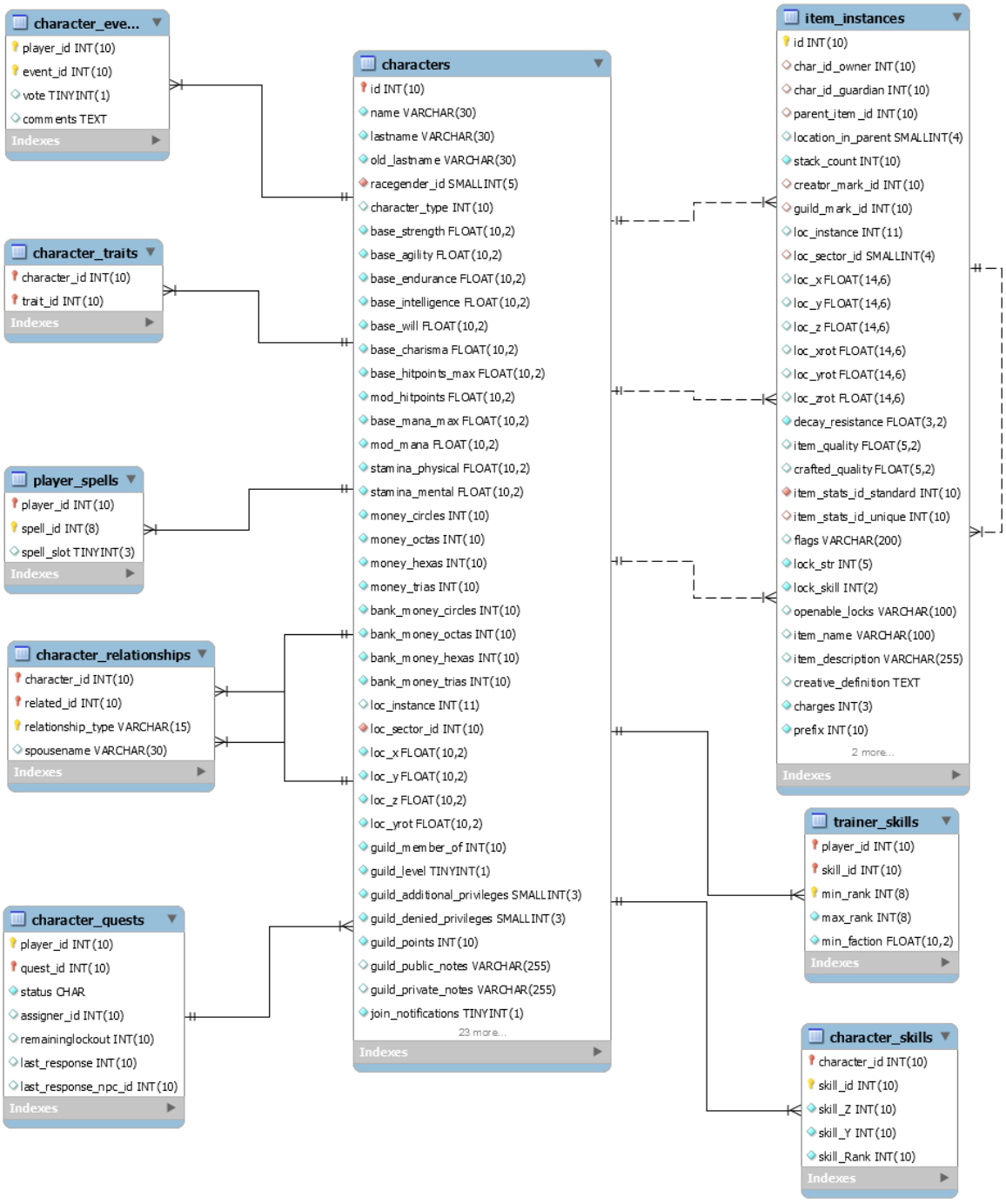


Figure 6.8: Character State Data Related Tables in the PlaneShift Database

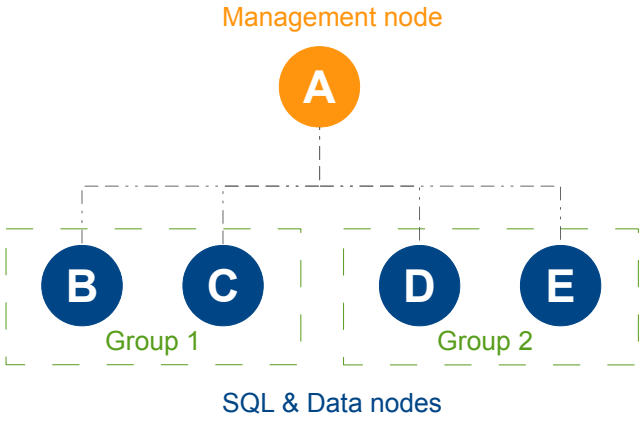


Figure 6.9: Database Architecture of Testbed-MySQL

all nodes. And it is recommended putting it on a separate node so as to avoid a server failure caused by other processes. Thereby, the cluster is configured with one management node. The other nodes are divided into two groups (number of replicas is two) and configured as both SQL and data nodes (see Figure 6.9). That means, data are distributed on only four nodes.

We have created nine tables in the database to imitate the database schema of PlaneShift (see Figure 6.10). Table names and the number of attributes keep the same with that of original tables. But we have simplified dependencies among tables. Each table has now only one foreign key related to the *characters table*. The name and type of some attributes also have been modified in order to simplify the code of the testbed. The impact of these modifications on the experimental results is negligible.

In practice, developers always use some advanced technologies or methods to optimize the performance of accessing RDBs. For this reason, Testbed-MySQL also supports prepared statements and stored procedures.

Prepared statement : is typically used with SQL statements, which is a feature in DBMSs used to repeatedly execute similar database statements with high efficiency. The statement is a template created with placeholders instead of actual values by the application and sent to the DBMS. At a later time, the certain constant values are passed to substitute placeholders during each execution. The statement is compiled by the DBMS once, so it enhances the performance considerably. Furthermore, using prepared statement can also protect the database from SQL injection.

Stored procedure : is a set of SQL statements with an assigned name and parameters (if it has) that is stored in the database in compiled form. Business logic could be embed in the procedure. The conditional logic applied to the results of a SQL statement can determine which subsequent SQL statements are going to be executed. Furthermore, it can be shared by a number of applications by calling

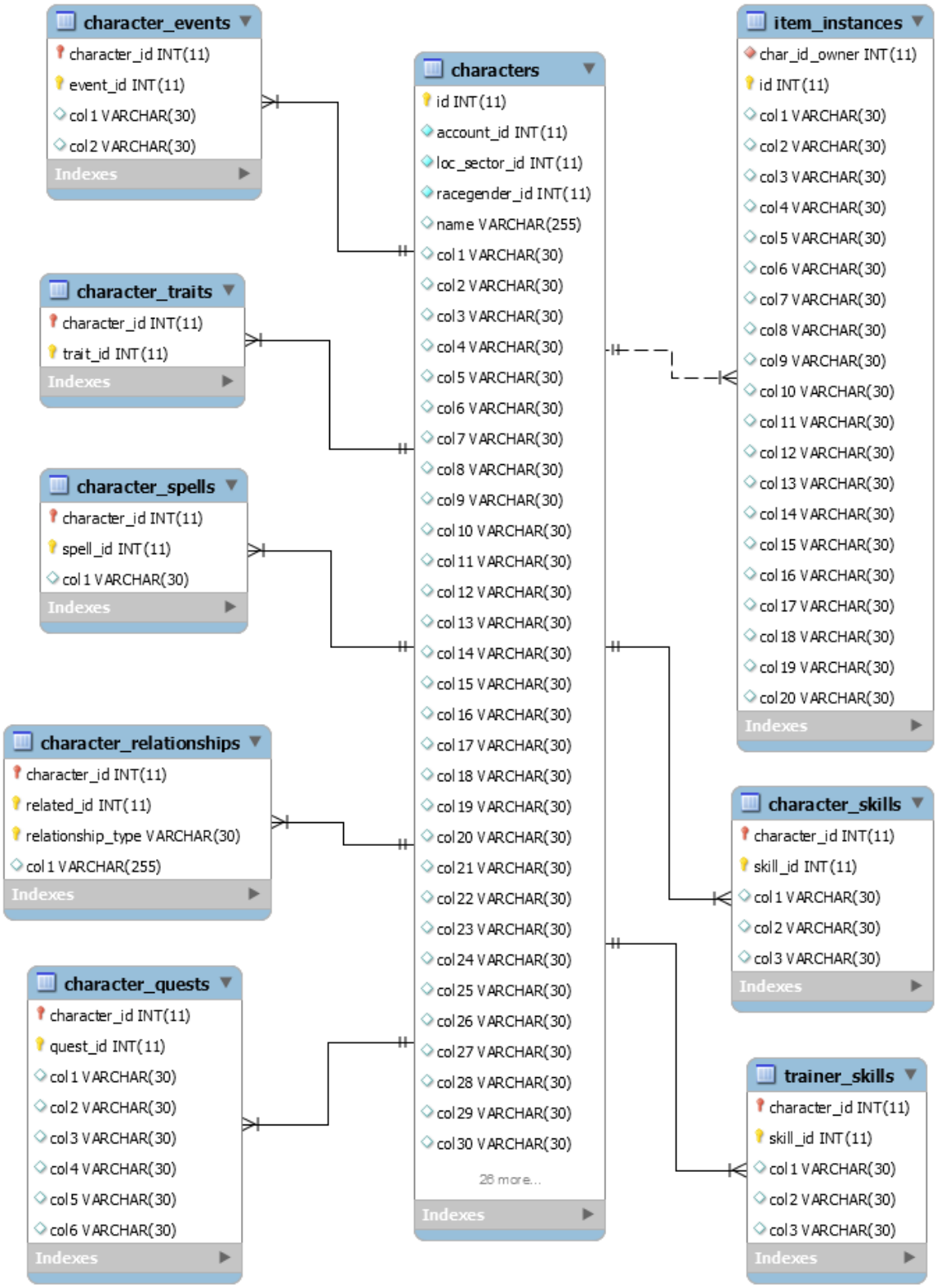


Figure 6.10: Database Schema of Testbed-MySQL

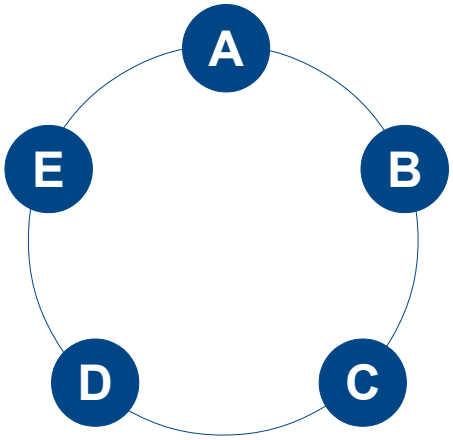


Figure 6.11: Database Architecture of Testbed-Cassandra

the procedure. A stored procedure is only compiled when it is created. Therefore, it improves the database performance.

We will use them on all three kinds of operations, and choose the best experimental result of each operation to compare with results from testbed-Cassandra.

Moreover, for data checkpointing, a strategy called *CopyUpdated* is adopted. That means, only the changed values will be updated into the database. This strategy can significantly reduce the number of operations for each checkpointing. In order to realize it, we have used an in-memory database, H2[10], in the testbed to store the information of the last checkpoint, which will be used to compare with the current one. Comparative results are used to determine things like which row/column in a table needs to be updated, which row needs to be removed, which data need to be inserted into a table.

6.3.2.2 Implementation of the Database using Cassandra

In the testbed-Cassandra, we have applied Cassandra 2.1.12 to manage data. On the client side, Java Driver is applied to access Cassandra. Similar with that in the Testbed-MySQL, we have deployed a five-node cluster (see Figure 6.11). Different with that in MySQL Cluster, all five nodes are responsible to store data. Furthermore, the *replication factor* is specified to two. Another significant difference is that there is only one table (*Characters table*) in the database, which is nested (see Figure 6.12). Other bridge tables have been mapped as map/set type columns of this table. That means, all information of one character is stored in one row. Listing 6.1 on page 89 shows the script for creating the *Characters table* by using CQL. The type and number of attributes in both testbeds are the same.

Cassandra also supports prepared statements, but not yet stored procedures. However, in this testbed, we have used none of them to optimize the performance because we

[10]H2 website: http://www.h2database.com/html/main.html (accessed 20.12.2015)

characters										
id	account_id	loc_sector_id	racegender_id	name	col1	...	col56	character_events	...	item_instances
								map<int, frozen <events>>	...	map<int, frozen <items>>

Figure 6.12: Database Schema of Testbed-Cassandra

	Testbed-MySQL	Testbed-Cassandra
DBMS	MySQL Cluster 7.4.4	Cassandra 2.1.12
API	JDBC	Java Driver
Number of Nodes	5	5
Number of Data Nodes	4	5
Number of Replicas	2	2
Number of Tables	9	1
Optimization	prepared statement & stored procedure	none
Strategy for Checkpointing	CopyUpdated	CopyAll

Table 6.2: Comparison of Two Testbeds

want to use the basic operations for the later comparison. For the data checkpointing, we have adopted another strategy called *CopyAll*. That means, the current checkpoint will completely substitute the stale one in the column family. This strategy leads to a number of repeated writes, if the change between two checkpoints is small. However, it is ideal for Cassandra because in this way there is only write operations for checkpointing without delete and query operations.

A comparison of two testbeds shows in Table 6.2.

6.3.2.3 Related Work

We have proposed to use different strategies (*CopyUpdated* and *CopyAll*) in our testbeds for data checkpointing. The idea of these strategies comes from others' research. In [VCS+09], authors have evaluated the overhead, checkpoint, and recovery times of several consistent checkpointing algorithms. They have proposed two fast checkpoint recovery algorithms for MMOGs in another work [CVS+11]. In our project, we focus on how to flush structured checkpoints to Cassandra as well as MySQL Cluster, and fetch them efficiently, which could be considered as an extension of their research.

6.3.3 Experimental Setup

The experiment is carried out in a multi-process operating environment, which simulate a real application scenario. We have executed 10 processes in parallel to access the database. The average time of running 10000 operations in one process will be returned as an experimental result.

Cassandra is eventually consistent. In order to guarantee the read-your-writes consistency, we specify the write consistency level to *ONE* and the read consistency level to *ALL*. The disk-resident database in MMORPGs is write-intensive. We must try to shorten the processing time of write operations. Hence, the consistency level of write is lower than read.

```
//Creation of a user-defined type named event
CREATE TYPE PlaneShift_db.event (
  col1 text,
  col2 text
);

CREATE TYPE ...

...

//Creation of a column family named characters
CREATE TABLE PlaneShift_db.characters (
  id uuid PRIMARY KEY,
  account_id int,
  loc_sector_id int,
  racegender_id int,
  name text,
  col1 text,
  ...
  col56 text,
  character_events map<int, frozen <event>>,  // a collection map
  character_spells map<int, frozen <spell>>,
  character_relationships map<int, frozen <relationship>>,
  character_quests map<int, frozen <quest>>,
  item_instances map<int, frozen <instance>>,
  character_skills map<int, frozen <cskill>>,
  trainer_skills map<int, frozen <tskill>>,
  character_traits set<int>     // a collection set
);
```

Listing 6.1: Creation of Characters Column Family

We have carried out two groups of experiments under different experimental environments.

Experimental environment I (no character) : we have simulated the scenario that an online game is just released, so more and more players start to join the game. That means, at the beginning there is no record in the database. We will evaluate the system performance of adding new characters' data into the database, as well as checkpointing and querying data in this case.

Experimental environment II (one million characters) : an online game has already robustly run for a long time, which has accumulated a large number of players. To simulate this scenario, we have previously inserted one million characters' information in the database. That means, in the *characters table* of both testbed databases, there are already one million rows. Additionally, we restrict that each character has at most 20 records in each bridge table (in testbed-

MySQL) or collection type column (in testbed-Cassandra). During the checkpointing we generate a random number between zero and twenty for each bridge table or collection type column as the new number of records storing in it. For this reason, every bridge table in the Testbed-MySQL has around ten million rows.

6.3.4 Experiments

We will evaluate each testbed separately, and then compare their results at last.

6.3.4.1 Experimental Results from Testbed-MySQL

The experimental results shown in Figure 6.13 make it easy for us to reach the following conclusions:

1) The best performance of inserting state data is gotten from using prepared statements. For update and read operations in both experiential environments, the use of stored procedures helps more. We will then use the best result for each operation in this testbed to compare with that from testbed-Cassandra.

2) The experiential results have proven that both prepared statement and stored procedure can help for enhancing the system performance.

3) Stored procedure is more suitable for our application scenario. Some complex database operations, like update, insert, query and delete data from multiple tables can be packaged together as a stored procedure, and manage as a transaction in the database.

4) The read performance of MySQL Cluster is high. The reason is that MySQL Cluster partitions and distributes data by hashing on keys. It uses then a hash index to query data, rather than scanning all rows.

5) The performance of update is higher than insert in our experiment. That is because by using the *CopyUpdated* strategy less data have been modified during the updating.

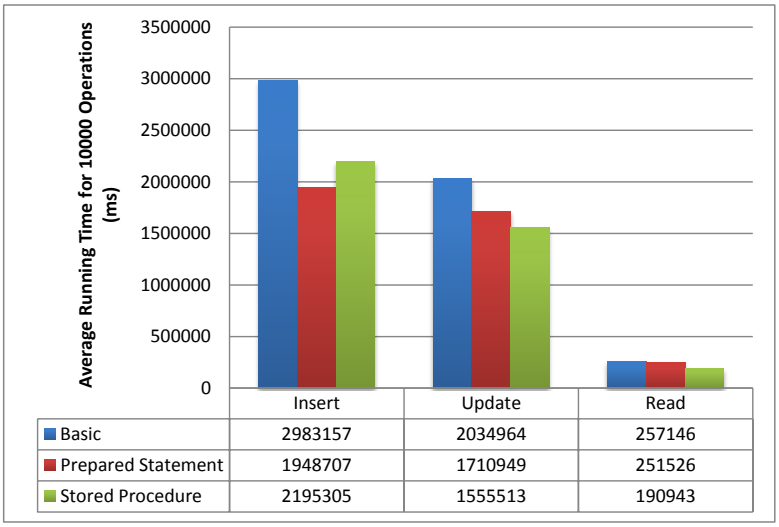
6) The data volume in the database does not affect on the results significantly. The reason could be that in this experiment, it is not a challenge to use a five-node MySQL Cluster to manage records of one million characters.

6.3.4.2 Experimental Results from Testbed-Cassandra

Figure 6.14 shows the experimental results of testbed-Cassandra. Accordingly, we get the following conclusions:

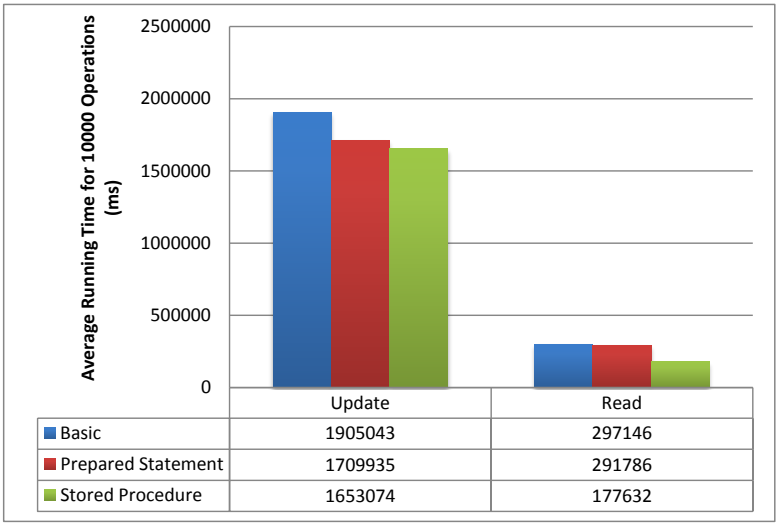
1) In contrast with the results from testbed-MySQL, the write performance of Cassandra is higher than read. The reason has been discussed in Section 3.4. Especially in this experiment, the read consistency level is *ALL*, which is higher than the write consistency level (*ONE*).

2) The data volume does not affect the experimental results. A five-node Cassandra cluster can also deal with the records of one million characters.



	Insert	Update	Read
Basic	2983157	2034964	257146
Prepared Statement	1948707	1710949	251526
Stored Procedure	2195305	1555513	190943

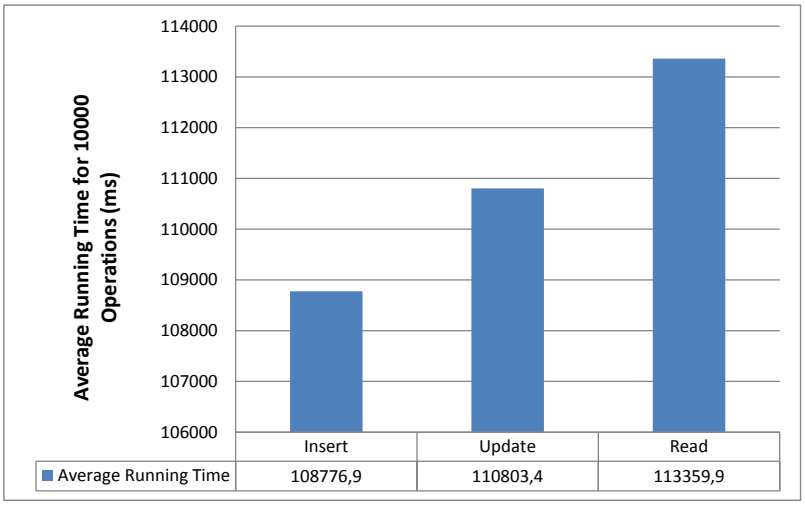
(a) Comparison of the Performance of Insert, Update and Read Using Three Methods in the Experimental Environment I (No Character)



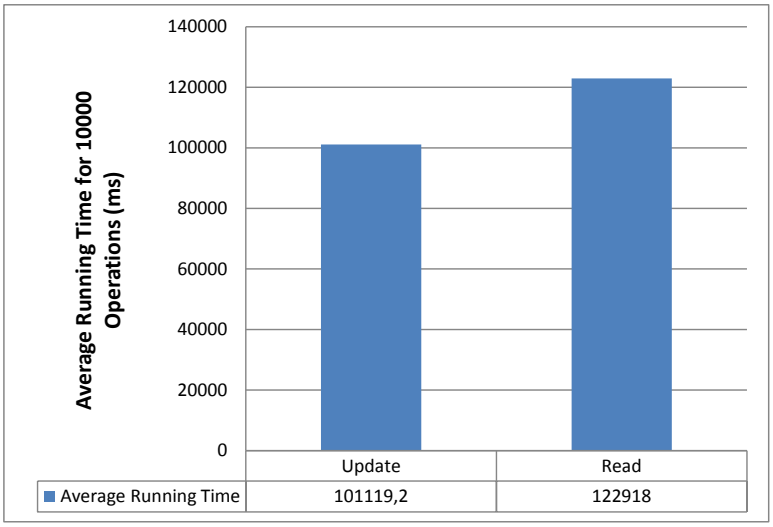
	Update	Read
Basic	1905043	297146
Prepared Statement	1709935	291786
Stored Procedure	1653074	177632

(b) Comparison of the Performance of Update and Read Using Three Methods in the Experimental Environment II (One Million Characters)

Figure 6.13: Comparison of the Performance (Average Running Time for 10000 Operations) of Different Operations of Testbed-MySQL Using Three Methods in Two Experimental Environments



(a) The Performance of Insert, Update and Read in the Experimental Environment I (No Character)



(b) The Performance of Update and Read in the Experimental Environment II (One Million Characters)

Figure 6.14: Performance (Average Running Time for 10000 Operations) of Different Operations of Testbed-Cassandra in Two Experimental Environments

6.3.4.3 Comparison of Experimental Results from Two Testbeds

We have integrated the best results from two testbeds into two diagrams (see Figure 6.15 and Figure 6.16). The comparative results are obvious and convincing. In both two experimental environments, use of Cassandra can significantly improve the efficiency of data processing, even though we have not used any optimization technique. For example, in Figure 6.15 the running time for inserting, updating and reading state data has been reduced respectively by 94.4%, 92.9% and 40.6% through using Cassandra. Besides the different ways of data processing between RDBMS and NoSQL DBMS, there are following specific reasons:

Database Schema : in testbed-MySQL, the information of one character are assigned to nine tables. Moreover, each bridge table holds a maximum of 20 records of a character. Thus, each data operation could involve at most 161 rows, which are distributed in different tables and even nodes in the cluster. The communication and I/O costs are consequently high. In contrast, in testbed-Cassandra, the entire information of one character is kept together in one row. Therefore, the overhead for data access is minimized.

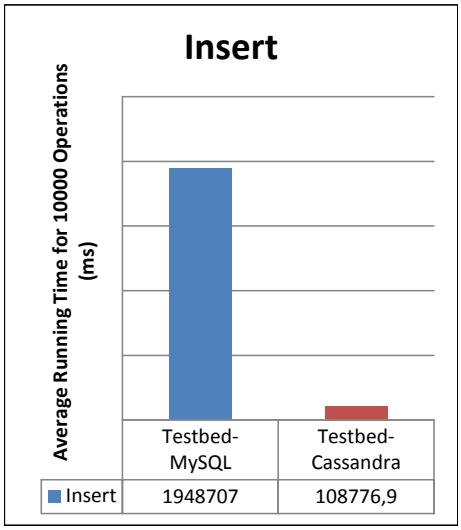
Consistency mechanism : the MySQL Cluster used in testbed-MySQL hosts two replicas for all data. By using the default two-phase commit mechanism, data are synchronized to both replicas during executing a write operation (see Section 2.2). Furthermore, distributed locks are required here to guarantee ACID transactions. However, in testbed-Cassandra, as long as one replica has been updated successfully, the write operation is considered to be complete.

Additionally, we can also reach some other conclusions:

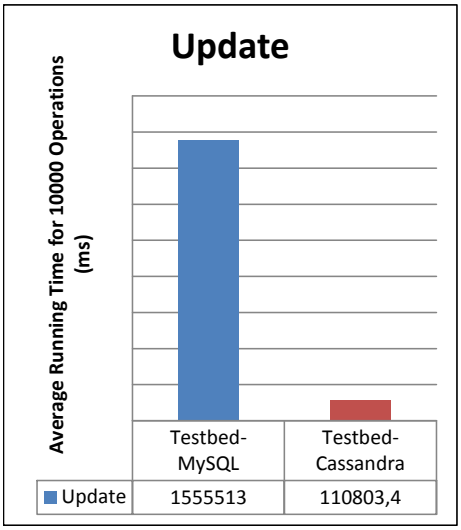
1) The read performance has not been enhanced as much as the write performance. There are two reasons: MySQL Cluster caches all data in memory. Cassandra only caches the latest or the most frequently accessed data. Therefore, disk I/O is inevitable; the read consistency level in Cassandra is specified to *ALL*. In contrast, MySQL Cluster fetched data only from one replica. In the next section, we will focus on improving the read performance of Cassandra.

2) The running time of checkpointing has been reduced obviously by using Cassandra. The benefit is that the frequency of checkpointing could be increased, which reduces the loss caused by a game server failure. Or we can checkpoint more characters' state data at the original frequency.

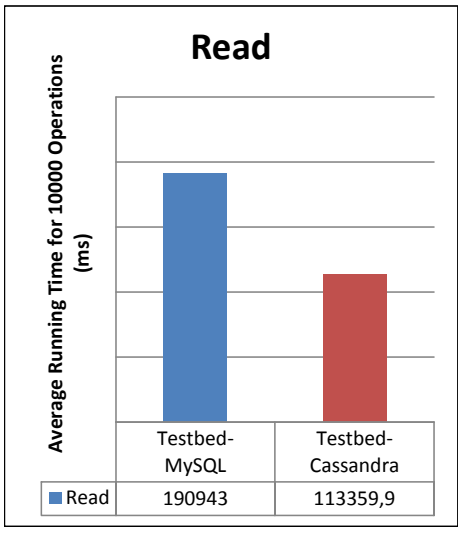
Overall, these experimental results have shown that the use of an RDBMS for data checkpointing and recovery in our game scenario is not efficient. The performance could be easily improved by applying Cassandra instead.



(a) Comparison of Insert Performance

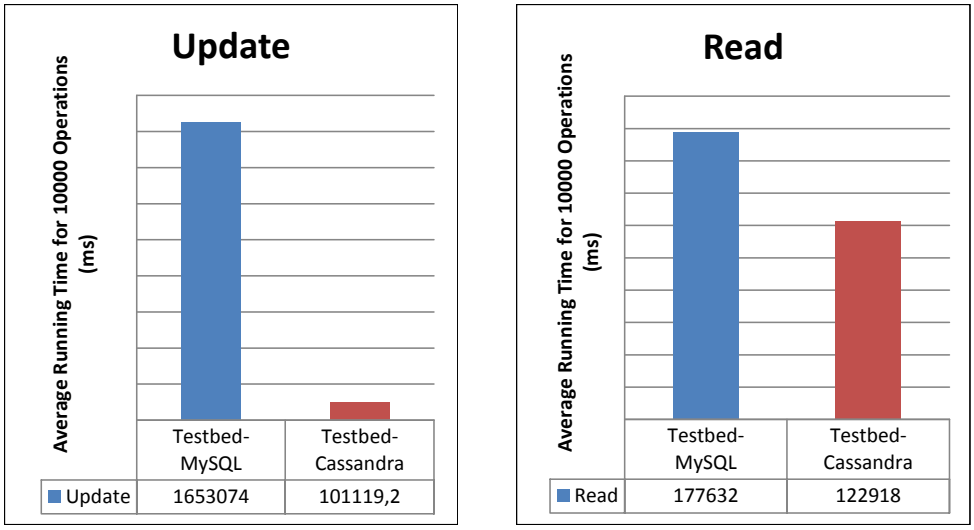


(b) Comparison of Update Performance



(c) Comparison of Read Performance

Figure 6.15: Comparison of the Performance (Average Running Time for 10000 Operations) of Two Testbeds in the Experimental Environment I (No Character)



(a) Comparison of Update Performance (b) Comparison of Read Performance

Figure 6.16: Comparison of the Performance (Average Running Time for 10000 Operations) of Two Testbeds in the Experimental Environment II (One Million Characters)

6.4 Experimental Proof of the Timestamp-based Model

In Chapter 5, we have pointed out Cassandra' problem in guaranteeing read-your-writes Consistency. Moreover, in Section 6.3, experimental results have shown that using Cassandra cannot enhance the read performance as much as write performance. Therefore, we have proposed the TSModel to address this problem. In this section, we want to verify it by experiments.

6.4.1 Implementation of the Testbed

In this experiment, we need to implement a data access server (DAS) and deploy a Cassandra cluster to cooperate with it.

6.4.1.1 Implementation of the Data Access Server

Based on the experimental requirements, we have implemented the following functionality for the application on the DAS:

Communication with Cassandra and TST : the application on the DAS is a bridge between the game server and the Cassandra cluster. State data are exchanged here for the data checkpointing and recovery purpose. Hence, it holds the connection with Cassandra; furthermore, the timestamp of each operation needs to be stored on the DAS. Accordingly, we have embedded H2 (a lightweight in-memory RDBMS) in the application to host the TST.

Tunable configuration : we will carry out a series of experiments to verify our Timestamp-based Model. The configuration of Cassandra Cluster/application for each experiment is different. For this reason, configurations like operation type (read/write/delete), consistency level, load balancing policy and data range should be tunable in the application.

Support for Node-aware Policy : in Chapter 5, we have proposed a new load balancing policy to communicate with Cassandra by using Java Driver. Hence, we have added a new class, *NodeAwarePolicy*, in Java Driver, which implements the *ChainableLoadBalancingPolicy* class. The process of this class is showed in Algorithm 3. Furthermore, we have overwritten the *getReplicas()* function in the *Metadata* class to make it possible to get the information about the *replication factor* as well as the replica placement strategy of the ring. In this way, if a user applies the *NodeAwarePolicy*, the new function can return all replica nodes for a data object. If the *TokenAwarePolicy* is used, the original *getReplicas()* function will be called, which only returns the "primary" replica node calculated by the token.

Powerful control panel : in order to facilitate setting of experimental parameters manually, we have designed a flexible and powerful control panel (see Figure 6.17). For example, we can use dialog I (see Figure 6.17a) to create/drop a keyspace/column family, and use dialog II (see Figure 6.17b) to execute different operations. Moreover, this application provides an interface to accept command line arguments from another application to set up all these parameters.

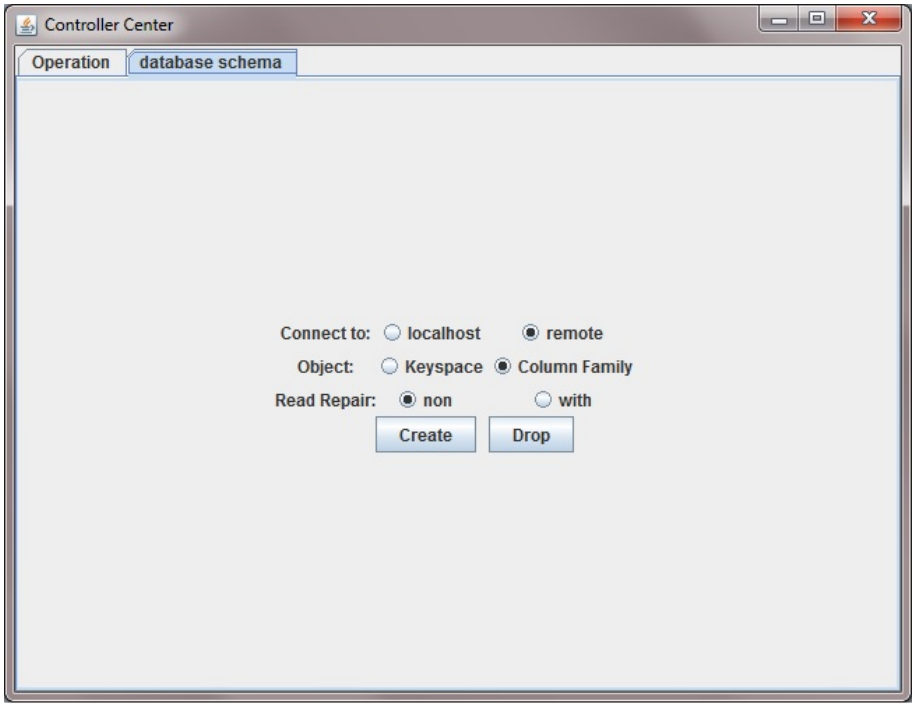
Support for multi-processes : the experiment is carried out under a multi-processes environment, so we have also designed a master application to call the application introduced above. This master application can call any number of slave applications, and pass different commands (parameters) to each of them. Furthermore, we also use the master application to manage the Cassandra cluster (start/stop one or several specified nodes).

6.4.1.2 Database Schema

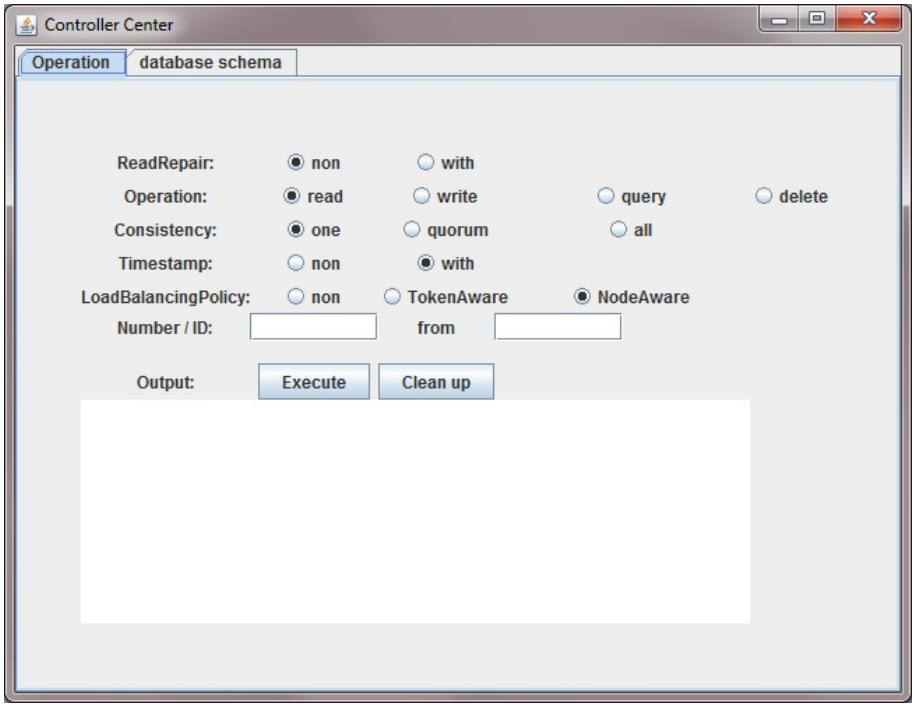
When we implemented this testbed, the collection type was not supported by CQL. For this reason, the structure of the column family is like Figure 5.8 with up to 160 columns, and the timestamp is stored as in Figure 5.6b. The column name is a string type, and the column value is an integer type.

6.4.2 Experimental Setup

Table 6.3 shows the setup of the testbed. We have deployed a five-node Cassandra cluster. Although the number of nodes is less than that in a practical game database, it is enough for our evaluations because we can already perform tests with different consistency levels, simulate a node failure, and get inconsistent data (see Table 5.1);



(a) Dialog I of the Control Panel



(b) Dialog II of the Control Panel

Figure 6.17: Control Panel

```
Input: an avatar/game object's UUID and the operation type (read, write, or
       delete)
Output: the coordinator for this operation
begin
    //***step I***//
    if is a read operation then
        get the host address (IP) from H2 based on the UUID
        while not yet checked all replica nodes in Cassandra do
            if a replica node with that IP is found && is up then
                return(use this replica node as the coordinator)
            end
        end
    end

    //***step II***//
    //is not a read operation//
    //or did not find any alive replica node with that IP//
    while not yet checked all replica nodes in Cassandra do
        if a replica node is up then
            return(use this replica node as the coordinator)
        end
    end

    //***step III***//
    //all replica nodes are down//
    while not yet checked all other nodes in Cassandra do
        if a node is up then
            return(use this node as the coordinator)
        end
    end
end
```

Algorithm 3: Process of the *NodeAwarePolicy* Class

30 million rows have been previously inserted into Cassandra cluster and the TST to simulate a real number of registered players in an MMORPG; in practice, there could be multiple TSTs working in parallel. We have only used one in order to evaluate its performance under a heavy workload; Each row in Cassandra contains a flexible number of columns from 110 to 160, which simulates the different number of properties that an object has; Cassandra is mainly used in this scenario to back up data, so writes are significantly more than reads. During the experiment, we have executed 10 processes in parallel to access Cassandra, with nine for writing and one for reading, which is used to simulate a real game environment; we use the average running time for executing

Cassandra Version	1.2.13
Number of Nodes	5
Replication Factor	3
Number of Rows	30 million
Number of Columns	$110 \sim 160$
Data Size	$\geqslant$ 120 GB
Client Driver	DataStax Java Driver 1.0
Writes : Reads	9 : 1

Table 6.3: Experimental Setup

10000 operations under this experimental environment as the criterion to evaluate the system performance.

6.4.3 Experiments

We will evaluate the efficiency of our timestamp-based model (TSModel), and compare it with some built-in methods in various environments (e.g., all nodes are up, or some nodes are down). Furthermore, there are two factors (accessing of the TST and data size in the cluster) that affect the efficiency of the new model. We will also assess their impact through experiments.

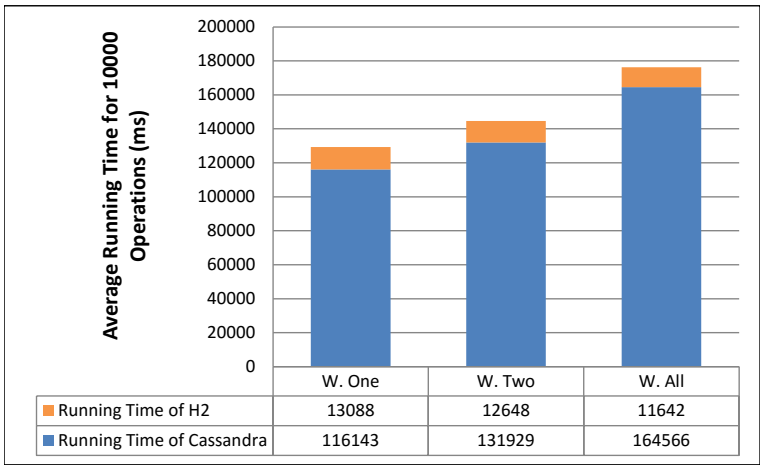
6.4.3.1 Effect of Accessing the Timestamp Table (TST) in H2

We have introduced a TST in the new game architecture, which certainly enhances the running time for data processing. However, the TST has a simple structure (only four columns), and is held in memory. Compared to the data accessing in a distributed disk resident database with data replication, its effect is negligible.

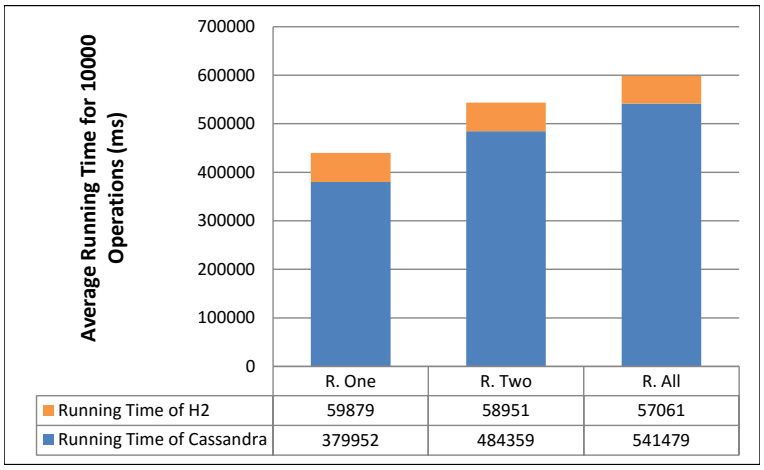
Figure 6.18 presents the running time of writes and reads with different CLs. The running time of accessing the TST only occupies about 9% in writes and about 12% in reads. Moreover, even if the running time of H2 is calculated, the total running time of querying with a low CL is still shorter than the running time of querying Cassandra with a high CL. For instance, the total time of write *ONE* is 129231 ms, which is still shorter than performing write *TWO* in Cassandra (131929 ms).

6.4.3.2 Write/Read Performance Using the Timestamp-based Model

We have proposed to use *TSModel* to guarantee the game consistency, and also proposed to integrate *TSModel* with the *NodeAwarePolicy* strategy (*N_TSModel*) to improve the system performance. In this experiment, we will evaluate the write/read performance of *TSModel*, *N_TSModel* and *T_TSModel* (*TSModel* integrates with the build-in strategy *TokenAwarePolicy* in Java Driver), and compare them with basic operations in Cassandra. The write/read CL of the first three methods is set to *ONE*, and the running time of H2 is calculated in the total running time. The write/read CL of basic operations is set to *ONE*, *TWO*, or *ALL*, and the total running time only includes the running time of Cassandra. During the experiment, all five nodes are available. The results are showed in Figure 6.19. We can reach conclusions from the figures that:



(a) Effect on the Performance (Average Running Time for 10000 Operations) of Basic Write Operations with Different Consistency Levels

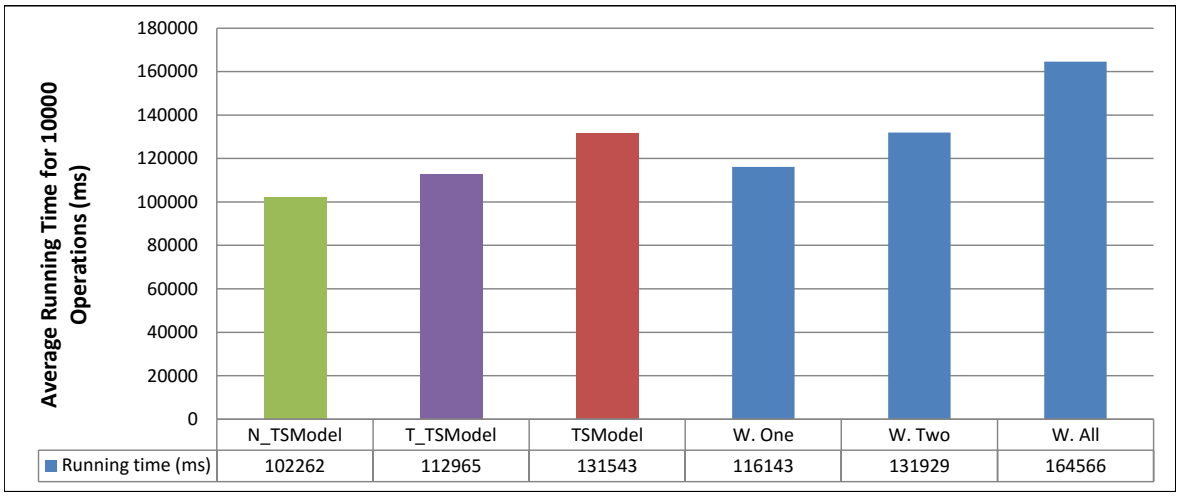


(b) Effect on the Performance (Average Running Time for 10000 Operations) of Basic Read Operations with Different Consistency Levels

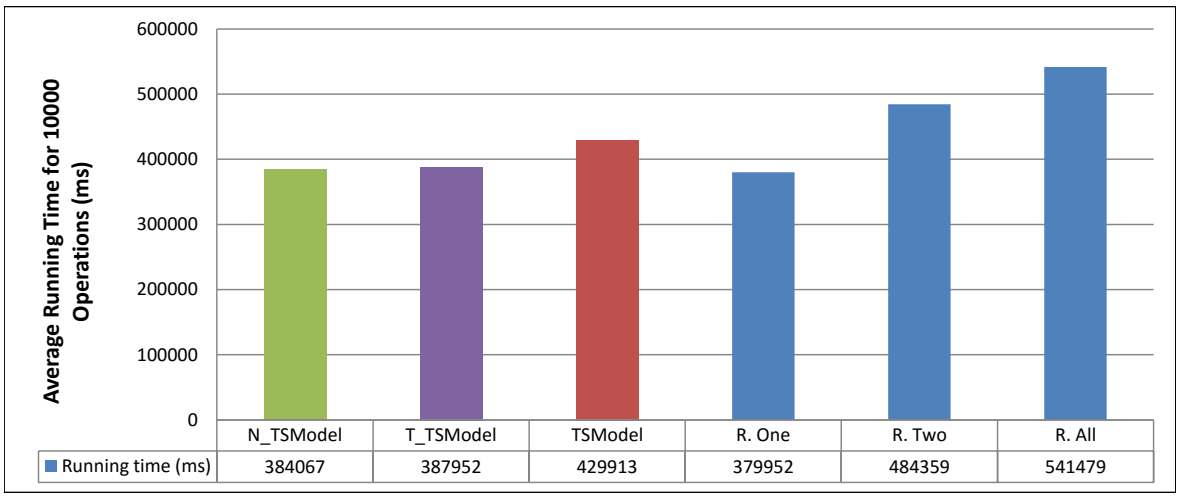
Figure 6.18: Effect of Accessing the TST in H2

1) the query performance using *TSModel* is between write/read *ONE* and *TWO*. That means when all nodes in the cluster are available, the write/read performance to guarantee the read-your-writes consistency is now close to CL *ONE*, which is efficient.

2) the query performance of both *N_TSModel* and *T_TSModel* is closer to or even better than write/read *ONE*. Actually, through applying the *TokenAwarePolicy* and *NodeAwarePolicy* strategy, the write/read performance of Cassandra has been increased. The reason is that there is less communication among nodes in the cluster (coordinator is one of the replica nodes).



(a) Comparison of Write Performance (Average Running Time for 10000 Operations)



(b) Comparison of Read Performance (Average Running Time for 10000 Operations)

Figure 6.19: Performance Comparison of *TSModel* and its Derivative (*N_TSModel* and *T_TSModel*) with Basic Operations (Write/Read with Different Consistency Levels) in Cassandra

3) the performance of *N_TSModel* is better than *T_TSModel*, especially the write performance. That is because the most efficient replica node has been chosen as the coordinator, so the running time is short. And the workload of each node in the cluster is more balanced.

6.4.3.3 Read Performance under a Node Failure Environment

Similar to the test four presented in Table 5.1, we have repeated the above experiment under a temporary node failure environment (Two nodes are failed during writing, all nodes are available during reading. The write Cl is set to *ONE*).

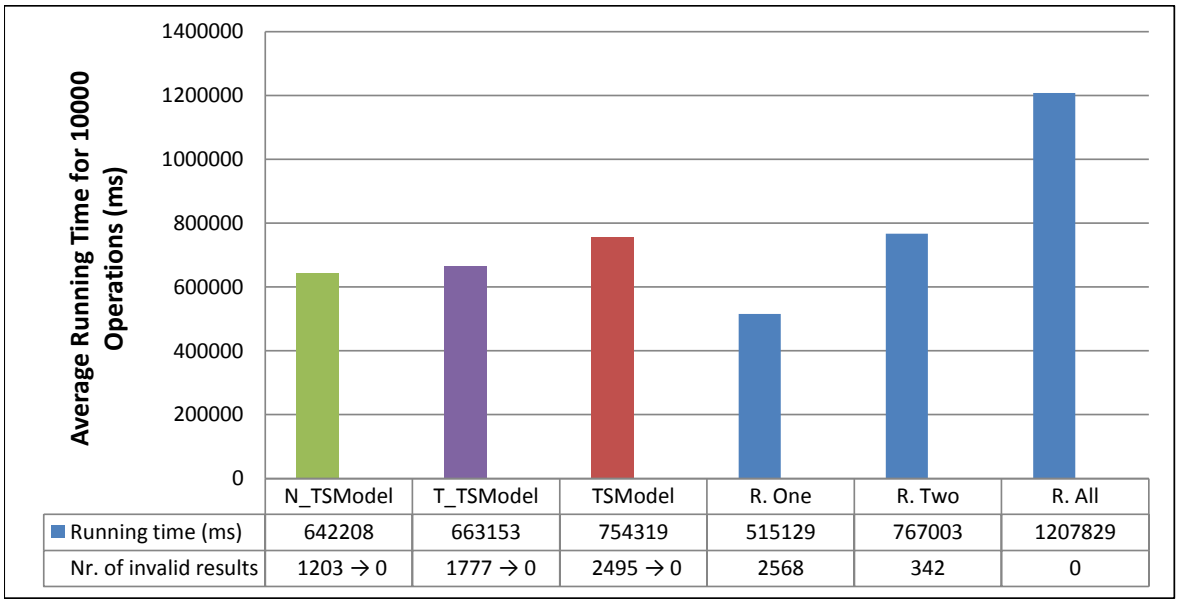


Figure 6.20: Comparison of Read Performance under Node Failure

Figure 6.20 leads us to the conclusion that:

1) only read *ALL* could ensure to fetch the up-to-date data by just retrieving once. However, its performance is the worst.

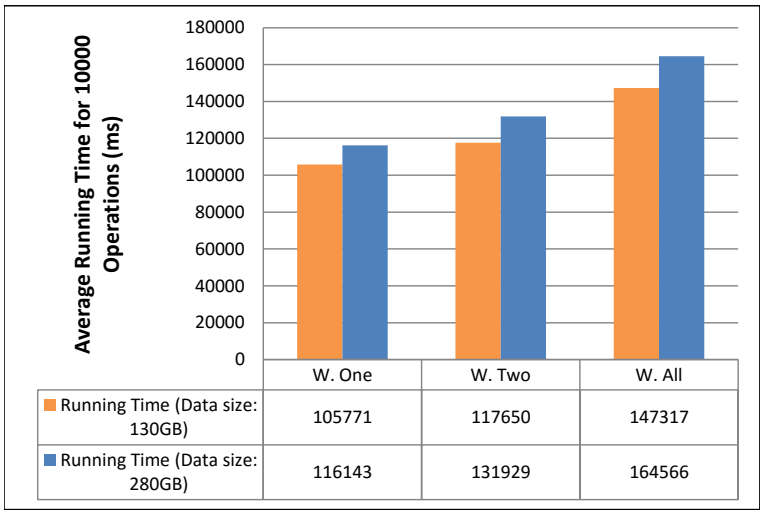
2) although the running time of read *ONE* and read *TWO* is relatively shorter, both of them cannot guarantee data currency.

3) by using the timestamp-based model (*TSModel*, *N_TSModel* and *T_TSModel*), the total time is compressed between read *ONE* and read *TWO*, whereas all up-to-date data are fetched eventually.

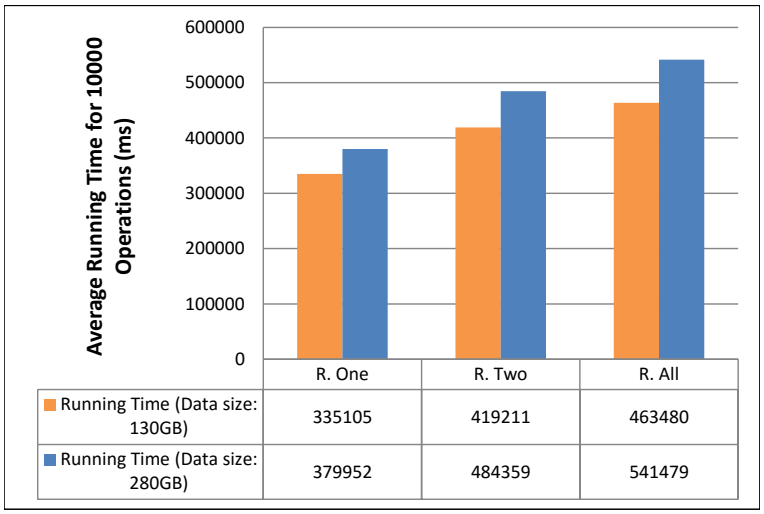
4) in theory, by using the *NodeAwarePolicy*, no invalid data (null value) should be found since the coordinator already holds the update-to-date data. However, in practice, the invalid data are still returned. Through tracing the query, we found that the coordinator does not always execute the request locally. Sometimes it forwards the request to another replica node, which might be just recovered from a node failure. Consequently it does not hold the up-to-date data. The reason could be that the workload of this coordinator is too heavy, so the other replica node can process the request faster. However, we can still state that the invalid data are halved (from 24995 to 1203), and consequently, the read performance is much closer to read *ONE*.

6.4.3.4 Effect of Data Size

As discussed in Section 5.2.2, the data size of the cluster will increase obviously caused by the stale data. Figure 6.21 describes the system performance under different data



(a) Effect on Write Performance (Average Running Time for 10000 Operations)

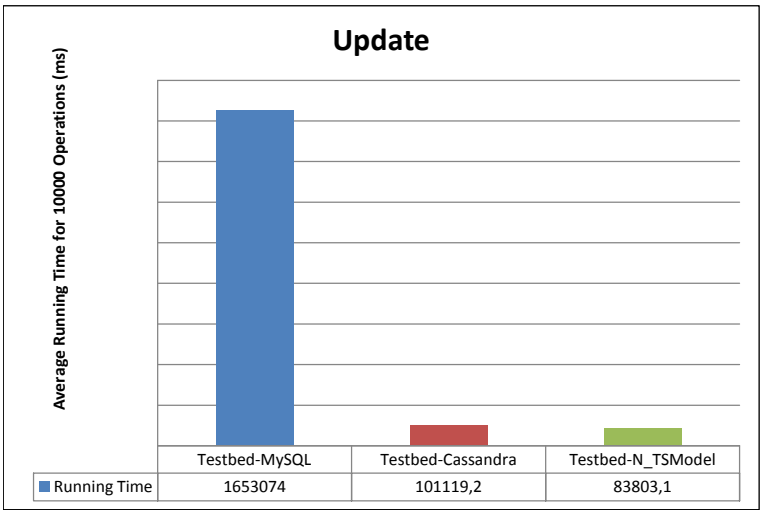


(b) Effect on Read performance (Average Running Time for 10000 Operations)

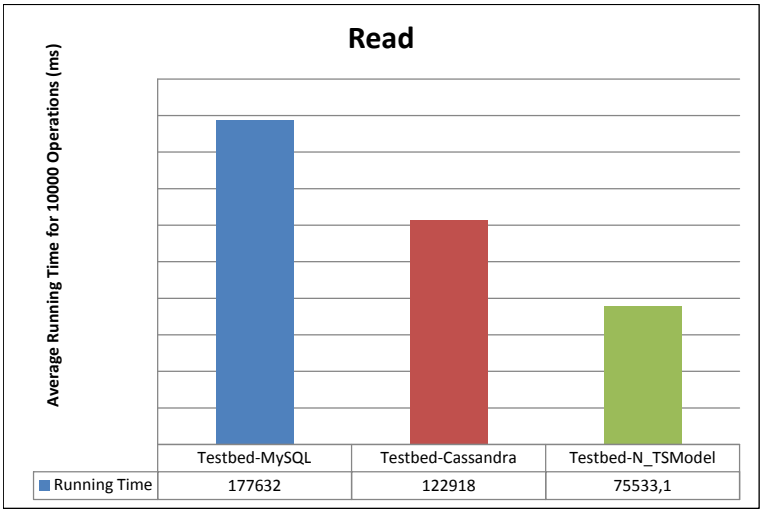
Figure 6.21: Effect of Data Size in Cassandra on System Performance

size (130 GB and 280 GB). Both write and read performance have been affected. The time is wasted by retrieving a large number of files (SSTables) in disk. Therefore, we conclude that it is imperative to clean up all stale data timely.

The strategy of deleting stale data could be classified into two groups, namely, eager deletion and lazy deletion. Eager deletion refers to deleting the stale data instantly after flushing a new checkpoint; Lazy deletion describes that stale data will be deleted together asynchronously during a garbage collection at a specific time or under a certain condition (e.g., when the cluster is idle). Cassandra does not yet support a range query on the second compound primary key (TS in our experiment). That means, the stale



(a) Comparison of Update Performance (Average Running Time for 10000 Operations)



(b) Comparison of Read Performance (Average Running Time for 10000 Operations)

Figure 6.22: Performance Comparison of Testbed-N_TSModel with Testbed-MySQL and Testbed-Cassandra (Two Replicas in Each Database)

data could only be deleted one by one, thereby spending the same time, whichever strategy is chosen. Lazy deletion prevents bringing extra workload during peak hours. However, we have to record the timestamp of each checkpoint on the server side, or get it by executing an expensive read *ALL* in the cluster and use it to detect stale data. Overall, we need to choose the strategy based on the actual scenario.

6.4.3.5 Performance Comparison with Testbed-MySQL and Testbed-Cassandra

We have rebuilt the testbed-Cassandra introduced in Section 6.3 to make it support the *N_TSModel*, and named it as testbed-N_TSModel. And then, we have compared its

update and read performance with that of the other two testbeds when guaranteeing the Read-Your-Writes consistency. The experimental setup is the same with the experimental environment II (one million characters) (see Section 6.3.3). There are two replicas for each row in all three testbeds. The write consistency level is *ONE* and read consistency level is *ALL* (*TWO*) in testbed-Cassandra. In testbed-N_TSModel, if there is no node failure, both write and read consistency level is *ONE*. Experimental results are showed in Figure 6.22.

We can reach the conclusion that by using the *N_TSModel*, both read and update performance have been enhanced. The average time for processing 10000 updates and reads is reduced by 94.9% and 57.5% comparing with the testbed-MySQL, which is satisfactory.

6.5 Summary

In this chapter, we have implemented a number of testbeds for different experimental purposes. The experimental results have proven that using NoSQL DBMS (e.g., Cassandra) for checkpointing and recovering state data is reasonable. It brings a potential scalability and high performance for persisting data of MMORPGs.

7. Conclusion

Within the CloudCraft project, we proposed a Cloud-based architecture for MMORPGs. The introduction of a NoSQL DBMS in the game scenario is aiming at addressing the issue caused by using RDBMSs for managing large amounts of game data. For instance, RDBMSs are hard to scale out, and are not always available in a distributed environment. In order to choose a suitable NoSQL DBMS as a substitute, we analyzed the existing architectures of MMORPGs and the management requirements of different data sets in them. We found that the state data and log data must be managed in the database, which pose a challenge to conventional RDBMSs. Furthermore, there are also some requirements that cannot be fulfilled by a NoSQL DBMS, like the real-time processing and complex query. For this reason, we only proposed to use NoSQL DBMSs to persist data checkpoints, which are applied to recover data to an in-memory database in game servers. For this use case, the chosen NoSQL DBMS must be write-intensive. Finally, Cassandra entered our field of vision. We demonstrated then the feasibility of using it in theory.

To provide an experimental proof for the potential scalability of Cassandra in the game scenario, we created a prototype based on an open source project, JMMORPG. We redesigned the database schema of it to meet the features of Cassandra's database model. The experimental results showed that under a certain workload, the system performance could be improved by adding more nodes in Cassandra cluster, and the write/read performance is satisfactory.

We also compared the checkpoint and recovery times of MySQL Cluster and Cassandra when processing the same amount of state data. We used different methods/strategies to optimize the performance of MySQL Cluster, but only executed the basic operations in Cassandra. The experimental results confirmed the correctness of the choice of Cassandra. The running time for executing inserts, updates and reads has been reduced by 94.4%, 92.9% and 40.6%. From the result we also found that Cassandra cannot guarantee data consistency efficiently.

In a distributed system (like Cassandra cluster), the implied trade-off between consistency and run-time performance and/or availability cannot be solved easily, as we demonstrated. For this purpose, we introduced several concepts to manage consistency in a multi-layered architecture. A key ingredient is a simple timestamp-based approach, which checks for inconsistencies on the fly and only mends these if they occur. As the check itself is quite light-weight and, as also shown, situations triggering the inconsistency are very unlikely in our scenario, the new approach provides excellent run-time performance compared to strongly consistent operations as provided by Cassandra itself. The average time for processing 10000 updates and reads is reduced by 94.9% and 57.5% respectively comparing with the testbed using MySQL Cluster.

Overall, by using the Cloud technology, we can solve the data persistence issues that MMORPGs are facing. The new Cloud-based architecture helps to improve the scalability, availability and performance of the game system significantly.

8. Future Work

We have carried out many experiments on the system scalability, performance and game consistency. However, limited by the infrastructure and time there are more experiments to be done, and some open issues are left to address. This research could continue from the following aspects:

1) All experiments in this work were carried out in an intranet. In practice, data of an MMORPG are distributed and replicated in multiple data centers to avoid getting data from remote geographic locations. However, data synchronization across multiple data centers would delay the running time of writes. The write performance of RDBMSs supporting strong consistency is even worse in this situation. By using a NoSQL DBMS like Cassandra, we can specify the write/read consistency to the *LOCAL* level (e.g., *LOCAL_ONE* or *LOCAL_QUORUM*), which could improve the write performance. That means, in theory the performance difference between using two kinds of DBMSs in a multi-data center environment should be more obvious. In the future, we would like to evaluate the performance of our prototype in this environment.

2) We proposed a node-aware policy to specify the coordinator for each query, which only works at the moment with the *SimpleStrategy*, that places the additional replicas on the next nodes of the first replica node clockwise in the ring. In a multi-data center environment, another replica placement strategy *NetworkTopologyStrategy* is recommended, which places replica across multiple data centers and on distinct racks. In this case, the method of getting information of all replica nodes using in our prototype does not work. We would modify our program to adapt to this environment in the future.

3) With the help of the node-aware policy, we can specify the replica node holding the current checkpoint as a coordinator. And we also have checked the source code

of Cassandra 2.1[1], which showed if a coordinator holds the data locally, it returns them directly without forwarding the request to other replica nodes. However, in practice sometimes the coordinator still gets data from other replica nodes, which could be stale. Currently, it is still an open issue for us. In the future, we would use the node-aware policy in the latest release of Cassandra (currently is version 3.6) to check whether it works or not. If not, we would check the code of Cassandra and trace a query request to find out the reason.

4) Limited by the infrastructure, we could only take at most five nodes in a Cassandra cluster, so we can only conclude that our Cloud-based prototype has potential scalability. The number of nodes in a practical game database is far more than that. Hence, we would increase the number of nodes in the future, and redo the experiment. Furthermore, we also would like to evaluate the scalability of RDBMS in this experimental environment. We hope the experimental result could prove that using a NoSQL DBMS is more suitable in the game scenario.

5) In this project, we have taken Cassandra as an example in the Cloud-based prototype, but there are many other NoSQL DBMSs could be used, like HBase, MongoDB and Riak. Each NoSQL DBMS has its own characteristics as well as issues when using in the game scenario. For example, if we use HBase here, the guarantee of read-your-writes consistency would not become a problem any more because it ensures strong consistency in the row level. However, the write performance especially under a node failure could be worse than Cassandra. In the future, we would like to compare Cassandra with other NoSQL DBMSs in the game scenario to find out the best alternative to RDBMSs in MMORPGs.

[1]https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/service/AbstractReadExecutor.java#L78 (accessed 26.11.2014)

A. Appendix

The database schema of PlaneShift is presented as follows[1]:

[1]Database Schema of PlaneShift: https://github.com/baoboa/planeshift/blob/master/src/server/database/planeshift_db_rev1256.png (accessed 20.12.2015).

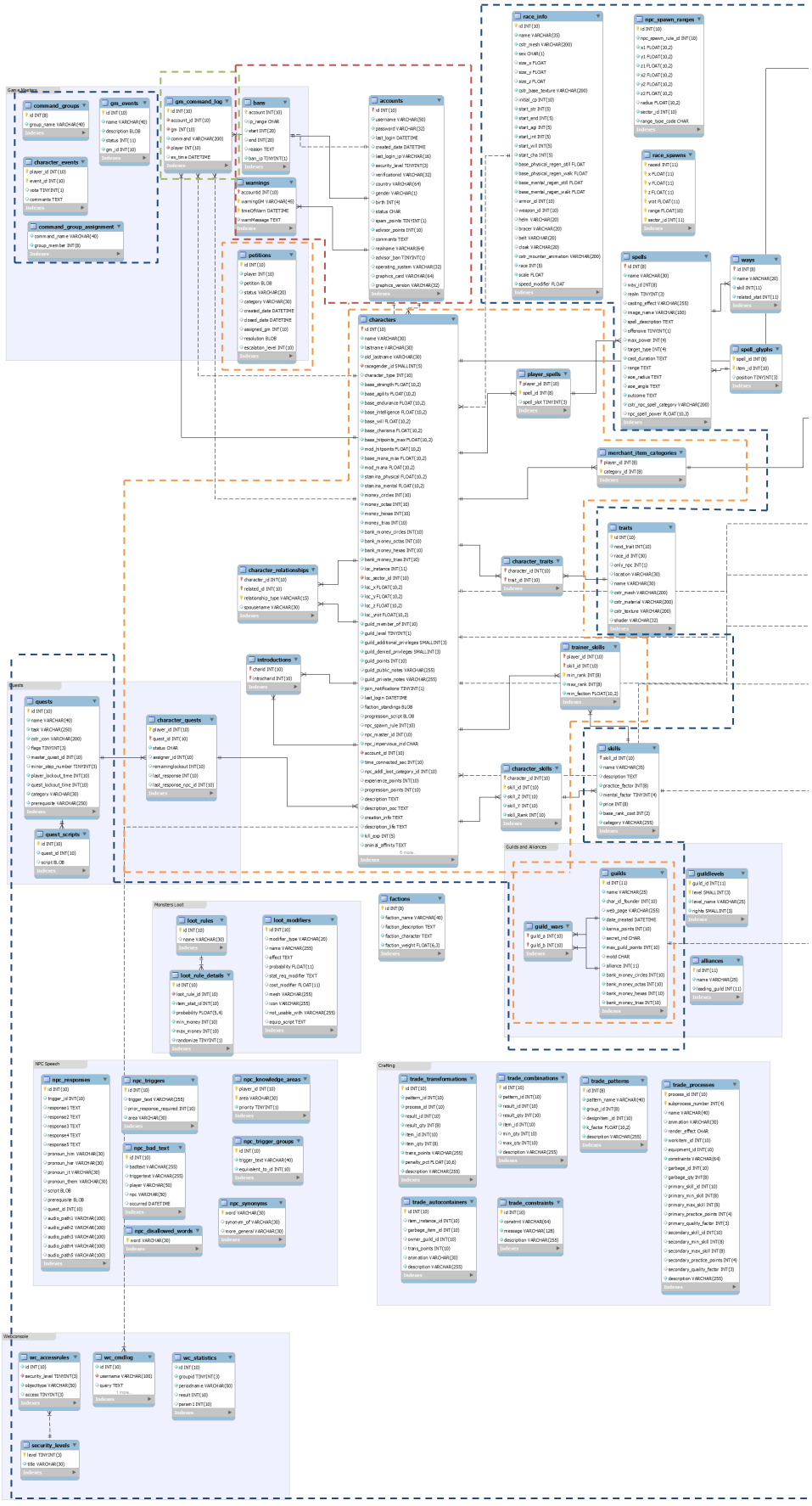
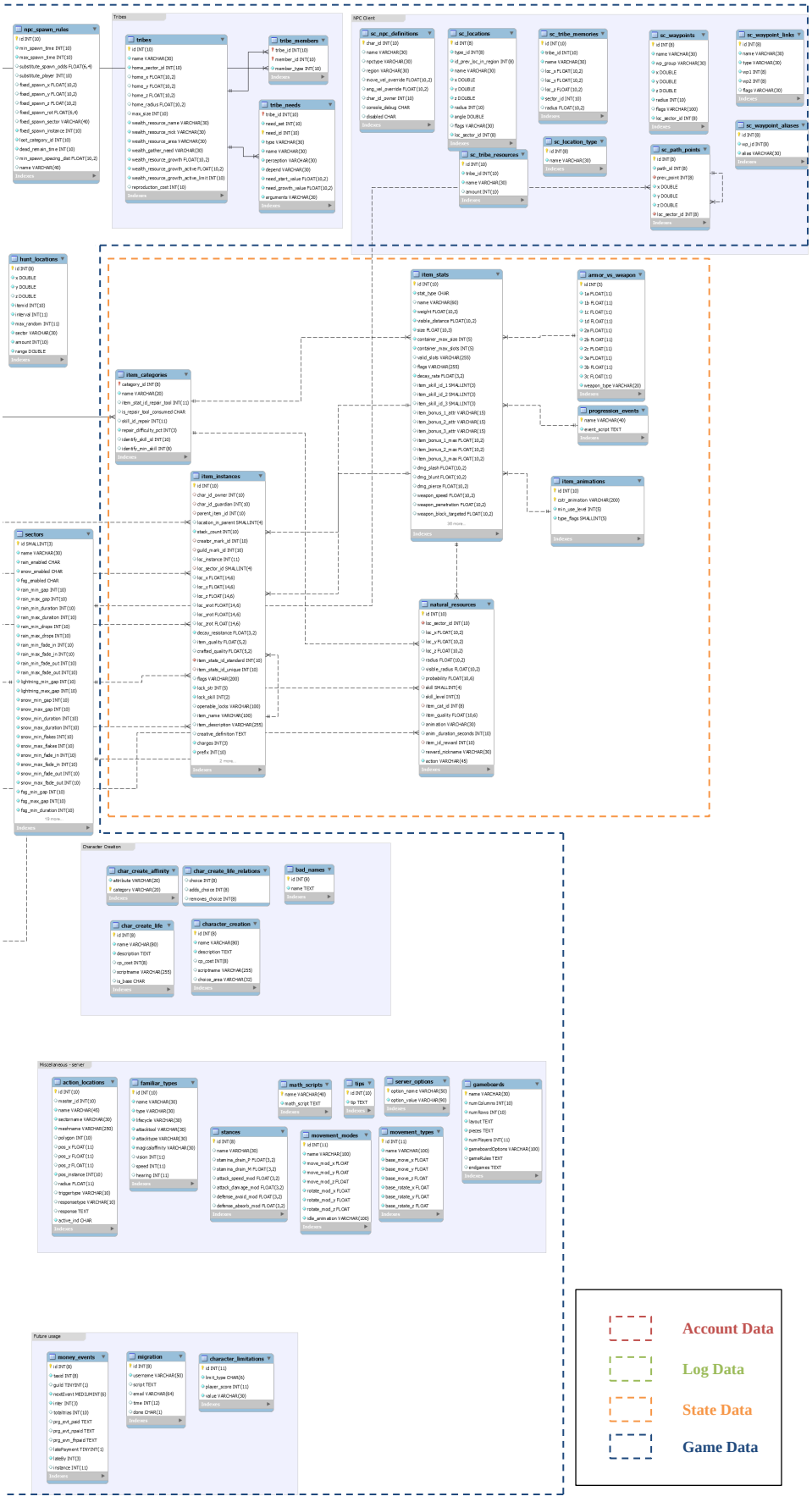


Figure A.1: Database schema of PlaneShift[1]


Account Data
Log Data
State Data
Game Data

Bibliography

[1UP08] 1UP. E3 2008: Sony Announces MAG: Massive Action Game. Website, July 2008. Available online at http://www.1up.com/news/e3-2008-sony-announces-mag; visited on July 23th, 2015. (cited on Page 7)

[Aba09] Daniel J. Abadi. Data Management in the Cloud Limitations and Opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009. (cited on Page 2)

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010. (cited on Page 21)

[Ale03] Thor Alexander. *Massively Multiplayer Game Development*. Thomson Learning, 1 edition, 2003. (cited on Page 10)

[APS08] Leigh Achterbosch, Robyn Pierce, and Gregory Simmons. Massively multiplayer online role-playing games: The past, present, and future. *Theoretical and Practical Computer Applications in Entertainmen*, 5(4):Article No. 9, 2008. (cited on Page 1 and 7)

[APV07] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Data Currency in Replicated DHTs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data (SIGMOD 2007)*, pages 211–222, June 2007. (cited on Page 60 and 67)

[AT06] Marios Assiotis and Velin Tzanov. A Distributed Architecture for MMORPG. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page Article No. 4, 2006. (cited on Page 10)

[BAT15] BATTLE.NET. Free Character Migration. Website, June 2015. Available online at https://us.battle.net/support/en/article/free-character-migration; visited on August 20th, 2015. (cited on Page 11)

[BBC+11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research(CIDR)*, pages 223–234, January 2011. (cited on Page 50)

[BFG+08] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 251–264, June 2008. (cited on Page 26)

[Bre12] Eric Brewer. CAP Twelve Years Later: How the "Rules" have Changed. *Computer*, 45(2):23–29, 2012. (cited on Page 26 and 27)

[Bur07] Brendan Burns. *Darkstar: The Java Game Server*. O'Reilly Media, 2007. (cited on Page 9 and 76)

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012. (cited on Page 58)

[BW09] Tim Blackman and Jim Waldo. Scalable Data Storage in Project Darkstar. Technical report, 2009. (cited on Page 76)

[Casa] Cassandra. Limitations. Website. Available online at http://wiki.apache.org/cassandra/CassandraLimitations; visited on January 24th, 2016. (cited on Page 32 and 54)

[Casb] Apache Cassandra. Cassandra Query Language (CQL) v3.3.1. Website. Available online at http://cassandra.apache.org/doc/cql3/CQL.html; visited on February 16th, 2016. (cited on Page 31, 41, and 43)

[Cat10] Rick Cattell. Scalable SQL and NoSQL Data Stores. *ACM Special Interest Group on Management of Data (SIGMOD 2010)*, 39(4):12–27, 2010. (cited on Page 2, 14, and 17)

[CBKN11] Navraj Chohan, Chris Bunch, Chandra Krintz, and Yoshihide Nomura. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing (CLOUD 2011)*, pages 692–699, July 2011. (cited on Page 66)

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS 2008)*, 26(2):4, 2008. (cited on Page 30)

[CHHL06] Kuan-ta Chen, Polly Huang, Chun-ying Huang, and Chin-laung Lei. Game Traffic Analysis: An MMORPG Perspective. *Computer Networks*, 50(16):3002–3023, 2006. (cited on Page 15 and 81)

[Cod85a] Edgar Frank Codd. Does your DBMS Run by the Rules? *ComputerWorld*, 19:49–64, 1985. (cited on Page 17)

[Cod85b] Edgar Frank Codd. Is Your DBMS Really Relational? *ComputerWorld*, 19:ID/1–ID/9, 1985. (cited on Page 17)

[com10] EVE community. This Weekend: The Alliance Tournament Finals. Website, Juni 2010. Available online at http://community.eveonline.com/news/dev-blogs/this-weekend-the-alliance-tournament-finals/; visited on July 23th, 2015. (cited on Page 7)

[CSKW02] Sergio Caltagirone, Bryan Schlief, Matthew Keys, and Mary Jane Willshire. Architecture for a Massively Multiplayer Online Role Playing Game Engine. *Journal of Computing Sciences in Colleges*, 18(2):105–116, 2002. (cited on Page 9)

[CVS+11] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *Proceedings of the 2011 international conference on Management of data (SIGMOD 2011)*, pages 265–276, October 2011. (cited on Page 88)

[DAA10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, pages 163–174, June 2010. (cited on Page 55 and 66)

[Dat16] DataStax. Configuring Data Consistency. Website, February 2016. Available online at http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html; visited on February 14th, 2016. (cited on Page xv and 41)

[DBM07] DBMS2. The Database Technology of Guild Wars. Website, June 2007. Available online at http://www.dbms2.com/2007/06/09/the-database-technology-of-guild-wars/; visited on August 21th, 2015. (cited on Page 13)

[DBM09] DBMS2. MMO Games are Still Screwed Up in Their Database Technology. Website, June 2009. Available online at http://www.dbms2.com/2009/06/14/mmo-rpggames-database-technology/; visited on August 21th, 2015. (cited on Page 13)

[DEAA09] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. Elastras: An elastic transactional data store in the cloud. *HotCloud*, 9:131–142, 2009. (cited on Page 66)

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, and Peter Vosshall. Dynamo: Amazon' s Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 205–220, October 2007. (cited on Page 30)

[Dia13] Ziqiang Diao. Consistency Models for Cloud-based Online Games: the Storage System's Perspective. In *25th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken)*, pages 16–21. CEUR-WS, May 2013. (cited on Page 57 and 60)

[DS13] Ziqiang Diao and Eike Schallehn. Towards Cloud Data Management for MMORPGs. In *The 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*, pages 303–308. SciTePress, May 2013. (cited on Page 47)

[DSWM13] Ziqiang Diao, Eike Schallehn, Shuo Wang, and Siba Mohammad. Cloud Data Management for Online Games: Potentials and Open Issues. *Datenbank-Spektrum*, 13(3):179–188, 2013. (cited on Page 5, 21, 47, 51, 57, and 70)

[DWS14] Ziqiang Diao, Shuo Wang, and Eike Schallehn. Cloud-based Persistence Services for MMORPGs. In *The 11th International Baltic Conference on DB and IS (DB&IS2014)*, pages 303–314. IOS Press, June 2014. (cited on Page 57 and 73)

[DWSS14] Ziqiang Diao, Shuo Wang, Eike Schallehn, and Gunter Saake. CloudCraft: Cloud-based Data Management for MMORPGs. *Databases and Information Systems VIII*, 270(9):71–84, 2014. (cited on Page xiv, 57, 73, and 77)

[DZSM15] Ziqiang Diao, Pengfei Zhao, Eike Schallehn, and Siba Mohammad. Achieving Consistent Storage for Scalable MMORPG Environments. In *The 19th International Database Engineering & Applications Symposium (IDEAS 2015)*, pages 33–40. ACM Press, July 2015. (cited on Page 57, 60, and 73)

[FBS07] Wu-chang Feng, David Brandt, and Debanjan Saha. A long-term study of a popular mmorpg. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames 2007)*, pages 19–24, September 2007. (cited on Page 1)

[FDI+10] Thomas Fischer, Michael Daum, Florian Irmert, Christoph Neumann, and Richard Lenz. Exploitation of Event-semantics for Distributed Publish/Subscribe Systems in Massively Multiuser Virtual Environments. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium (IDEAS 2010)*, pages 90–97. ACM Press, August 2010. (cited on Page 2 and 55)

[FGC+97] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 78–91, October 1997. (cited on Page 26)

[FMC+11] Craig Franke, Samuel Morin, Artem Chebotko, John Abraham, and Pearl Brazier. Distributed Semantic Web Data Management in HBase and MySQL Cluster. In *IEEE International Conference on Cloud Computing (CLOUD 2011)*, pages 105–112. IEEE, July 2011. (cited on Page 17)

[GBS11] Francis Gropengießer, Stephan Baumann, and Kai-Uwe Sattler. Cloudy Transactions Cooperative XML Authoring on Amazon S3. In *Datenbanksysteme für Business, Technologie und Web (BTW 2011)*, pages 307–326, Febuary 2011. (cited on Page 60, 66, and 67)

[GDG08] Nitin Gupta, Alan Demers, and Johannes Gehrke. SEMMO : A Scalable Engine for Massively Multiplayer Online Games. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 1234–1238. ACM Press, June 2008. (cited on Page 15 and 54)

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM Special Interest Group on Algorithms and Computation Theory (SIGACT 2002)*, 33(2):51–59, June 2002. (cited on Page iii, v, 2, and 24)

[GS11] Francis Gropengießer and Kai-Uwe Sattler. Transactions a la carte-Implementation and Performance Evaluation of Transactional Support on Top of Amazon S3. In *25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011)*, pages 1082–1091, May 2011. (cited on Page 27 and 66)

[Hew10] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, 2010. (cited on Page 31)

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983. (cited on Page 18 and 26)

[Hva99] Svein-Olaf Hvasshovd. *Recovery in Parallel Database Systems*. Vieweg+Teubner, 1999. (cited on Page 10)

[HWW09] Shang Hwa Hsu, Ming-Hui Wen, and Muh-Cherng Wu. Exploring user experiences as predictors of mmorpg addiction. *Computers & Education*, 53(3):990–999, 2009. (cited on Page 1)

[IGN14] IGN. World of Warcraft Subscriptions Back Over 10 Million. Website, November 2014. Available online at http://www.ign.com/articles/2014/11/20/world-of-warcraft-subscriptions-back-over-10-million; visited on July 23th, 2015. (cited on Page 7)

[IGN15] IGN. Games at E3 2015. Website, July 2015. Available online at http://www.ign.com/wikis/e3/Games_at_E3_2015; visited on July 16th, 2015. (cited on Page 1)

[IHK04] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned Federation of Game Servers : a Peer-to-peer Approach to Scalable Multiplayer Online Games. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES 2004)*, pages 116–120. ACM Press, August 2004. (cited on Page 15)

[ILJ10] Alexandru Iosup, Adrian Lascateu, and Nicolae Japui. CAMEO: Enabling Social Networks for Massively Multiplayer Online Games through Continuous Analytics and Cloud Computing. In *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games (NetGames 2010)*, pages 1–6. IEEE, November 2010. (cited on Page 1 and 8)

[Kav14] Michael J. Kavis. *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014. (cited on Page 22)

[KHA09] T Kraska, M Hentschel, and G Alonso. Consistency Rationing in the Cloud: Pay Only When It Matters. *Proceedings of the VLDB Endowment (PVLDB 2009)*, 2(1):253–264, 2009. (cited on Page 47)

[KK10] Donald Kossmann and Tim Kraska. Data management in the cloud: Promises, state-of-the-art, and open questions. *Datenbank-Spektrum*, 10(3):121–129, 2010. (cited on Page 22 and 47)

[KKL10] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010)*, pages 579–590, June 2010. (cited on Page 22)

[KKR14] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking Scalability and Elasticity of Distributed Database Systems. *Proceedings of the VLDB Endowment*, 7(13):1219–1230, 2014. (cited on Page 30)

[KLXH04] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, volume 1. IEEE Press, March 2004. (cited on Page 10)

[KVK+09] Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, and Michael Hawker. Mammoth A Massively Multiplayer Game Research Framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games (FDG 2009)*, pages 308–315. ACM, April 2009. (cited on Page 2 and 54)

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. (cited on Page 27)

[LBC12] Huaiyu Liu, Mic Bowman, and Francis Chang. Survey of State Melding in Virtual Worlds. *ACM Computing Surveys*, 44(4):1–25, 2012. (cited on Page 27)

[LKPMJP06] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Applying Database Replication to Multi-player Online Games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames 2006)*, page Article No. 15, October 2006. (cited on Page 9)

[LLL04] Frederick W.B. Li, Lewis W.F. Li, and Rynson W.H. Lau. Supporting Continuous Consistency in Multiplayer Online Games. In *Proceedings of the 12th Annual ACM International Conference on Multimedia (ACM Multimedia 2004)*, pages 388–391. ACM Press, October 2004. (cited on Page 14 and 55)

[LLMZ11] Justin J. Levandoski, David Lomet, Mohamed F. Mokbel, and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *Conference on Innovative Data Systems Research (CIDR 2011)*, pages 123–133, January 2011. (cited on Page 66)

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. (cited on Page 30 and 31)

[McF13] Patrick McFadin. The Data Model is Dead, Long Live the Data Model. Website, May 2013. Available online at http://de.slideshare.net/

patrickmcfadin/the-data-model-is-dead-long-live-the-data-model; visited on February 3rd, 2016. (cited on Page xv and 32)

[MGG06] Jens Müller, Andreas Gössling, and Sergei Gorlatch. On correctness of scalable multi-server state replication in online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames 2006)*, page Article No. 21, October 2006. (cited on Page 54)

[Mic] Microsoft. Maximum Capacity Specifications for SQL Server. Website. Available online at https://technet.microsoft.com/en-us/library/ms143432; visited on January 24th, 2016. (cited on Page 54)

[Muh11] Yousaf Muhammad. Evaluation and Implementation of Distributed NoSQL Database for MMO Gaming Environment. Master's thesis, Uppsala University, 2011. (cited on Page 55)

[New14] Newzoo. Infographic: Global PC/MMO Gaming Revenues to Total $24.4Bn This Year. Website, November 2014. Available online at http://www.newzoo.com/insights/pcmmo-gaming-revenues-total-24-4bn-2014/; visited on July 16th, 2015. (cited on Page 1 and 5)

[NIP+08] Vlad Nae, Alexandru Iosup, Stefan Podlipnig, Radu Prodan, Dick Epema, and Thomas Fahringer. Efficient Management of Data Center Resources for Massively Multiplayer Online Games. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC 2008)*, pages 1–12. IEEE Press, November 2008. (cited on Page 10)

[Oraa] Oracle. Logical Database Limits. Website. Available online at https://docs.oracle.com/cd/B28359_01/server.111/b28320/limits003.htm#; visited on January 24th, 2016. (cited on Page 54)

[Orab] Oracle. MySQL Cluster Overview. Website. Available online at http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-overview.html; visited on August 21th, 2015. (cited on Page 14)

[Pan] Luca Pancallo. PlaneShift CrystalSpace Conference 2006. Website. Available online at http://crystalspace3d.org/downloads/conference_2006/planeshift_conf.pdf; visited on October 10th, 2015. (cited on Page 82)

[Pat12] Jay Patel. Cassandra Data Modeling Best Practices. Website, July 2012. Available online at http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/; visited on February 3rd, 2016. (cited on Page xv and 33)

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, pages 251–264, October 2010. (cited on Page 66)

[PGH06] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. Consistency Requirements in Multiplayer Online Games. In *Proceedings of the 5th Workshop on Network and System Support for Games (NETGAMES 2006)*, page Article No. 51, October 2006. (cited on Page 55)

[Roe11] Kevin Roebuck. *Cloud Storage: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors.* Emereo Pty Limited, 2011. (cited on Page 23)

[Sco10] Ben Scofield. NoSQL Death to Relational Databases(?). Website, January 2010. Available online at http://de.slideshare.net/bscofield/nosql-codemash-2010; visited on January 6th, 2016. (cited on Page xv, 28, and 29)

[She13] IE Sherpa. There are 628 Million MMO Players Worldwide Accounting for $14.9 Billion in Annual Revenue. Website, July 2013. Available online at http://www.iesherpa.com/?p=1177; visited on July 16th, 2015. (cited on Page 1)

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE, May 2010. (cited on Page 23)

[SKS+08] Arne Schmieg, Patric Kabus, Michael Stieler, Bettina Kemme, Sebastian Jeckel, and Alejandro Buchmann. pSense - Maintaining a Dynamic Localized Peer-to-peer Structure for Position Based Multicast in Games. In *IEEE International Conference on Peer-to-Peer Computing (P2P 2008)*, pages 247 – 256, September 2008. (cited on Page 54)

[SLBA11] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys and Tutorials*, 13(3):311–336, 2011. (cited on Page 21 and 24)

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*, pages 1150–1160, September 2007. (cited on Page 17)

[SSM11] Mirko Suznjevic, Ivana Stupar, and Maja Matijasevic. Mmorpg player behavior model based on player action categories. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games (NetGames 2011)*, page Article No. 6, October 2011. (cited on Page 8)

[SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris : Reliable Transactional P2P Key/Value Store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG (ERLAND 2008)*, pages 41–48, September 2008. (cited on Page 55)

[Sup14] Superdata. Top 10 Subscription-based MMOs. Website, July 2014. Available online at http://2p.com/7801190_1/Top-10-Subscribtion-based-MMOs-Elder-Scrolls-Online-Tops-772k-Subscribers-by-Apopl htm; visited on July 23th, 2015. (cited on Page 6)

[TRAR99] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed Consistency for Shared Distributed Objects. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing (PODC 1999)*, pages 163–172, May 1999. (cited on Page 27)

[Vah14] Vivek Vahie. 99 Problems but the Cloud ain't One: What are SaaS, PaaS and IaaS? Website, August 2014. Available online at http://www.tgdaily.com/enterprise/127086-99-problems-but-the-cloud-aint-one-what-are-saas-paas-and-iaas; visited on November 26th, 2015. (cited on Page xiii and 22)

[VCO10] Hoang Tam Vo, Chun Chen, and Chin Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. *Proceedings of the VLDB Endowment*, 3(1):506–517, 2010. (cited on Page 66)

[VCS+09] Marcos Vaz Salles, Tuan Cao, Benjamin Sowell, Alan Demers, Johannes Gehrke, Christoph Koch, and Walker White. An Evaluation of Checkpoint Recovery for Massively Multiplayer Online Games. *Proceedings of the VLDB Endowment*, 2(1):1258–1269, 2009. (cited on Page 88)

[Vog09] Werner Vogels. Eventually Consistent. *Communications of the ACM (CACM)*, 52(1):40–44, 2009. (cited on Page 14 and 27)

[Vol10] LLC VoltDB. Voltdb technical overview. *Whitepaper*, 2010. (cited on Page 17)

[Wan13] Shuo Wang. Towards Cloud Data Management for Online Games - A Prototype Platform. master thesis, University of Magdeburg, Germany, September 2013. (cited on Page 74)

[WFZ+11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages:

the Consumers' Perspective. In *Conference on Innovative Data Systems Research (CIDR 2011)*, pages 134–143, January 2011. (cited on Page 59)

[WKG+07] Walker White, Christoph Koch, Nitin Gupta, Johannes Gehrke, and Alan Demers. Database Research Opportunities in Computer Games. *ACM SIGMOD Record*, 36(3):7–13, 2007. (cited on Page xiii, 2, 9, 10, 11, 12, 15, and 54)

[WPC09] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Scalable Transactions for Web Applications in the Cloud. In *15th International Euro-Par Conference (Euro-Par 2009)*, pages 442–453, August 2009. (cited on Page 60 and 66)

[YK13] Amir Yahyavi and Bettina Kemme. Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey. *ACM Computing Surveys (CSUR)*, 46(1):Article No. 9, 2013. (cited on Page 54)

[YV05] Anthony Peiqun Yu and Son T Vuong. MOPAR : A Mobile Peer-to-Peer Overlay Architecture for Interest Management of Massively Multiplayer Online Games. In *Proceedings of the international workshop on Network and operating systems support for digital audio and video (NOSSDAV 2005)*, pages 99–104, June 2005. (cited on Page 54)

[Zep16] Zephoria. The Top 20 Valuable Facebook Statistics – Updated September 2016. Website, September 2016. Available online at https://zephoria.com/top-15-valuable-facebook-statistics/; visited on October 5th, 2016. (cited on Page 21)

[ZK11] Kaiwen Zhang and Bettina Kemme. Transaction Models for Massively Multiplayer Online Games. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 31–40, October 2011. (cited on Page 55)

[ZKD08] Kaiwen Zhang, Bettina Kemme, and Alexandre Denault. Persistence in Massively Multiplayer Online Games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES 2008)*, pages 53–58. ACM Press, October 2008. (cited on Page 1, 2, and 55)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 03.02.2017