University of Magdeburg

School of Computer Science

Master Thesis

Towards Cloud Data Management for Online Games - A Prototype Platform

Author:

Shuo Wang

14, October, 2013

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Ziqiang Diao
Department of Technical and Business Information System

Wang, Shuo:
Towards Cloud Data Management for Online Games - A Prototype Platform
Master Thesis, University of Magdeburg, 2013.

Abstract

Over the last decades, Massively Multi-player Online Role Playing Games (MMORPGs) have become big business. Along with the development of MMORPG, the game's content and entertainment is enriching. MMORPG has attracted a quite number of players. With the steeply increasing count of players, the data volumes of a game that needs to be persisted will also increase starkly. Nowadays the most popular data store system in MMORPGs is Relational Database Management System (RDBMS). However the traditional RDBMS cannot completely meet the requirements for data storage. Many MMORPGs, in the case of large users accessing, will encounter the database's bottlenecks, such as the big data volume and lack of scalability and so on. The traditional RDBMS has very limited scalability because of its subjection to the ACID rules. Most of MMORPGs have very complex server-side architecture, which makes the background database even harder to scale out. The traditional RDBMS are facing challenges in the MMORPG industry. My study focuses on using the Cloud database to solve these problems that RDBMS are facing. In this work, A simple MMORPG testbed has been implemented. It contains a server side with some game logic and a client side. The popular Cloud database Cassandra has been chosen to store the game data. We evaluate the reading and writing performance as well as the scalability of our game prototype. The results show that our prototype is able to meet the data management requirements of MMORPGs and it has a big potential to scale out in a large range and support very large number of players.

Contents

List of Figures viii

List of Tables ix

List of Code Listings xi

List of Acronyms xiii

1 Introduction and Background 1
- 1.1 Introduction 2
- 1.2 Motivation 3
- 1.3 Structure of Thesis 4

2 Related Fundations 5
- 2.1 MMORPG 5
- 2.2 MMORPG Architecture 6
- 2.3 Relational Database Management System 8
  - 2.3.1 Brief History 8
  - 2.3.2 Structured Query Language 8
  - 2.3.3 ACID Properties 8
  - 2.3.4 Schema 9
- 2.4 Limitations of RDBMS in MMORPG 10

3 Cloud Database Management 13
- 3.1 Cloud Computing 13
- 3.2 Cloud Computing Service Model 14
- 3.3 Cloud Database 15
  - 3.3.1 SQL Database vs. NoSQL Database 17
  - 3.3.2 Data Model 18
  - 3.3.3 ACID vs. BASE 19
- 3.4 Cloud Database Products 22
- 3.5 MMORPG Data Storage in Cloud Database 23

4 Cassandra 25
- 4.1 History and Development 25

4.2 Cassandra Data Model . . . 25
4.2.1 Cassandra Ring . . . 28
4.3 Transaction and Concurrency Control . . . 30
4.3.1 Atomicity . . . 30
4.3.2 Tunable Consistency . . . 30
4.3.3 Isolation . . . 31
4.3.4 Durability . . . 31
4.3.5 Read and Write . . . 31
4.4 Query Tools in Cassandra . . . 32

5 Design and Implementation **37**
5.1 Starting Point . . . 37
5.1.1 Infrastructure . . . 37
5.1.2 Functional Requirements . . . 38
5.2 Prototype Architecture . . . 39
5.3 MMORPG Database Schema Design in Cassandra . . . 40
5.3.1 Tranditional Game Database Schema . . . 40
5.3.2 Different Design Concepts . . . 43
5.3.3 Schema Transformation . . . 44
5.3.4 Comparison . . . 48
5.3.5 Creation and Query of Column Families in Cassandra . . . 49
5.3.6 Deployment of Cassandra Cluster . . . 52
5.4 Game Server Design and Implementation . . . 53
5.4.1 Darkstar . . . 53
5.4.2 Data Structures . . . 56
5.5 Game Client Implementation . . . 58
5.5.1 JMMORPG . . . 58
5.5.2 Client Libraries . . . 59

6 Evaluation **61**
6.1 Experimental Environment . . . 61
6.1.1 Experimental Clients . . . 61
6.1.2 Configuration of Cassandra . . . 62
6.1.3 Methodology . . . 62
6.1.4 Acquisition and Expectation of Experimental Results . . . 63
6.2 Experiment Results and Discussion . . . 64
6.2.1 Performance of the Prototype . . . 64
6.2.2 Scalability of the Prototype . . . 66

7 Conclusion and Future Work **73**

Bibliography **75**

List of Figures

2.1 Typical architecture of a MMORPG [WKG+07] 6
2.2 A Cloud-based MMORPG architecture adapted from[DSWM13] 7
2.3 A simple E-R model and relational data model 10

3.1 Cloud service model adapted from[Bok10] 14
3.2 Efficiency vs. control[Spr12] 15
3.3 Different properties in NoSQL according to Brewer's theorem[GL02] . . 19
3.4 The popular databases products in different group of CAP adapted from [Hew10] 20

4.1 The structure of a column 26
4.2 The structure of a row 26
4.3 Cassandra column family stores sorted rows and columns 27
4.4 Cassandra cluster ring adapted from [Dat13e] 29
4.5 Cassandra client write request[Dat13e] 32
4.6 Cassandra client read request[Dat13e] 33

5.1 The infrastructure of the prototype 38
5.2 The test platform architecture 39
5.3 E-R model design of prototype 42
5.4 Cassandra column family design derived from E-R model 45
5.5 Cassandra column family inventory sample 47
5.6 Cassandra column family event log sample 48
5.7 Datastax Cassandra OPSCenter 53

5.8 Class diagram 54
5.9 DarkStar architecture adapted from [Bur07] 55
5.10 DarkStar core component adapted from [Bur07] 56
5.11 Sequence diagram 58
5.12 Server class implementation 59
5.13 A screen shot of our prototype(client side) 60

6.1 Average response time of 200 clients 63
6.2 Reading/Writing response time for group 1 65
6.3 Reading/Writing response time for group 2 65
6.4 Reading/Writing response time for group 3 66
6.5 Group 1: Performances of 1 nodes Cassandra 67
6.6 Group 2: Performances of 2 nodes Cassandra 68
6.7 Group 3: Performances of 3 nodes Cassandra 68
6.8 Group 4: Performances of 4 nodes Cassandra 69
6.9 Group 5: Performances of 5 nodes Cassandra 69
6.10 Average reading time of 5 groups 71
6.11 Average reading time of 5 groups(modified) 71
6.12 Average writing time of 5 groups 72

List of Tables

3.1 Classification of Cloud database products adapted from [MCO10] . . . 22
3.2 Data classification and requirements analysis adapter from [DSWM13] . 24

4.1 Relational model analogy 28
4.2 Replicas table 30

List of Code Listings

4.1 Using CQL creating keyspace sample 33
4.2 Using CQL creating column family sample 34
4.3 CQL insert and select sample 34
4.4 CLI creating a keyspace 35
4.5 CLI creating a column family 35
4.6 CLI set users information 36
4.7 CLI get uses information 36
5.1 SQL syntax sample 43
5.2 CLI creating a keyspace 49
5.3 CLI creating a column family 50
5.4 Using a ColumnQuery to get money of a user 51
5.5 A sample of using a SliceQuety 51
5.6 A sample of using a RangeSliceQuety 52
5.7 Data structure game player 56
5.8 Data structure command 57

List of Acronyms

API	Application Programming Interface
CLI	Command Line Interface
CQL	Cassandra Query Language
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language
E-R	Entity-Relation
IaaS	Infrastructure as a Service
JDK	Java Development Kit
JMX	Java Management Extensions
MMOFPS	Massively Multi-player First Person Shooter
MMOG	Massively Multi-Player Online Games
MMORPG	Massively Multi-player Online Role Playing Game
MMORTS	Massively Multi-player Online Real-Time Strategy
NF	Normal Forms
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
PDA	Personal Digital Assistant
PDS	Project Darkstar Server
RDBMS	Relational Database Management System
RPC	Remote Procedure Call

SaaS Software as a Service
SQL Structured Query Language
SSH Security Shell

VPN Virtual Private Network

1. Introduction and Background

Massively Multi-Player Online Games (MMOG) has become more and more popular over the last years. In an MMOG, players can play games with others in the world. The Online games are usually more interesting than the normal PC game, because the players of online game are playing against or cooperating with other human rather than the machine. Despite the artificial intelligence getting more and more advance, it is still quite far from replicating the challenge and pleasure of playing against another human. Normally, a player in an online game is not alone, he will build their own social circle and find new friends[FBS07]. This social circle will make the game more attractive to the player. People could get more fun in online games and even a sense of accomplishment. According to the Niko report, the Chinese online gaming market has grown to more than 9 billion dollars at 2012 year end[int13]. Another report on global MMOG Market predicts, the MMOG will become the second largest gaming segment in the next few yeas, and the most growing area will be the market APAC[1] and Eastern European regions[17113].

The MMORPG is an important member of MMOG. It distinguish from other online games by allowing thousands of player playing in a single game world. Our study will focus on this kind of game. The growing number of players makes the data volume increasing and getting the architecture more complex. The existing game architecture adopting a relational database for storage of massive user data, with the growing data volume and the number of players, the drawback of Relational Database Management System (RDBMS) is slowly exposed. The players of a game has elasticity, thus no one can predict how many users can a new MMORPG attract. Bruce Wookcock' research finds that, when a new MMORPG coming into market, it will attract many users at beginning and then this number will decreasing and the workload for online game is unpredictable over longer periods of time[PS09]. If the game gets popular over time, the number of player may exponentially growth. A large amounts of play-

[1]APAC: Asia and Pacific

ers may playing game at same time, and this will cause a big access pressure on the database. The game vendor have to purchase more game servers and database servers to support the growing players. In contrast, the purchased servers could be a kind of waste if the players continuously reduced[FBS07]. In some large popular MMORPG, there will be usually thousands or even ten thousand players access the game database concurrently. The database of an MMORPG must have scalability, consistency and highly performance. However the traditional RDBMS cannot satisfy the requirement mentioned above simultaneously[DSWM13]. So we need a kind of new database, which should have elastic scalability and has the ability to handles large amount of data. A Cloud-based database could be a potential solution for MMORPG game data store.

1.1 Introduction

Many Massively Multi-player Online Role Playing Games (MMORPGs) have a lot of subscriptions, for instance the World of WarCraft has millions of players in the world and many game companies have relying MMORPG achieved big commercial business. A single-node game server and database is not possible to support so many concurrent users playing. Many studies have focus on how to scale a game server so that it can support more user concurrent online. Those researches proposed Peer-to-Peer based architecture to scale the server in order to improve the performance of the massively multi-player games [KLXH04][AT06]. Some researches proposed novel algorithm to improve networking or reduce latency. Assiotis has a solution for MMORPG. In this model, the game world is divided into many small areas, every filed will be handled by a single node, he has also proposed a new algorithm to reduce the package size. The proposed model is scalable and the players in different field can interact with each other seamlessly [ZKD08]. Some studies give new idea to optimize the game state, they present a new way to classify and store game data. Some of the optimization is based on object database. It could be very convenient when we persistent the game data. But object database is hard to scale. Some other paper optimized the data structure onto relational database, for instance, Zhang's paper describes a new efficient way to persist game state [DGK$^+$09], he identifies different consistency categories and gives them different priority. For category with low consistency level, the storage strategies with low overhead will be applied. This method provides efficient writes and can be dynamically adjusted. It worth nothing that most of studies are focused on RDBMS, which has limited scalability due to its ACID constraint and complex architecture. The ACID constraint guarantees the database transactions are reliably processed. Many database rely upon locking to provide ACID capability. Lock means that a transaction marks the data as locked, so that other transactions cannot modify the data until it releases the lock. Locks means competition, which will result some processes waiting until the resource be released. This lock mechanism will lead to performance loss if the RDBMS scaled to a large size. Beside, the RDBMS is difficult to scale out due to its complex structure. But many of popular MMORPGs adopt a RDBMS to persist game data[BBG10], for example EVE has applied SQL database[2]. Even an

[2](http://doublebuffered.com/2008/10/05/can-eve-evolve/)

MMORPG with a mature commercial database be applied. The MMOGs struggle to achieve much more than 500 transactions per second. And this transaction rate cannot be improved by simply adding new node in cluster. With the increasing number of players and concurrent database access, the RDBMS becomes a bottleneck in a lager scale MMORPG.

Nowadays the research on Cloud computing is a hot topic. The Cloud-based NoSQL storage model could be a potential solution to the problem that RDBMS faced. The advantages of Cloud database are easy to scale and high performance and availability[BBG10]. Cloud database is in infancy, so researches on the possibility to apply a NoSQL Cloud database on a MMORPG are not so much. A master thesis from Muhammad has implemented a Snowbox battle game[Muh11]. The game persists game state in a NoSQL Cloud database Riak. This architecture has good scalability and performance. However, in his paper, he didn't describe the transaction and consistency problem. Other paper from Diao has also proposed a Cloud-based MMORPG model[DSWM13]. He has classify the game data in different group and made an appropriate Cloud-based persistence solution for MMORPG. He point out that, the Cloud-based database can be used to store non-sensitive data. The privacy data and the data need highly consistency or transactions are not suitable for Cloud database. Cloud database can't displace RDBMS completely. There is also no "one fit all" database products. We should for different propose chose different database products. The Cloud database has many drawbacks need to be addressed, such as consistency, transaction and data privacy[DWC10], access control[Ram09] and so on. Cloud database are not intend designed for MMORPG, and the existing SNS(Social Networking Services) on top of Cloud database are very different from MMORPG. Comparing MMORPG to SNS, such as Facebook and Twitter [Aba09], the SNSs have normally very large number of users, and it doesn't need higher consistency. Because it is total acceptable that my friends two minutes later see my status after I updated it. However, MMORPG needs some kind of consistency, some of the game data need to be performed in transaction[DSWM13]. For this reason, the Cloud database cannot be applied to MMORPG directly.

1.2 Motivation

The MMORPG architecture will benefit from Cloud database and get more scalability. However, there are only few publications that discuss the Cloud based MMORPG. This thesis will introduce the character of Cloud database to the architecture of MMORPG. As the idea from [DSWM13], who classified the game data as several Categories according to different consistency demands. Thesis thesis will combine the characteristics of Cloud database and the requirements of MMORPG. The Cloud database should be more suitable for storing MMORPG game state after improvement. We will design and implement a Cloud-based MMORPG testbed. The problems that encountered by implementing will be discussed. And we will evaluate the performance and scalability of the game testbed.

1.3 Structure of Thesis

This Thesis will be structured as follows:

Chapter 1 Introduction and Background:

This chapter introduces the background information of this thesis and the development of MMORPG market as well as the motivation of this paper.

Chapter 2 General Information:

This chapter introduces the architecture of MMORPG and describes the foundation of the Cloud database. The most popular Cloud database products are compared and we also illustrated why we chose Cassandra as the database for our testbed.

Chapter 3 Cloud Database Management:

In this chapter, the basis and features of Cloud computing and Cloud database, the possibility of its application in MMORPG as well as the advantages and drawbacks are introduced. The Cloud database has normally high scalability and can handle large data volume. It is the highest potential solution for the bottleneck of database problem in MMORPG.

Chapter 4 Cassandra:

This chapter gives the basics information about the data model and infrastructure of Cassandra, which will be important for our prototype design and implementation. Cassandra is a key-value NoSQL database, which has the high scalability and availability.

Chapter 5 Design and Implementation:

Chapter 5 illustrates the processes from design to implementation of our prototype's client side and server side, and the solutions of the problems we met during this processes. During the implementation of our prototype, it is quite different between RDMBS table design and Cassandra column family design. These differences and relative merits will be discussed in detail here.

Chapter 6 Evaluation:

This chapter gives the methodology and result of our experiments, based on which the performance and scalability of our prototype are evaluated.

Chapter 7 Conclusion:

In this chapter, the conclusion and summary based on our experiment results are presented here. Some limitations and possible improvements of this thesis work are also outlined here.

2. Related Fundations

2.1 MMORPG

MMORPG is an abbreviation of Massively Multi-Player Online Role Playing Games[1]. It is the subset of the MMOG (Massively Multi-Player Online Game). An MMOG is a computer game, which is capable of supporting multilayer in parallel. It normally consists of a client side and a server side. The game logic of an MMOG runs on server side to protect the game from malicious manipulation. The MMOG server simulates a giant persistent virtual world through computing, database operating, receiving and handling the request from client [Bar04]. Depending on the different game world, the MMOG has lots of other variants[2]. For instances, Massively Multi-player Online Real-Time Strategy (MMORTS), the players of this kind of game can build their own military in the game world and use different strategies to against others, such as"settlers Online" [3]. "MMO Rhythm game" is a kind of music game, of which the players have to do some actions following the music rhythm, and the winner is the player who do with the most accuracy. MMO Racing simulate a virtual world, in which the player can buy cars and racing with friends; MMO Social game is a game aimed for player's social communications in a virtual world; Massively Multi-player First Person Shooter (MMOFPS) creates a virtual world, in which players can shoot in their own views with or against other players. However the virtual worlds of those MMOGs mentioned above are mostly divided into many small sessions, in which only several players compete in a separate room or a battle field. A server will create an isolate space for each group and the space will be destroyed after the one single game. A MMORPG distinguishes itself from the others by allowing a very large number of players to interact with each other in one virtual shared game world and the game world of MMORPG will keep running even in case of no player in it. moreover, unlike other type of MMOG, which only need

[1] http://en.wikipedia.org/wiki/Massively_multiplayer_online_role-playing_game
[2] http://www.sciencedaily.com/articles/m/massively_multiplayer_online_game.htm
[3] http://www.mmobomb.com/review/the-settlers-online

to persist users' account and player's informations. An MMORPG must persist large data volume such as inventory, skills, social relation, etc. for all of the players for a very long time and part of data will be modified constantly by many current users. In an MMORPG, a player can create an avatar, who controls his avatar to kill monsters or complete quest to get experience. If the avatar of the player gets enough experience, he will get stronger and gain more skills and abilities. The MMORPG are meant to be played for a long-term and the players will spend months or years on a single character [ZKD08]. My thesis will investigate the performance of a MMORPG by introducing Cloud database into the architecture of traditional MMORPG storage and a testbed environment will be create as a small MMORPG world to verify my idea.

2.2 MMORPG Architecture

Normally a highly abstracted structure of a basic MMORPG is composed of a client and a server. The client is an interface through which the users connect with the server and interact with other players. Game client could be variety, such as a smart phone client, Personal Digital Assistant (PDA) client, browser or a PC application. A game server seems to be a complicate software system. However, a MMORPG is just a carefully organized relative simple component. Although individual games may differ slightly in details of their implementation, the underlying architecture are usually similar [DSWM13][Bur07]. A highly simplified view of MMORPG server architecture is typically consist of four parts[HYC04], Map/Application server, Database server, Transactional Data Cache and Directory server[CDG+08][CKSW02]. The Figure 2.1

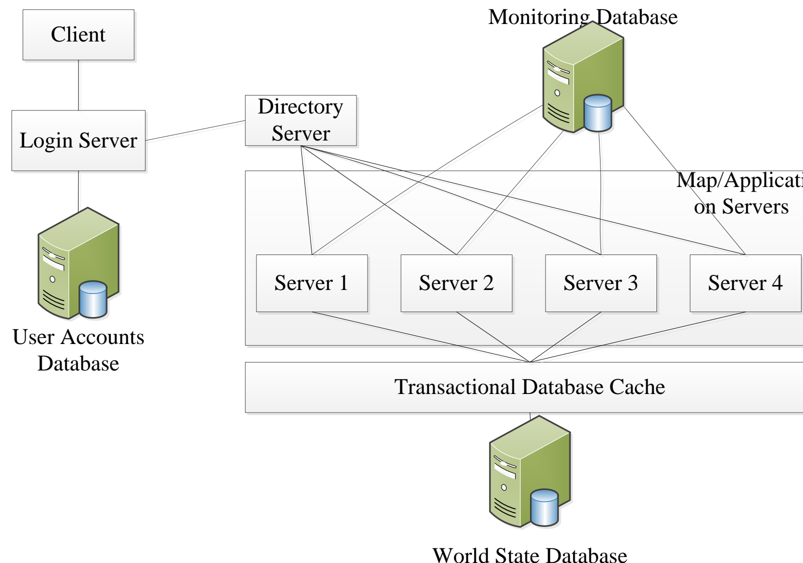

Figure 2.1: Typical architecture of a MMORPG [WKG+07]

shows the typical basic structure of a full functional game server running on top of

RDBMS. If a player sends a request to the server for logging in, he/she will type in his/her account from client side. The login server will receive the login data and verify the user's identity in cooperation with the users' accounts database. If the validation is passed, the login server will send an encrypted user id along with a token back to the client. The client use this token to communicate with the Directory server and the Directory server will then arrange the validated client session in one of the Map/Application servers. All the users' behaviors in the game must be monitored and stored in the monitoring database in order to maintain the order of the virtual world. While the player is gaming, the server computes the global state of the game world based on the actions submitted by players, and the new relevant state information will be sent to the client in real time within 200ms[CHL06]. All the game related data are stored in World State Database by the transactional data cache. According to a study, to ensure fluent play, the response time of an MMORPG must be kept less than 1250ms [FRS05], while keeping lower latency is critical for keeping the player engaging. With the increasing number of players, the game architecture based on RDBMS is hard to maintain and the underlying RDBMS will face the bottleneck of limited scalability.

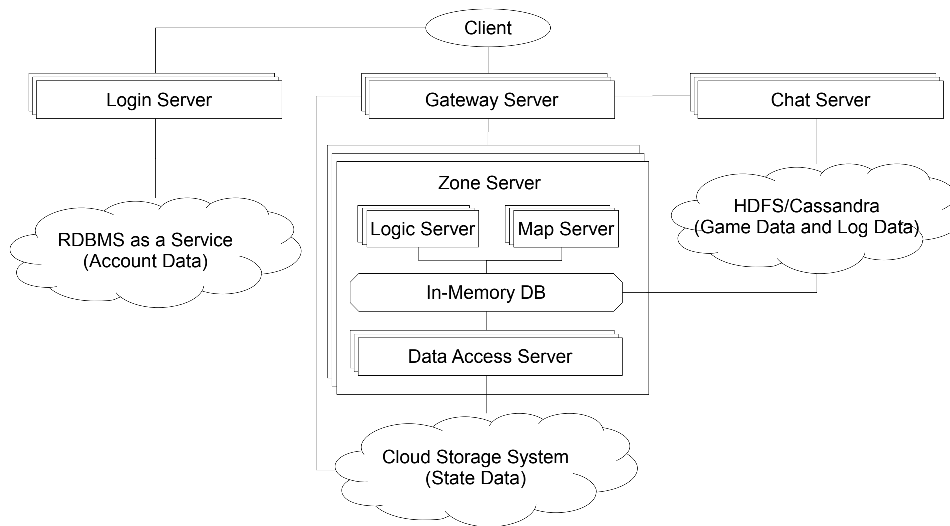

Figure 2.2: A Cloud-based MMORPG architecture adapted from[DSWM13]

Furthermore, for supporting thousands of concurrent players, the game vendor should maintain a large static infrastructure with hundreds to thousands of distributed servers in order to provide the required quality of service[AT06]. For example, the World of Warcraft, has over 10,000 serversnaemassively. As the study shows, the MMORPG's demands of resource are highly dynamic and thus a large portion of the statically allocated resources are wasted [NIP^{+}08]. This leads to very inefficient resource utilization. The author proposed an ecosystem for MMOGs, whose underlying technology is Cloud computing. His system reached efficient on-demand resource usage by hosting the MMOGs on the Cloud [NPI11]. But this study focused on the game server itself,

so the drawbacks of RDBMS problems are still not solved. To addressing the database limitation in the MMORPG architecture, Diao has proposed a new way by introducing the Cloud database into the MMORPG[DSWM13].

Figure 2.2 on the previous page shows the simplified architecture proposed by Diao. The general idea of this structure is managing the data of online games according to their different consistency requirements. Such as game account data and billing data need strong consistency and highly security, so they should always be performed as a transaction and must be stored in a RDBMS which has good support for transactional operations. And for game and state data which has strong requirements for flexibility and availability will be stored in the Cloud. The log data will not be modified any more after it's persisted in a database. It's typically large-scale and usually analysed after a long period of time. So the demand of consistency and performance are also not so strict that they can be stored in a Cloud database such as Cassandra[DSWM13].

2.3 Relational Database Management System

2.3.1 Brief History

The first idea of relational Database was brought by Edgar Codd with his paper "A relational model of data for large shared data banks" [Cod01]. In this paper, he has firstly proposed the theory of a relational database, which has a big effect on today's database. RDBMS has been developed and evolved for more than 40 years and it has been proved to be very mature and reliable . Nearly 80% applications have applied a RDBMS for their data persist[Hew10]. We can say that, In RDMBS stored the whole world.

2.3.2 Structured Query Language

There are many reasons why the relational database has become so popular in the world. One of the important reasons is RDBMS has adopted a Structured Query Language (SQL). SQL is powerful for some reasons. First, it allows user to perform complex operation with data by using simple Data Manipulation Language (DML) such as insert, delete, update and it supports grouping aggregate and summary functions. SQL provides a variety of operations for creating, altering and deleting data schema in run time by using Data Definition Language (DDL). SQL also allows users to manage access rights by using Data Control Language (DCL). Second, SQL has a standard basic syntax, which can be learned quickly. Even you changed your storage system from one vendor to anther, the standard SQL language will be still supported by every RDBMS. Except SQL, there are also many intuitive graphical interfaces supported by different database vendors for viewing and working with relational database.

2.3.3 ACID Properties

Besides the features we mentioned above, the RDBMS support an important feature, it is transaction. A transaction is like the "transformation of state". As Jim Gray described

[4], the transaction executes the operation virtually in the database at first and checks if all the operations is going well. If yes, the operations will be committed; if not, it allows programmer to roll back. Operations of a transaction must be performed successfully all together or none. The transaction in RDBMS must obey the ACID rules.

- **Atomicity**: identifies that the transaction is atomic or so called “all or nothing”, all the statements within a transaction is either fully completed or none of them is done. That means if one part or some parts of a transaction fails, the entire transaction is recognized as failed. In this case the database state is returned to the state it was in before the transaction was started and left the system totally unchanged. An atomic system must guarantee atomicity in all kinds of unexpected and urgent situations, such as power failures, errors, and unknown crashes. An example of an atomic transaction is dealing. The money is removed from the player’s account and then the item goes into the player’s inventory. If the system fails after removing the money from the player’s account, then the transaction processing system will give the money back to player. This is so called “a rollback”.
- **Consistency**: means the data in database can only move from one valid state to another. Any data which be persisted into the database must in a valid status according to all defined rules. It is impossible to provide the readers with different values what make no sense together. Once a error happens in a transaction, the system will in a invalid state, then any of the changes made by this transaction must be rolled back.
- **Isolation**: means that transactions executing concurrently will not affect each other. This property ensures that the concurrent transactions results in the same system state as they were executed one after the other. If there is one transaction runs in isolation, it seems to be the only action that the system is executing; if there are more than one transactions that are all performing the same function at the same time, transaction isolation will ensure that one transaction will not aware of the others. If transactions were not running in isolation, they could access data in an invalid state from the database.
- **Durability**: means that once a successful transaction result is persisted in a database, it will be permanent and never get lost even in case of power loss, crashes, or errors. To defend against unexpected cases, transactions must be recorded in a hard disk instead of a volatile memory. The concept of durability allows the developer to know all the transactions have been permanently stored in database, regardless of what happens to the system later on.

2.3.4 Schema

In RDBMS, data is organized in tables. A Table consists of a set of tuples and represents a relation. Each column in the table called an attribute that describes a certain domain

[4]http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf

and the attribute(also called fields) is usually named. One or a group of columns can be specified as a table's primary key. The primary key are used to uniquely identify a row. To design a relational schema, the Entity-Relation (E-R) Model is often used to mapping the real world into relational model. The ER-Model represents the relation between entities. In order to reduce data redundancy the 3-Normal Forms (NF)[5] are always considered during the database design. 3-NF is introduced by the inventor of the relational model, Edgar F. Codd, which is used to organize the fields and tables in a relational database in order to minimize redundancy and dependency. The core idea of 3-NF is to divide a large tables into smaller tables according the relationship between them. Figure 2.3 shows a simple example of E-R model and the corresponding relational table.

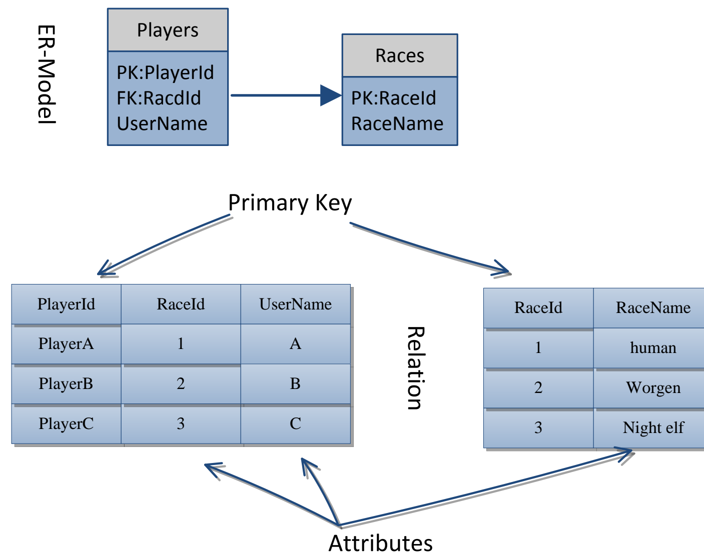

Figure 2.3: A simple E-R model and relational data model

2.4 Limitations of RDBMS in MMORPG

As we discussed above, one single node Database System is impossible to satisfy the requirements of data accessing in a MMORPG with highly concurrent players. The qualified data persistent System in an MMORPG must ensure data consistency, efficiency and scalability[ZKD08]. Unfortunately, the existing RDBMS are hard to satisfy all the requirements simultaneously[WKG+07]. Assuming we have an MMORPG running on top of RDBMS, if the number of players in this Game goes up, the database will encounter a scalability problem. Almost all relational database are support join operations, which reduce the system performance. The transaction guarantees the relational database consistency by locking some portion of the database, so this portion

[5]3 Normal Forms: http://de.wikipedia.org/wiki/Normalisierung_(Datenbank)

of data is not available to other clients. This can induce the game server to very heavy workloads because locking means the clients have to wait for their turn in queue to access data. Typically, we can solve those problems by the following ways: firstly, we can add more memories, more faster processors and disks, and moving the game to more powerful computer. This way called vertical scaling[6]. Vertical scaling has reasonably well for data but has its limitations: expensive and always need the next generation larger system. If we already have a most powerful machine but the problems still arise, the solution is similar: since one machine is heavily loaded, we can simply add new machines into the database cluster. We group data by function and spread them across databases. We also need to split data within functional areas across multiple databases. If we do so, we have to face the data replication and consistency problems in the multi-node cluster, which will never arise with single machine. Fortunately there are lots of researches on how to manage transaction and maintain ACID rules in a distributed database scenario[ÖV96]. In order to solve the conflicts in distributed database, a locking based concurrency control are widely used. A simple example is the two-phase locking rule, which make sure that no transaction should request lock after it releases one of its locks, alternatively, a transaction should keep all locks until it has no request for another lock[TèV11]. Other algorithms are a little bit different, but the basic idea is also locking some data. So we want to go ahead optimize our system and try to improve indexes and queries. The Game is built and we understand the primary query paths. To avoid join operations in big tables, we can also denormalizes some data to make it more like the queries that access it. This way is somehow against the Codd's 12 commandments for relational data[Cod85]. But it bring us more performance by generating some data redundancy. In short, the traditional database has following limitations when it applied to MMORPG.

1. Hard to scale-out: The RDBMS, due to its strictly ACID constraints, is hard to scale-out. As mentioned above, the MMORPG need an elastic database.

2. Performance: Relational database is based on relational algebra. It models and stores the data to relational model, reads data from RDBMS, and then models data back to their original model. This consumes considerable computing resources and hence the reading and writing are not efficient.

3. Data Complexity: Data in an RDBMS are stored in multiple tables, which link to each other through foreign keys. In MMORPG scenario, the state data are even more complex, so in this highly concurrent data accessing situation, RDBMS is not able to handle the operation efficiently.

4. Cost: In order to set up a relational database cluster, the game publisher should purchase.an expensive license from the database vendor, which normally increase the cost dramatically.

[6]Vertical scaling, also described as scale up, typically refers to adding more processors and storage to an Symmetric Multiple Processing to extend processing capability. Generally, this form of scaling employs only one instance of the operating system(IBM).

5. Structured Limits: RDMBS has a fixed schema, so that the structure of a table must be pre-defined. However, a MMORPG's life circle is always bug-fixed. If some new features are added into MMORPG, alteration of the existing table structure.

In summary, RDBMS is good at solving certain problems, but it cannot freely scale out due to its constraints. If we want to scale a relational database, we should avoid join operation, which means denormalizing the data to achieve a better performance, In order to do that, we have to maintain multiple replications of data, which causes the development of application and database harder than before. For this reason, a new DBMS is required to address these problems. In the next chapter, we will discuss the Cloud database management in MMORPG scenario.

3. Cloud Database Management

3.1 Cloud Computing

Cloud computing is a an increasing hot technology today. However, there are lots of definitions of Cloud computing available[VRMCL08][BBD+08][GK08]. Most of them are either too general or too specific. From technique, commercial or users' perspective, here is a rather fitting one from the United States National Institute of Standards and Technology (NIST)[1].

> *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configure computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".*

Cloud computing is not a new technology rather than a new way to make use of existing technology and devices. It is a new approach to shared data and computing resources over a scalable network[Aba09]. Cloud computing system connects the huge computing pool together to provide a variety of IT services. Its core idea is unifying management, scheduling the computing resource in network and providing the end user a on-demand service[AFG+10]. It provides companies the ability to quickly react to the increasing demand on computing power. From the users' perspective, the resource on the Cloud is unlimited scalable ,instant acquirable and pay as you go.

The emergence of Cloud computing makes it possible that the computing resources(physical resources,infrastructure,middle-ware platforms and applications) can be provided and consumed as other kind of products just like gas, water and electricity. The enterprise

[1]NIST http://www.nist.gov/itl/cloud

and users no longer have to purchase expensive hardware and maintaining infrastructures but could rent these computing resource via internet. The Cloud computing technology will bring the computer industry a essential and revolutionary change.

3.2 Cloud Computing Service Model

The service models of Cloud computing can be typically categorized to three different levels as Figure 3.1 shows. The lowest level is called Infrastructure as a Service (IaaS). Platform as a Service (PaaS) is built on top of IaaS. The Software as a Service (SaaS) is built on PaaS and IaaS. Every layer has different aimed user groups. The essential layer, aims to be the network architects, provides more flexible resources but also needs more efforts for management. The users of PaaS are normally application developers but the top layer SaaS usually provides applications for common users. So generally the top layer has the most population of users.

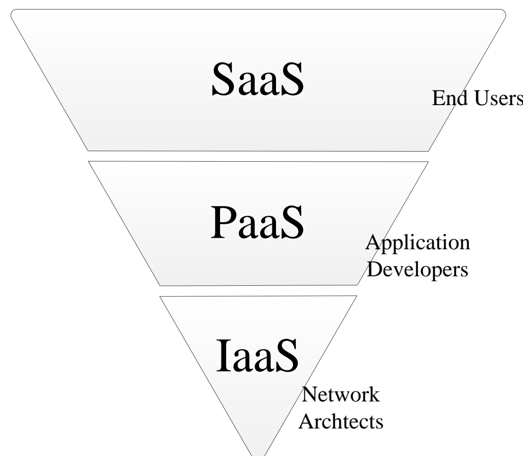

Figure 3.1: Cloud service model adapted from[Bok10]

- **Infrastructure as a Service** - is the lowest layer and offers the most basic services. This layer offers virtualized computer hardware resources such as databases, CPU power and memories. This layer also allows user to install their Operating system and deploy their software. By doing so, the users are free to create their own environment and gain more flexibility. However, they are also responsible for updating and patching their operations systems in this level[Var08a]. Amazon Web Server[2] is a popular IaaS example.

- **Platform as a Service** - within this layer, the providers offer a computing platform which includes operating systems and programming language execution

[2]AWS http://aws.amazon.com

environments. The user can develop their applications by using computing platform and tools offered by their providers. In this situation, programmers don't need to worry about maintaining the infrastructure[VVE09] and can better focus on the business logic. Some popular examples of this layer are Google App Engine[3] and Microsoft Azure[4].

- **Software as a Service** - The providers of this layer offer the users a collection of software and application programs. The users can simply use a web browser to get access to the software that the others have developed already without worrying about the back-ups and maintenance of the software. SaaS is also known as "software on demand" or "software as needed". In this level, the users can not get access to the underlying infrastructure[VVE09] so they have little flexibility. Google mail and Google office[5] are popular examples for this layer.

The conflicts between efficiency, flexibility and cost of the above mentioned three different service models can be seen from Figure 3.2. The efficiency is increasing when the user moving from standalone servers to a SaaS service model and the cost can decrease but the flexibility is decreased because they cannot control the hardware.

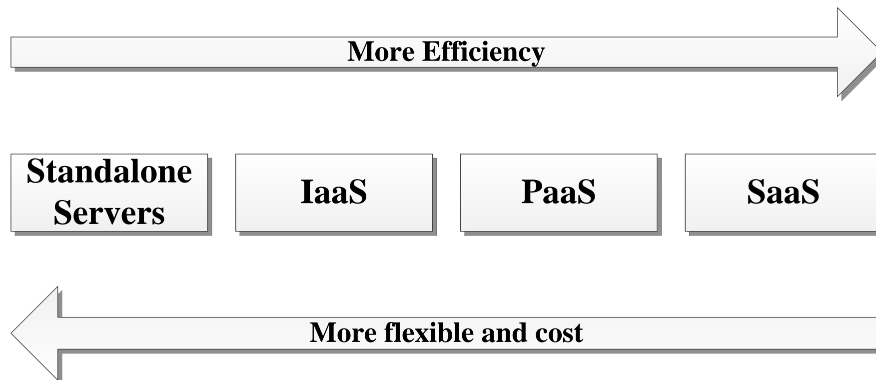

Figure 3.2: Efficiency vs. control[Spr12]

3.3 Cloud Database

Cloud database, working as a normal database for users, generally deploys a database software on top of a Cloud computing infrastructure. Cloud database may be directly accessed via a browser or Application Programming Interface (API) provider

[3]Google App Engine https://cloud.google.com
[4]Microsoft Azure http://www.windowsazure.com
[5]Google http://www.google.com

by Cloud database vendor. It is primarily a PaaS in the Cloud service model as we mentioned above[MCO10]. Many big database vendors are commercializing Cloud database into Cloud database products. such as Amazon SimpleDB[Var08b], Google bigtable[CDG+08] and so on.

Attributes of a Cloud Database

Cloud database is a new data management concept with the development of Cloud computing. But it is not simply taking a traditional RDBMS and running it in a Cloud environment. Cloud database is capable to handle large-scale of data volume and it provides availability and high scalability. We could say that, the Cloud database represents the future of database industry. A Cloud database has typically the following attributes.

- **Elasticity**: using a Cloud database, the user can add and remove nodes any time. The Cloud database could then balance workloads automatically. The MMORPG can especially benefit from this "load balancing" feature, which enables the game vendor to add or remove their storage resources in response to the number of players.
- **Scalability**: theoretically, the Cloud database has the unlimited scalability and can always satisfy the increasing demand on data accessing. For this reason, a Cloud based MMORPG can simply hold large amounts of players' data and heavy accessing pressure by simply adding new nodes in the cluster.
- **High availability**: a Cloud database has normally high availability because the nodes in the cluster are distributed in different physical locations and the data are duplicated so that every node holds only a subset of the whole data. This ensures the whole database always available even in case of single node failure or data unavoidable emergency.
- **Low-cost**: Cloud database applies typically a multi-tenancy. That means the user can rent the database from a database provider, so they could save the cost for maintaining infrastructure and human resources. A big Cloud database cluster can provide services for many users, so the using-cost will also be reduced. When a user rent a Cloud database, the cost for purchase of hardware and software can be avoided but the payment for consumed resource is necessary.

There are also some drawbacks. For example, all the users share the physical resources in a Cloud infrastructure, the data security and privacy cannot be guaranteed[MT05]. Moreover, without maintaining the infrastructure means users cannot fix some unexpected problems in time.

3.3.1 SQL Database vs. NoSQL Database

In this section, we will make a simple comparison between SQL database and NoSQL database. There are many differences between SQL and NoSQL database from different aspects. In a RDBMS stored structured table, the data in the table can be queried by using standard SQL language, which NoSQL don't have. NoSQL, is a summary term describes a set of non-relational databases, may scale out horizontally to a very large size but not guarantees ACID properties. NoSQL stands for "not only SQL". So NoSQL is not about to do not using SQL any more, but not only limited to SQL. NoSQL is not a terminator of SQL, but an alternative or enrichment to the SQL World[Sto10]. Generally, SQL is not in minority compare to NoSQL, It is still the first choice for most database problems. SQL database exists for a very long time and almost everyone who works related with programming is familiar with relational database. In addition, with the development of many extensions of SQL systems, working with SQL system becomes more and more easier. In the SQL database field, it has a big amount of mature products and a large number of tutorial,support, etc. available. That is why SQL database still remains the first choice position for all database problems. We will now give a short comparison from aspects of scalability, performance and consistency between SQL database and NoSQL database.

- **Scalability:**

  The historical NoSQL database models are aimed to fix some disadvantages of RDBMS. Scalability is one of the most important characters for distributed database and it is also the main reason why NoSQL comes into the world[Sto10]. Especially the horizontal scalability, the horizontal scaling of a SQL database requires more administrative overhead after it scale out to a certain size, the performance of the scaled SQL database will decrease significantly. While on the other hand, NoSQL don't need a fixed table structure that makes NoSQL particularly suitable for scaling out. The architecture of many NoSQL databases are also running on inexpensive computers for data storage. Normally, the NoSQL system can reach a very high scalability by simple adding new nodes into the cluster, even during the runtime. NoSQL systems can provide well and stable scalability constantly despite of the high volume of data. So the NoSQL database is superior compare to SQL database with the aspect of scalability.

- **Performance:**

  With similar reasons as in the scaling, the NoSQL databases has flexible schema, no join and n lock, that makes NoSQL databases are more efficient than the SQL databases. One of the reasons why NoSQL exists is that SQL systems has limited performance when it extends to a certain scale. The performance of NoSQL is much more better than SQL nevertheless of writing or reading. When the data volume increase, the contribution of a flexible schema and scalability to the performance of the database is more obvious [TB11].

- **Consistency:**

  Some NoSQL system support the idea eventual consistency instead of fully consistency in RDBMS[Vog09]. This property does not guarantee that the user's read always returns the latest value after the update. In order to return the up-to-date value to all users, a number of conditions must be met. Some middle-ware appliances (such as CloudTPS for Google's BigTable and Amazons SimpleDB[ZPC11]) also exists, which are adding full ACID features to some NoSQL systems. In this type of database, it can determine which operation is governed by the ACID and which is by "Eventually consistency". But even so the NoSQL in terms of consistency is still not as strong as SQL, because SQL database has an absolute consistency which means it doesn't allow any inconsistent state exists. When choosing the type of database, users have to decide which is more important for them, a good consistency or good performance.

3.3.2 Data Model

In contrast to the standard SQL databases, there are a few types of supporting data model of the NoSQL databases. So there are several subtypes of NoSQL as following[TB11].

- **Document-style**: a document-style stores data recorder consist of a set of key-value pairs with a payload. Simple example are MongoDB[6] or OrientDB[7].
- **Key-value stores**: which stores key-payloads pairs. This style database is usually implemented by distributed hash tables. It is generally called key-value stores for simplicity. Simple key-value stores example are Apache Hadoop[8], Riak[9] or Amazon's Dynamo[10].
- **Wide column store**: this kind of database stores data tables as sections of columns rather than as rows. Some examples fall into this category are HBase[11],Cassandra[12] and Amazon SimpleDB[13].

There are probably two reasons for users to chose a NoSQL database as storage system: performance and flexibility. From the performance point of view, managing RDBMS' distributed data across several sites is very complex and the RDBMS multi-nodes structure is hard to maintain. Storing different types of data into a RDBMS has low flexibility because all the data stored in a RDBMS must be normalized to conform to a rigid relational schema. The user needs something with more flexible data structure.

[6]MongoDB: http://www.mongodb.org/
[7]OrientDB: http://www.orientdb.org/
[8]Apache Hadoop: http://hadoop.apache.org/
[9]Riak: http://basho.com/riak/
[10]Dynamo: http://aws.amazon.com/de/dynamodb/
[11]HBase: http://hbase.apache.org/
[12]Cassandra: http://cassandra.apache.org/
[13]SimpleDB: http://aws.amazon.com/cn/simpledb/

NoSQL database should be considered in the situations when a SQL database has reached its limitation or fulfills the resource-consuming task that induces the SQL system need to be distributed. NoSQL database with better horizontal scalability, can manage much larger amounts of data without hitting limitation of storage. For complex storage requirements, such as storing of unstructured data, video, audio or image files in the same area, some NoSQL databases support tree-shaped structures of the meta-data without a firmly defined data schema. Although there are a verity of files should be stored, the NoSQL databases can store them well and easily, because the data no longer have to be forced into tables and relations.

3.3.3 ACID vs. BASE

The ACID properties are the essentials rules for RDBMS, which guarantees RDBMS transaction works reliable. But they are also the obstacles of the distributed database system. Distributed database system cannot guarantee the full ACID properties, but support BASE instead of ACID[Pri08]. BASE is derived from the CAP theorem which is proposed by professor Eric Brewer in 2000 in University of California [14]. In 2002, Seth Gilbert and Nancy Lynch from MIT has proofed the correctness of this theorem[GL02]. The CAP theorem is now the basic foundation while describing a distributed database. The CAP explains the tradeoffs between the consistency, availability and partition

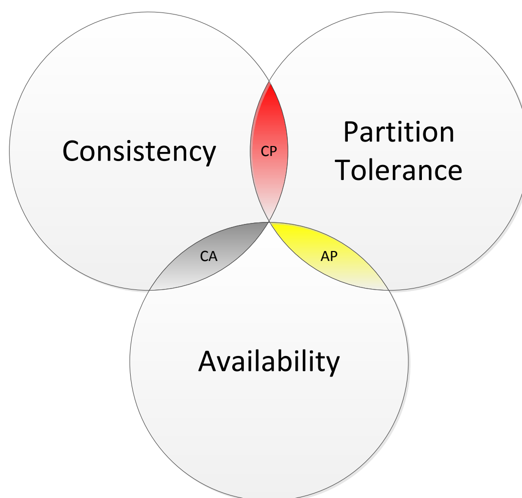

Figure 3.3: Different properties in NoSQL according to Brewer's theorem[GL02]

tolerance, and explores that a distributed system cannot have all of them simultaneously but only two of them can be selected at the same time[Bre00]. Figure 3.3 shows the CAP theorem, from which we can see that, the three circles has no common area.

- **Consistency**: means the data on all nodes of the distributed system is consistent. If one client has written something to an arbitrary node successfully, the changes

[14] http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

will simultaneously propagate to all the other nodes. The system is thus able to return readers the same result at the same time.

- **Availability**: guarantees that every request can receive a response from the server, even the request is failed. In practice, it is more meaningful to bounded the response in an appropriate time.
- **Partition Tolerance**: means the system will continue working even in case of messages lost between nodes or parts of the system is failed.

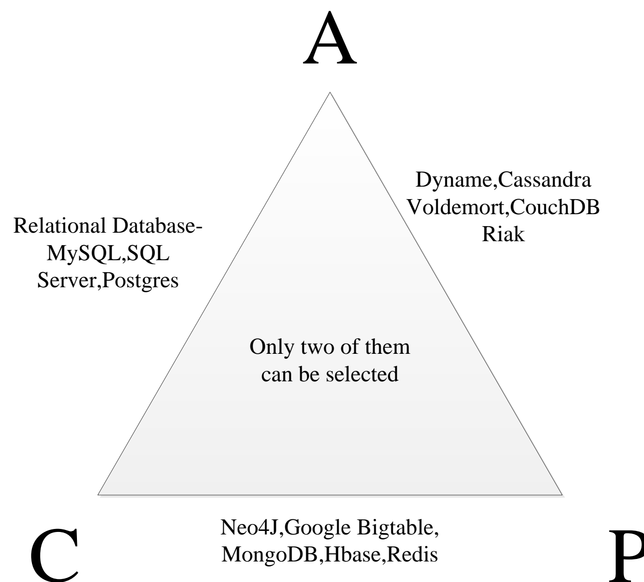

Figure 3.4: The popular databases products in different group of CAP adapted from [Hew10]

According to CAP theorem, we have the following choice when we design a system. Figure 3.4 shows the corresponding database products in different grouped classifications. All systems can only support at most two of these properties. Those three groups are three distinct combinations of CAP and each with different characteristics. The horizontal scaling strategy of a Cloud database is based on data partitioning, therefore, Cloud database is forced to choose one between consistency and availability[Pri08].

- **Group CA**: this group of system is primarily provides a consistent and available service. The relational database falls usually into this category, because RDBMS will first guarantee all nodes availability and consistency. The RDBMS normally run on a high available network and servers, thus it's not so necessary for them to deal with partition.

- **Group CP**: distributed systems with this combination usually have strongly consistency even in the presence of partition. However, the nodes in the partition may not be able to response to requests when they are waiting for the latest updating data. In this case, the system becomes unreachable. Hence, the availability cannot always be guaranteed in this group. Some example products are Google Bigtable, MongoDB, Habse etc.

- **Group AP**: has very high availability and partition tolerance. However, it cannot be always consistent. Some distributed systems, such as DNS(Domain Name System) and Cloud computing which need high availably, will take this combination. Cloud platforms rely on a horizontally scaled cluster and its workloads are also distributed in the whole system. Therefore, a Cloud application must have the tolerance to the failures. The example systems in this group are Cassandra, CouchDB, Riak etc.

As we introduced before, ACID in RDBMS make sure that the database has a consistent state after every transactions. For a distributed NoSQL database, it is enough to eventually be in a consistent state[Bar10]. So the BASE consistency model is widely used in NoSQL systems. BASE is not like ACID that forces consistency at the end of every transaction. BASE is optimistic and accept a eventually consistency which mean all nodes will reach to a consistent state after a reasonable time . It sounds a little bit hard to deal with but in reality it is quite feasible and leads a higher scalability that cannot be achieved by ACID.

- **B**asic **A**vailability: refers to the perceived availability of the data. It ensures that the whole system will stay operational in case of single node fails. This single failure will only impact on the users whose data is on the failed machine. A simple example about MMORPG is: the MMORPG players' data are partitioned across five database servers. If one of them is not reachable, only 20 percent of the users whose data is on that particular host cannot get their data and the other players will not be aware of this failer. So it remains the whole system higher perceived availability.

- **S**oft-state: indicates that the system will change state without users' intervention. The state of the data may be changed without inputting over time due to the eventual consistency model. A simple example can illustrate this point. In an MMORPG, the players can transfer their money or objects to other players. The operation will be decoupled into two, one is taking the asset from one player and another is giving the asset to the other player. There is a time lag between sending and receiving, in which the asset has left one player and not immediately received by another. However, from the players' perspective, this lag is invisible or certainly tolerable. We consider the system's behaviour is consistent and acceptable to the players.[Pri08]

	Cloud "Native"	Cloud capable
Relational	Amazon Relational Database Service (RDS), FathomDB, Microsoft SQL Azure	IBM DB2, IBM Informix Dynamic Server, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, Sybase, Vertica, ...
Non-Relational	Amazon SimpleDB, Google AppEngine Datastore	Hypertable, MongoDB, HBase, Apache CouchDB, Apache Cassandra, Project Voldemort, ...

Table 3.1: Classification of Cloud database products adapted from [MCO10]

- **E**ventual consistency: means all nodes of the system will be in a consistent state after a certain time. It is informally guarantees that, all the accesses to a item will return the latest value if there is no new update made to that data item[Vog09]. In contrast to ACID, BASE provides less consistency, so the client would possible receive the stale state. This is totally acceptable in a SNS system that a user has several seconds later to see his or her friend's status updating. In an MMORPG, part of the data also do not need strong consistency, for example, the existence of a tree in the map or the synchronization of a bird animation. Those informations are allowed to be different among the game clients.

3.4 Cloud Database Products

Lots of Cloud databases are provided for developers by database vendor. Those database alternatives can be categorised into three groups, "Cloud native" relational database, "Cloud native non-relational database" and " Cloud capable relations/non-relational database"[MCO10]. Cloud "native" relational database is a fully featured relational database running on Cloud while let the users free to worry about infrastructure management and database administration. Cloud "native" non-relational database enables a simple index and query functions without administration and with seamless scalability. Cloud capable relations/non-relational database is running on virtual machines with many operating systems and DBMS. The uses can install a new database server to avoid the provision of infrastructure and hardware and in this way the users would get complete control over the administrative and tuning tasks.[MCO10]. The examples of each classification are shown in Table 3.1.

As mentioned above, relational database has to subject to ACID constraints so it has very limited scalability and its performance is also not good with very large scale. To address our MMORPG big data volume and scalability problem, we intend to apply a NoSQL Cloud database. In next section, the foundation of NoSQL will be explained, and here we want to introduce some NoSQL products firstly. In NoSQL, there are different categories of databases exists.

Apache CouchDB is a document-oriented database. It offers incremental replication with conflict detection and resolution. MongeDB is also a document-oriented database, which provides a relatively complete function. It has the new generation of Mast/Slave Replication to ensure data distribution. Those two databases are not suitable for MMO-PRG game data storage due to document-oriented feature.

Redia is a key-value store and has perfect support for list and set. The data are stored in the memory so the writing and reading operations can be execute very fast. However it has very limited distributed feature[SULS09]. It is more suitable for BBS rather than our MMORPG.

HBase is an open-source, distributed and column-oriented store like Google's Bigtable. It provides Bigtable like capabilities on top of Hadoop[15].

Cassandra is a highly scalable distributed key-value store. It brings Dynamo's decentralized design and Bigtable's Column Family database model together [LM10]. Cassandra is open-source and it has a very active community.

After Comparison of those NoSQL database and according to some studies such as [TB11], we think Cassandra is the most suitable alternative for our testbed, In addition, paper [DSWM13] also recommended Cassandra as a underlying storage system of MMORPG. Because Cassandra is open-source, has highly scalability and decentralized structure, there is no single failure in Cassandra and its document and tutorials are also very enriching. Moreover, we could find some supports or advises from some active communities when we meet some problems since there are many experts are investigating Cassandra.

3.5 MMORPG Data Storage in Cloud Database

After the discussion about the differences between RDBMS and Cloud Database, we can find that, RDBMS are still the first solution for most of applications. Cloud database with NoSQL provides us better scalability, availability and the ability to handle big data volume with the sacrifice of consistency. Cloud database are more suitable for the web applications such as SNS which have little demand on consistency. However, in an MMORPG, the availability and scalability is somehow more important than consistency. So we choose the Cassandra as the backend database for the prototype. To adopt a Cloud database in MMORPG, the MMORPG data must be classified and organized so that part of the MMORPG related data can be more suitable to be stored in a Cloud database. There is a study has reached this problem[DSWM13]. The author has classified the data into four data sets and managed them regarding their requirements respectively.

- **Account data**: the user's account informations fall into this group, such as user's ID, password and account balance. This kind of data are used to identify a user and for billing purpose.

[15]Hbase http://hadoop.apache.org/hbase/

Data	Consistency	Performance	Availability	Scalability	Partitioning	Flexible model	Simplified processing	Security
Account Data	★★★	★★	★★★	★	★★	★	★	★★★
Game Data	★	★	★★★	★	★★	★	★★	★
State Data	★★★	★★★	★★★	★★★	★★★	★★★	★★★	★★
Log Data	★★	★★	★	★★★	★★	★	★★★	★★★

Table 3.2: Data classification and requirements analysis adapter from [DSWM13]

- **Game data**: includes the basic data for supporting the virtual game world such as game map information, NPC locations, meta-data(name, race, appearance and etc.), system logs and game rules. Those informations are generated and managed by game developers and will not be frequently modified.
- **State data**: represents the character's informations of a player. The player's skills, inventory and attributes are modified constantly. This group of data is stored in an in-memory database and persisted to disk periodically.
- **Log data**: contains the user's chat and operation logs. Log data are recorded for analysing purpose, such as user's behaviours in the game and essence statistics to evaluate the game. This kind of data is important for game vendor to improve the game and know their customers.

The author summarized the requirements for consistency, performance, availability, scalability, partitioning, flexible model, simplified processing and security of the four groups data in Table 3.2. From the table we can see that, the account data have high requirement on consistency, availability, security and do not need too much scalability, hence in order to protect this kind of data and ensure the transaction operations on account data, the author purpose to persist this group of data in a RDBMS. The Log data need to be continually appended and stored in database for a long period of time, the data volume will be extremely large. Thus it will be stored cross many nodes and be protected securely. The game data are not be modified regularly but be frequently accessed, so they need highly availability and must be partitioned. State data are modified constantly by large number of concurrent players. Those data should be processed in real-time and persisted on disk efficiently. With the development and improvement of a MMORPG, the state date change and enrich a lot, hence a flexible model is important for lower the developing cost and complexity. They should also be stored partitioned for load balancing and improving the availability. Hence the state data have highly requirements on most of the aspects mentioned above. This idea provides the basic storage concept and inspires the prototype. In my thesis, the author's propose with Cloud database management for MMORPG will be implemented and the prototype will be tested.

4. Cassandra

4.1 History and Development

Cassandra is an open-source distributed key-value store which is capable to handle large data volume across many commodity servers. It was initially developed by Facebook[1] for its Inbox Search feature and in March 2009 it became an Apache project. Cassandra has adopted the column family data model from Google BigTable[CDG+08] and the decentralized architecture of Amazon Dynamo[DHJ+07]. It brings the advantages of both databases and provides high availability with no single point of failure[LM10]. Cassandra was designed to handle data spread across multiple geographical regions. The development of Cassandra is very fast. Many changes have been made, and more and more advanced features have been added to the new version since Cassandra was released. It has an active community and lots of users. Cassandra is now applied to commercial products by a lot of companies, such as Digg, Reddit and Twitter [HHLD11].

4.2 Cassandra Data Model

In this chapter, we will give an introduction of the data model that Cassandra have adopted. Some new terms in Cassandra in associate with data model will be also explained. Cassandra data model is schema-optional and column-oriented. Schema-optional means that the users can customize the data model according to data storage and access requirement. The following are the basic data model related terminologies in Cassandra.

Column

A *column* is the smallest unit in the Cassandra world, which consists of "name", "value " and "timestamp" as Figure 4.1 on the next page shows. For clarity of this structure,

[1] www.facebook.com

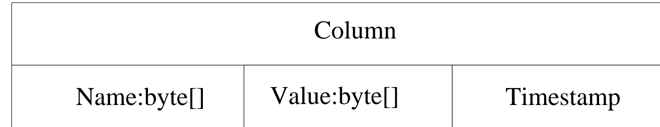

Figure 4.1: The structure of a column

here is an example of a column represented with JSON[2] notation. {
"name" : "email"
"value" : "sample@gmail.com"
"timestamp" : 1368472612939
}
This example shows the column has a name "email" and its value is an email address. Timestamp is used for conflict resolution on the server side. It is a Java long type which records the last time the column was updated. The timestamp is usually provided by clients along with the value when they trying to write[Hew10].

Row

In order to group the column values together, we need a container called *Row*. A *rowkey* in a row is used to uniquely identify this group of columns. However, in Cassandra, it is not forced to have same number of columns in every row. In other words, the row may have different set of columns as Figure 4.2 shows. A single row doesn't have a

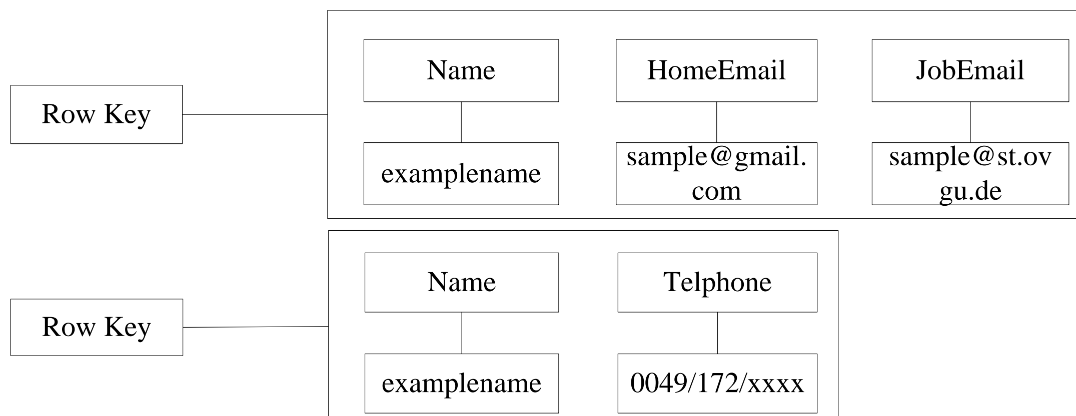

Figure 4.2: The structure of a row

timestamp, only each columns contained in a row has its individual timestamp. The columns stored in a row has a specific order, which will be explained later.

[2]JavaScript Object Notation

Column Family

In Cassandra, we define *column family* to be a logical view, which group the similar data together. The column family is somewhat analogous to a table in a RDBMS. For example, we might have a player column family, an inventory column family, a chat log column family and so on. A column family is a container which holds an ordered collection of rows. Every row in a column family holds a collection of ordered name-value pairs. We can use a JSON-like notation to describe a game user column family as following shows:

Player{
rowkey:player001 { name:player1, race:human, level:12, email:sample@email.com, address:sample }
rowkey:player002 { name:player2, race:night elf, level:23, birth:10/1985 }
rowkey:player003 { name:player3, race:human , level:33 }
rowkey:player004 { name:player4, race:Orc , level:18 }
}

While we define a column family, we can specify how column names are sorted in a row and the results returned to the client are in exactly same order. The following types are support by Cassandra column family for sorting the column name: *AsciiType, BytesType, LexicalUUIDType, IntegerType, LongType, TimeUUIDType* or *UTF8Type*[Hew10]. Figure 4.3 gives us a more intuitive view of how data stored in a column family.

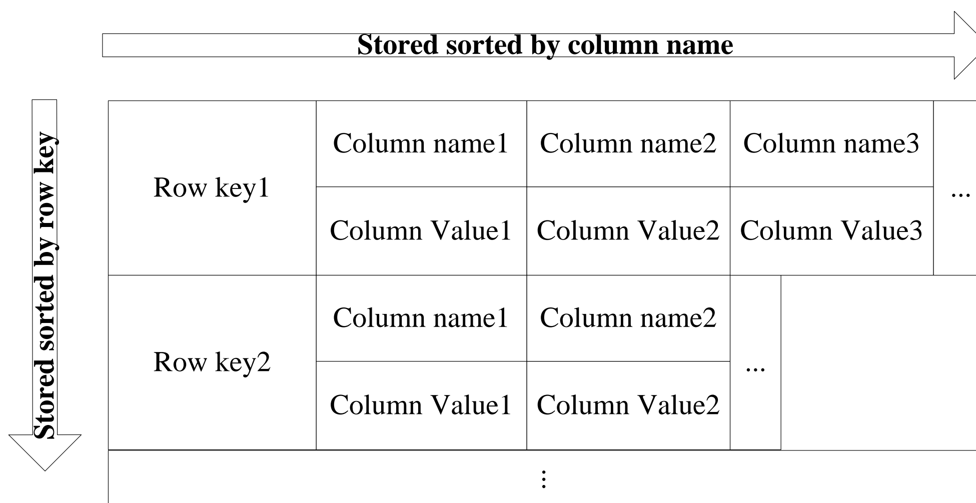

Figure 4.3: Cassandra column family stores sorted rows and columns

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

Table 4.1: Relational model analogy

KeySpace

Keyspace is the top level container that used to group column families together. It is similar to a schema in a RDBMS. Typically there is only one keyspace for each application in a cluster. When we define a keyspace, a "replication factor" must be specified. A "replications factor" indicates that how many replicas this keyspace will be copied on. Hence, data that have different replication requirements should be stored in different keyspace.

Cluster

From a logic view, cluster is the outermost container for keyspaces. A Cluster can have a variety of keyspaces. From a physical view, cluster is consisted of a number of nodes, each of which holds a subset of the whole data. Cassandra Cluster is decentralized, which means those notes are identical. None of them have a special task such as performing organizing operations.

The terms of Cassandra are not totally comparable with the relational world. In order to clarify the terms more simply and clearly for the newcomers, a relational model analogy is shown in Table 4.1. This table gives a first impression of the transition from a RDBMS to Cassandra, but the design concepts and underlying technologies are completely different.

4.2.1 Cassandra Ring

Cassandra supports elastic scalability, which means a Cassandra cluster can seamlessly scale out and scale down[Fea10]. This feature is particularly desired by an MMORPG storage. When the number of players goes up, Cassandea scales the system by adding new nodes; when it comes down, Cassandra removes some nodes for saving cost. As we introduced above, a cluster is consist of a number of nodes, each of which holds partial data. The nodes in Cassandra are arranged in a logical ring. Cassandra assigns a range of data to a particular node by calculating tokens. Each node is responsible for the range of itself (inclusive) and the predecessor (exclusive)[Dat13f]. When there is a datum need to be stored, the system will calculate the row key of this datum to a value of token and store this datum to the node which includes this token in its assigned token range. The copies of datum in Cassandra are called "replicas" and

normally how many replicas should be stored is determined by "replication factor". There are many strategies available for replicas storing, such as simple strategy and network topology strategy. When a user creates a key space, she/he has to specify the replica placement strategy. If the strategy is not specified, the simple strategy will be taken as default[Dat13e]. When the users sets a relatively higher replication factor, the data will be consequently replicated to the other nodes. Every node stores partial data from the other nodes to ensure the availability of the database in case of failures. Assuming that we have a five nodes cluster as Figure 4.4 shows,

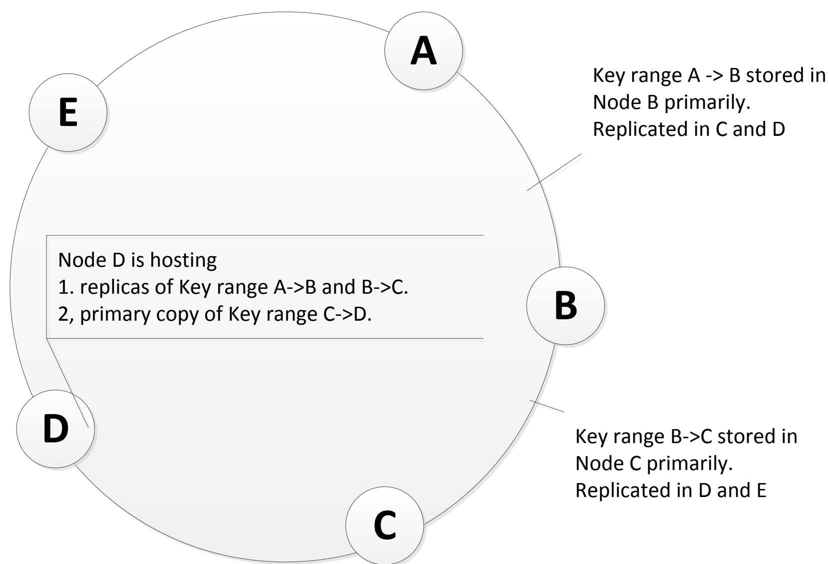

Figure 4.4: Cassandra cluster ring adapted from [Dat13e]

if we set the parameter "replication factor" as 3, then all the data will have 3 replicas in the cluster and every node stores its own primary key range and also holds the replicas of its next node clockwise in the ring. Such as Node D, it is hosting not only primary copy of key Range from C to D and also the replicas of key ranges from B to C and A to B. Key range from A to B is stored primarily on node B and it will be duplicated on node C and D. When a node B crashes, the clients can also get response from C and D. The data which B owned will be also transferred to C. The more detailed illustration can be seen from Table 4.2 on the following page. If we want to scale this cluster, we just need to add a new node into the cluster, and the system will automatically rebalance the data and sending the new node to work.

Node	Primary copy	Next node	replicas "clockwise in the ring"
A	E→A	B	C→D, D→E
B	A→B	C	D→E, E→A
C	B→C	D	E→A, A→B
D	C→D	E	A→B, B→C
E	D→E	A	B→C, C→D

Table 4.2: Replicas table

4.3 Transaction and Concurrency Control

Cassandra, like the other NoSQL databases, doesn't support fully ACID properties and the transactions. But it does offer the atomicity and isolation at the row-level[Dat13a].

4.3.1 Atomicity

Cassandra supports atomicity at the row-level, that means the new inserted or updating columns in a row are performed in one atomic operation[Dat13a]. Cassandra doesn't perform multi-row update transactions as a "one or nothing" operation like RDBMS does. When a write are not successful on all nodes, Cassandra will not roll back but report this to the client. This write operation will be still actually stored to a replica.

4.3.2 Tunable Consistency

Cassandra supports tunable consistency, which means during a querying a querying users can specify the level of consistency to trade for availability. Consistency in Cassandra refers to how to synchronize a row of data to all replicas[Dat13b]. It provides users a chance to make trade-off between consistency and availability. The higher the consistency level for one operation is specified, the more nodes need to respond to the operation. Cassandra uses timestamps to determine the up-to-date value in a column. Timestamp is provided by client-side. If two or more nodes contain values with different timestamps, then the value with highest timestamp will be returned to the client. In the back-end, Cassandra performs a "read repair", to be aware of one or more nodes stored a stale data which were possibly read by a client and then the system will update those nodes with the current version so that all nodes are consistent. There are several possible consistency levels for both read and write operations[Hew10].

Read Level Consistency

- **ONE**: Level One means the Cassandra returns the record read from the first responded node. In the meanwhile, Cassandra will perform the "read repair" if there are any stale data exists.
- **QUORUM** : QUORUM means query all nodes and waiting for the majority ((replication factor/2)+1)of replicas to response. The value with the most recent timestamp will be returned and if necessary "read repair" will be performed.
- **ALL**: All of the replicas in the cluster will be queried and the system will wait for all of them to respond. The latest value among all nodes will be returned. If there is any node unreachable, the read operation is consider as failed.

Write Level Consistency

For a write operation, the consistency level means how many replicas must accept the write request before a client get a response from the Cassandra[Dat13b]. The level ANY has the highest availability by sacrificing consistency, and ALL has the highest consistency but the lowest availability. The following consistency levels are available for write in Cassandra.

- **ANY**: In this case, a write operation must be successfully performed on a minimum of one node. If all nodes in the cluster are not reachable for the given rowkey, then a "hinted handoff" will be written, this write is still considered as successful [Dat13b].
- **ONE**: A write must be successfully written to both commit log and memory table of at least one node.
- **QUORUM**: A write must be written to both commit log and memory table of a quorum ((replications factor)/2+1) of replicas.
- **ALL**: ensures the write operation is performed on all replicas before the client get a response.

4.3.3 Isolation

Cassandra are now supporting "row-level isolation", which means changes to a row is invisible to any other user until the write operations are completed[Dat13g].

4.3.4 Durability

Cassandra writes always both in memory and in a commit log on the hard disk. In case of crash or server failure, the commit log is used to recover any lost writes on restart. So the write in Cassandra are durable.

4.3.5 Read and Write

A client's read and write request can go to arbitrary node in the cluster. In the cluster, one single node stores only a subset of whole data. However, from the user's perspective, all nodes are identical. Every node in Cassandra should be able to respond to the user's request. When a client connected with a node, then the node acts as a coordinator for this client's operation. The coordinator knows the data distribution and will help the client get the data from the right node. For a write operation, the coordinator sends the request to all nodes that stored the row being written, and all the living and available nodes will get the write regardless the consistency level specified by the clients. However, the consistency level determines how many nodes should response in order to consider this write as successful. A simple example shows in Figure 4.5 on the next page. A

cluster with six nodes has the replication factor three. The client sends a write to the cluster and sets the consistency level one. In this case, the request will be send to node A, F and C that stores the rows should be written, as long as one node responds to the coordinator, the coordinator considers this write as successful and sends an positive response to the client.

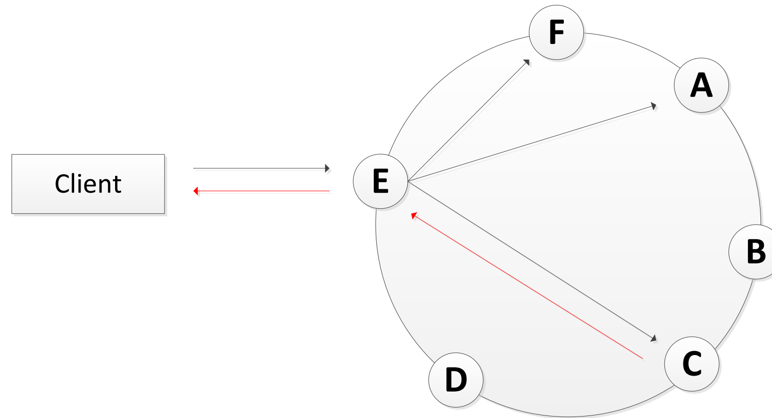

Figure 4.5: Cassandra client write request[Dat13e]

When a client sends a read request to the cluster, the coordinator forwards the request to replicas, the number of which is determined by the consistency level specified by client. The coordinator will send this request to the other replicas that did not receive the current request for background read repair. If the coordinator gets multi requested data, then those data will be compared if they are consistent. If not, the coordinator will send the most recent data with the highest timestamp back to client. The stale data on other nodes will be updated to the newest version by “read repair” [Dat13a]. For example, we have a six notes cluster with replication factor of 4 as shown in Figure 4.6 on the facing page Table 3.1 on page 22.

The client send a read request with consistency level one. Then the coordinator E will get the request and forward it to only one (which refers to node C in our example) of the four nodes which is responding relatively quicker. Then the client will receive the data from node C. At the meantime, the coordinator will also sent the read request to the other nodes (which are A, F, B in our example) which stored the rests of the replicas. The coordinator will get all the responses from these four nodes (A, B, C, F) and compare them. If they are not consistent, the coordinator will overwrite the stale data with the latest data. This is the so called “background read repair”.

4.4 Query Tools in Cassandra

Although Cassandra is a newborn Cloud database, there are lots of client’s API libraries available for different programming languages. Those API libraries are created by the

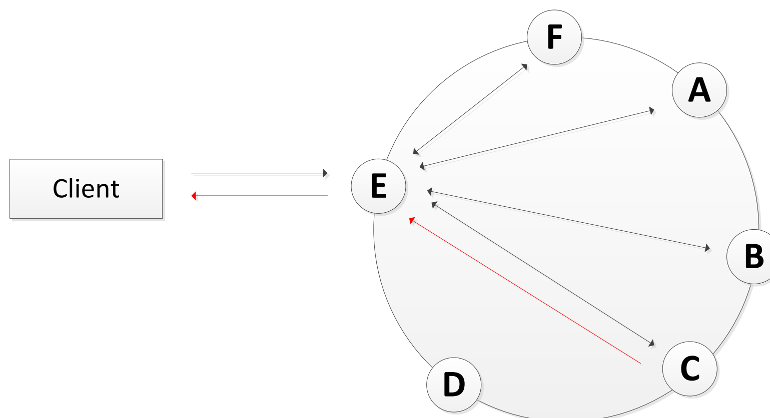

Figure 4.6: Cassandra client read request[Dat13e]

engaged programmers in order to help the client side users to interact with Cassandra more efficiently. Those APIs can also support more useful features that are helpful to accelerate programming process. The are normally two ways to get access to Cassandra: one is Cassandra Query Language (CQL)[3] client and another is Command Line Interface (CLI)[4] clients. Programmers could choose different client APIs which fit their needs to implement a specific application running on top of Cassandra for different programming languages.

CQL

CQL was firstly introduced with the Cassandra release version 0.8. It is an interactive command line interface for Cassandra[Dat13c] and can be found in the installed Cassandra folder. This client API is now highly recommended by the official Cassandra community. It provides user a SQL-similar syntax, which is mapped to Cassandra concepts and operations. To create a keyspace by using CQL, we can use the following syntax.

```
CREATE KEYSPACE JMMORPG
WITH strategy_class='SimpleStrategy'
and strategy_options:replications_factor='1';
```

Listing 4.1: Using CQL creating keyspace sample

With the syntax in Listing 4.1, we can create a new keyspace in Cassandra cluster. While we create a keyspace, we can also define the "replication factor" and the "strategy" for that keyspace at the same time. In Listing 4.2 on the next page shows how we can use CQL to create a column family for players' storage.

[3]Client CQL :http://www.datastax.com/docs/1.0/dml/using_cql
[4]Client CLI :http://www.datastax.com/docs/0.8/dml/using_cli

```
CREATE COLUMNFAMILY gamePlayer (
   rowkey varchar PRIMARY KEY,
   usename varchar,
   email varchar,
   session_token varchar,
   state varchar,
   birth_year bigint);
```

Listing 4.2: Using CQL creating column family sample

After the column family was created in a keyspace for the JMMORPG, we can insert some values into this column family. The syntax is very similar like what we did with SQL, as Listing 4.3 shows.

```
INSERT INTO gamePlayer (rowkey, password)
VALUES ('player1', 'sample@gamil.com');

SELECT * FROM gamePlayer WHERE rowkey='player1';
```

Listing 4.3: CQL insert and select sample

From the samples above we can find that CQL is aimed to be similar as SQL. This makes CQL very easy to learn and to use. In another word, programmers who are familiar with SQL can simply work with CQL in Cassandra. Moreover the official community of Cassandra also recommends that new applications should be developed with CQL instead of raw Thrift.

CLI

The Cassandra CLI client utility is a low-level Thrift RPC-based command line interface. It supports basic data definition language(DDL) and data manipulation language(DML) for a Cassandra cluster. CLI is released along with every Cassandra version and is the primary tool to interact with Cassandra before version 0.8[Dat13c]. When a Cassandra instance was installed, the "cassandra-cli" tool can be found in the bin folder. A Cassandra user can use CLI to connect to remote nodes in the cluster and to create/update schema or retrieve data. The first thing needed to be done to start using Cassandra is to create a keyspace for our MMORPG. The following syntaxes are available for doing that.

Because raw Thrift is a low-level API, it is inefficient in the programming environment. The programmer can also choose a high-level client API for their language. Lots of high-level APIs with convenient features are available such as Hector[5] for Java, Aquiles[6]

[5]Hector API :http://hector-client.github.io/hector/build/html/index.html
[6].NET Clients API :http://aquiles.codeplex.com/

for .Net and Fauna[7] for Ruby. The basic permissible operations on database are the following APIs.

- insert(table, key, rowMutation): specify the name of a distributed column family and the unique rowkey of the row which need to be inserted.
- get(table, key, column): specify the name of the column family and the unique rowkey of the row which will be queried along with the column name.
- delete(table, key, column): a name of the column family, the rowkey of the row which should be deleted and the column name.

Those three operations are the most basic fundamental and essential APIs which are supported by Cassandra. In practice, some more advanced tools are also available. The basic CLI syntax based on the APIs are a little bit different from CQL. In the MMORPG prototype, Listing 4.4 shows the syntax that is used to create a JMMORPG keyspace and to change the current word context to the created keyspace.

```
    CREATE KEYSPACE JMMORPG
    with placement_strategy = 'SimpleStrategy'
    and strategy_options = [{replication_factor:1}];

    use JMMORPG;
```

Listing 4.4: CLI creating a keyspace

This works exactly the same as CQL. The Listing 4.5 shows how to use CLI to create a column family.

```
CREATE COLUMN FAMILY gamePlayer
    WITH comparator = UTF8Type
    AND key_validation_class=UTF8Type
    AND column_metadata = [
{column_name: full_name, validation_class: UTF8Type}
{column_name: email, validation_class: UTF8Type}
{column_name: state, validation_class: UTF8Type}
{column_name: gender, validation_class: UTF8Type}
{column_name: birth_year, validation_class: LongType}
];
```

Listing 4.5: CLI creating a column family

We have a keyspace and a column family now. Then we can use the syntax shown in Listing 4.6 on the next page to insert some rows. With CLI we can only set one column at a time with a SET command.

[7]Ruby Clients API :https://github.com/twitter/cassandra

```
SET gamePlayer['player0001']['full_name']='Shuo␣Wang';
SET gamePlayer['player0001']['email']='shuo2.wang@st.ovgu.de';
SET gamePlayer['player0001']['state']='MD';
SET gamePlayer['player0001']['gender']='M';
SET gamePlayer['player0001']['birth_year']='1985';
```

Listing 4.6: CLI set users information

Once the keys and values are inserted, the GET command in 4.7 can be used to retrieve a particular row from a column family. LIST command can return a batch of rows and their associated columns.

```
GET users ['player0001'];
LIST users;
```

Listing 4.7: CLI get uses information

5. Design and Implementation

5.1 Starting Point

For the purpose of studying a Cloud-based MMORPG as well as evaluating its robustness and performance, we will design and implement a game platform that can be used to evaluate typical aspects of MMORPG. This platform consists of a client side and a server side. The server side should have a simplified game logic, such as receiving the commands from clients and sending game states (e.g. players' movements, chatting and trades) back to clients. For the prospect of this platform, the future work should be concentrated on the extension ability for furthering functionality, for instance, battle, player interact with non-player characters and so on.

5.1.1 Infrastructure

Hardware/Resources

For running this prototype test environment, we got 8 virtual machines with Ubuntu operating system from our university. Each virtual machine has a 2266MHZ CPU, 8 GB memory and 91GB hard disk. For the security reasons, those virtual machines cannot be accessed outside directly. We have to first use Security Shell (SSH) protocol to connect a stepping stone server and then get access to the virtual machines via it indirectly. Those virtual machines will be used to deploy Cassandra cluster and game servers. The detail Infrastructure of our prototype show in Figure 5.1 on the next page

Software/programming language

In this paragraph we will briefly introduce the technologies and software we are using for developing the test environment. As we analysed above, Cassandra is the most suitable open source Cloud database for the Cloud-based MMORPG due to its high

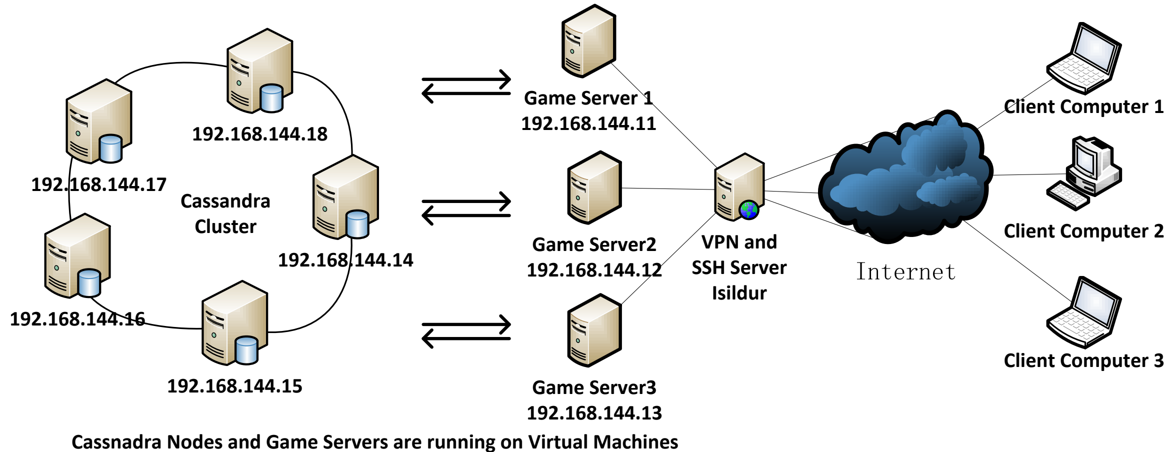

Figure 5.1: The infrastructure of the prototype

performance and scalability. The latest version of Cassandra while I am writing this chapter is 1.2.5[1]. Cassandra is written in Java, which is one of the most popular programming languages in the world and has the crossing-platform ability. For running the instance of Cassandra, the Java Development Kit (JDK) 1.6 is needed. Because of this reason, Cassandra has the most compatibility with Java environment and most of the existing relative mature tools are supported in form of Java libraries. With those libraries we can accelerate the development process. So we are also using Java[2] as programming language for implementing this test environment. As we know, a reliable network infrastructure is important for an MMORPG. In our prototype, we are using Darkstar[3] to realize the communication between client and server. Darkstar is an Apache project which is specifically designed for MMO game network architecture. we will give further introduction about Darkstar later.

5.1.2 Functional Requirements

The main purpose of this thesis is implementing a prototype to simulate an MMORPG running on a Cloud database. The prototype should comprise the following functions:

- A simple architecture of game's server side and client side, where a potentially high number of clients and game servers can be started.
- All of the game data should be stored in a Cloud database, which is Cassandra here .

[1]http://cassandra.apache.org/
[2]http://www.oracle.com/technetwork/java/javase/downloads/index.html
[3]DarkStar Website http://sourceforge.net/apps/trac/reddwarf/

- Game server side should contain a simplified game logic that implements simple players' movements on a map and interactions.
- Game client side should have a GUI(graphical user interface) which is capable to connect and communicate with game server. The user can use this client to interact with other users and also receive the state changes sent by game server.
- The test environment should have the ability to start a specified number of clients and servers. Those servers are used to evaluate the performance of Cassandra in MMORPG.

5.2 Prototype Architecture

According our analysis and the available hardware resource, we decide to use the architecture which is described in Figure 5.2 for the testbed.

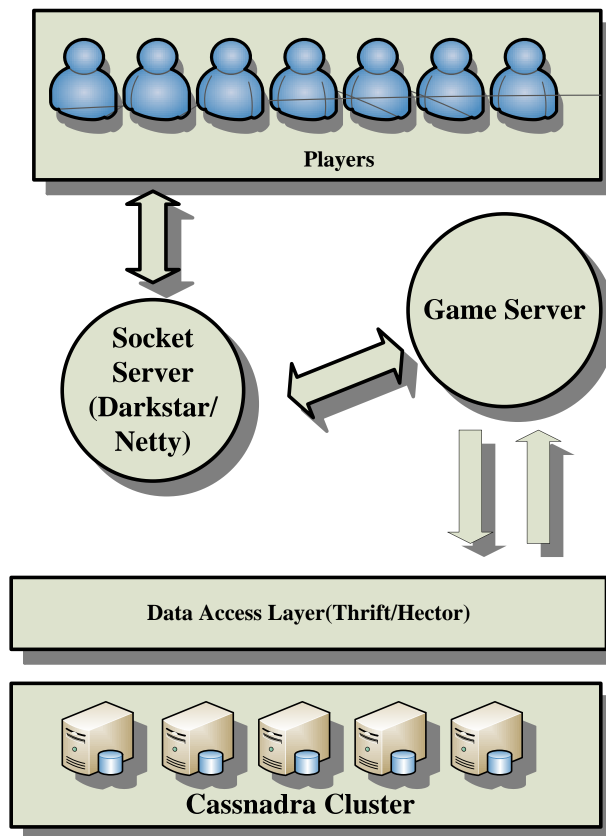

Figure 5.2: The test platform architecture

The client and server communicate via a socket server. The socket server is supported by a project Darkstar. Later we will briefly introduce the project Darkstar. The

game server stores game logic and manages game state data when the game is running. The database layer deals with the basic data accessing operations. It is responsible for data querying and inserting in all underlying column family which hosts the game player information, inventory information, user logging, user statistics and game world information in Cassandra cluster. A game player connects to the Darkstar Server by giving their credential information. When the user login is successful, the user can then create or choose a hero to play the game. Darkstar Server is response for calculating the world state and user state in associate with commands sending by players.

5.3 MMORPG Database Schema Design in Cassandra

As we mentioned in Section 3.3 on page 15 and Chapter 4 on page 25, The design concepts between Cassandra and RDBMS are different. In order to make the reader a clearer understanding of game database schema design in Cassandra, we organize this chapter like this: Firstly, we will map all game entities to a RDBMS so that the reader can have a intuitive understanding of game database; Secondly, we will explain the different design concepts between Cassandra and RDBMS; Finally, We will present our solution in Cassandra.

5.3.1 Tranditional Game Database Schema

If we are building a new data-driven application running on top of a RDBMS, we should firstly analyze the domain, then model it as some normalized tables, and at last set foreign keys to refer to related data in other tables. Here we will use the game entities required in our prototype as an example to introduce the game database schema in RDBMS. The basic basic entities as well as table design in a relational model is as follows:

- **Users**: This table stores the basic information of users. Actually, in order to achieve a high consistency and security, this part of data should be stored in a RDBMS as we analyzed in Chapter 3 on page 13. To reduce the workloads of the account server, user data used during the game will be replicated in Cassandra. But the account related data for billing purpose will not present in Cassandra.

  User[<u>userId</u>[4], userName, password ,Gender, Email, lastAccess, registerDate]

- **Hero**: This table stores the characters' information created by users. Normally, a user in an MMORPG can create more than one character. For this reason, the user table and hero table is one to many relationship. For example, when a user logs out of a game, the position information (map and coordinates) of her/his hero must be recorded to the hero table. In this way, the user can get the last position information of her/his hero and continue the game, when she/he logs

[4]Underline means it is a primary key

in to the game again. In addition, the heros' basic attribute such as attack or defence level should also be stored in this table.

Hero[heroId, userId[5], heroName, level, raceId, hp[6], hpMax, magic, magicMax, attack, defence, mapId, positionX, positionY, money]

- **Inventory**: This table stores user's private items gained during the game. user's sword, armour and some other items will be persisted here. It refers to the hero table and item table. One user can have more than one hero, and each hero has its own inventory.

  Inventory[inventoryId, heroId, itemId]

- **Logs**: For monitoring the user's behaviour, part of the user's operations will be recorded and stored in this table. This table can be very large over time and hence it becomes hard to maintain. It could consist of a primary key logId, a userId to identify the heroes, and the action associated with the performed time.

  Logs[logId, userId, action, time]

- **Item**: It stores the items which are defined by developers in the game world, such as a sword, a armour and so on. For simplify the structure, we assume that all the item types are in standard with the basic attribute, i.e. no specific item with additional attributes such as attack plus exist.

  Item[itemId, itemName, price, description]

- **Maps**: The game's map information goes into this table. While a user is trying to login the game, the system will firstly figure out which map her/his hero was in and send those information to the client. The client will load the map and present the hero in the previous position. Of cause, the user who logs into the game for the first time, will have a uniformed predefined map and location.

  Maps[mapId, mapname, size, path]

- **NPCs**: This table contains all the NPC(Non-player character) information in the game world. This table will be typically loaded firstly when the game is started. Unless there is an update for the game which needs to add new NPCs, the content of this table will remain relatively stable.

  NPCs[npcId, mapId, type, description]

- **Skills**: The hero of a user will always learn new skills, and all heros' skills will be stored here. Every hero can have different skills and some skills are only for certain race adaptable. There is no skill design in our prototype, so this table exists only for the extension purpose.

  Skills[skillId, skillName, raceid, description]

[5] Unterwave means it is a foreign key
[6] Hit Point

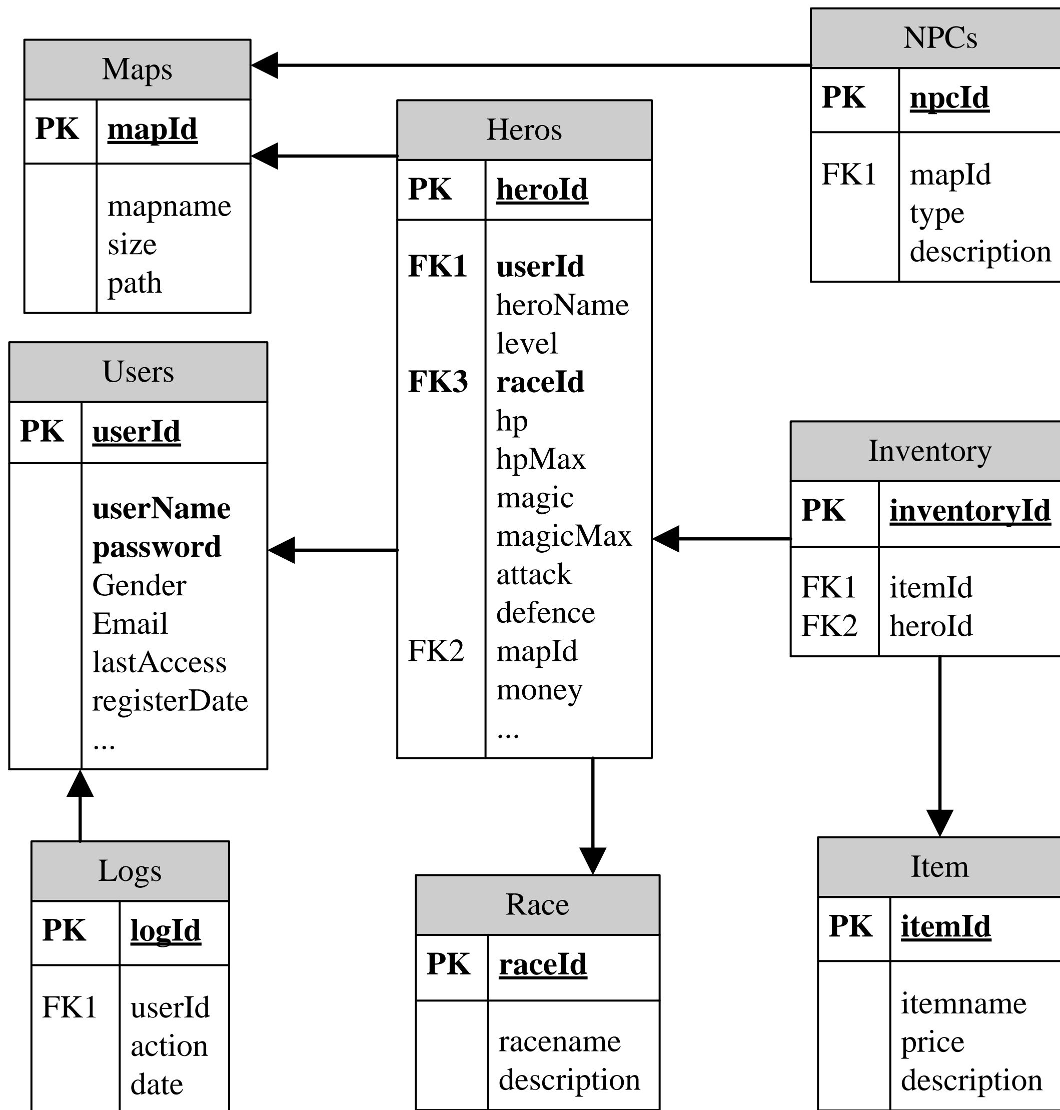

Figure 5.3: E-R model design of prototype

With those information, we can create a relational data model in traditional way. In Figure 5.3 on the facing page shows a E-R model according the above analysis. It gives us an intuitive view about how this system's data is constructed. Next we will illustrate the different design concepts in Cassandra and give a column family design to show how the data is constructed in the NoSQL Database.

5.3.2 Different Design Concepts

Cassandra is a new Cloud database and it has total different data structure from RDBMS. We have discussed a lot about the comparations between RDBMS and Cassandra, and now we will illustrate the differences in practice designing by using Cassandra with the example of our prototype. The following are some important different design concepts between Cassandra and RDBMS[Hew10].

No Query Language

The relational database uses a standard SQL as query language. Cassandra stores key-value pair and has no query language. But Cassandra has an API so that we can access through Remote Procedure Call (RPC) serialization mechanism. Thrift is the basic API for Cassandra. There is also higher level API such as Hector [7] that provides more useful functions We will discuss it later.

No Referential Integrity

Normally there should be a relation between Tables in RDBMS. We could specify a foreign key in one table to reference the primary key in another table. Cassandra has no such concept of referential integrity, hence there is also no join in Cassandra. However, it is still a common design requirement to store a unique ID to relate to other entities in other column families, but the cascading deletes are in this situation not available [Hew10].

Secondary Indexes

Secondary index is a feature in Cassandra. Assume that we want to find a user's ID by specifying her/his email. In a RDBMS we can use a query in traditional way like in Listing 5.1 shows:

```
SELECT playerID FROM gmaePlayers WHERE email='example1@gmail.com';
```

Listing 5.1: SQL syntax sample

If we only know the user's email and we want to find this user's ID, this query will be helpful. The RDBMS handles a query like this by performing a full table scan

[7] http://hector-client.github.io/hector/build/html/index.html

and inspecting each row to find the value we are looking for[Hew10]. This will be very slow if the RDBMS is doing a scan on a very large table. Hence, the solution for this problem in RDBMS is to create an index on "Email" column. The RDBMS can look up very quickly with an indexing column. Because the primary unique key "userID" is automatically indexed, to create another index on the email column would be considered as a secondary index. Cassandra has the similar solution with RDBMS, which is secondary index. The early version of Cassandra doesn't support secondary index and users must manually maintain a copy of data in a column family as index[8]. Now this feature is used as a standard feature and is integrated in the latest release of Cassandra.

Sorting is a design concept

We can easily change the order of returned values in RDBMS by specify "ORDER BY" in SQL query. If we don't specify an "ORDER BY", the default order is the order in which they are written. In Cassandra, the order of a column is a very important design decision. Column family's definitions have a "CompareWith" parameter. This parameter can have different types (ASCII, Long, Integer, TimestampUUID) as we introduced in Chapter 4 on page 25. This parameter determines the order in which rows will be sorted on reads and it cannot be changed during query.

Denormalization

In a relational world, the normalization is a very important design concept. With normalization we can reduce the redundancy by doing join with foreign key. Denormalization is against Codd's normal forms and it should not happen. Unlike in relational database, denormalization is quite normal in Cassandra. In contrast to RDBMS, Cassandra has the best performance when the data is denormalized.

5.3.3 Schema Transformation

So far we have introduced data storage mechanism as well as design concepts of Cassandra and the requirements of our prototype. As we have introduced the characteristics of Cassandra storage are in Chapter 4 on page 25. In Cassandra, all data are stored in ordered columns and the columns constitute a row which is uniquely identified by its rowkey. The column family of Cassandra is similar to a table of a relational database that contains columns and rows. However, there is no fixed set of columns like a table in Cassandra. Each row of a column family could have different set of columns. We can say that, the row is a container for columns. All columns are sorted in different types. Now we can present the column family design of our prototype as in Figure 5.4 on the facing page shows. From this Figure we can see that, we have designed five column families to store all the data analyzed above. Before we explain why those data looks like this construct, we want to firstly illustrate how data in Cassandra can be queried.

[8]http://wiki.apache.org/cassandra/SecondaryIndexes

Cassandra doesn't support powerful query language and cannot do join operations. There are only few query operations supported by Cassandra. It is very important for Cassandra data model design: starting with queries. We should first determine what queries do our application need and think about query patterns up front, and then organize the data around the queries, in the end design column families accordingly. But this doesn't mean the entities and relations are not needed any more. Relation and entities are the tools to help us to perceive the real world. It is still important to understand the entities and relations in a domain and then model data according the query pattern by denormalizing and duplicating[Eba13]. That is the reason we have illustrated the E-R model about our MMORPG design above. Here we will introduce the query pattern in our prototype, we are likely have the following queries.

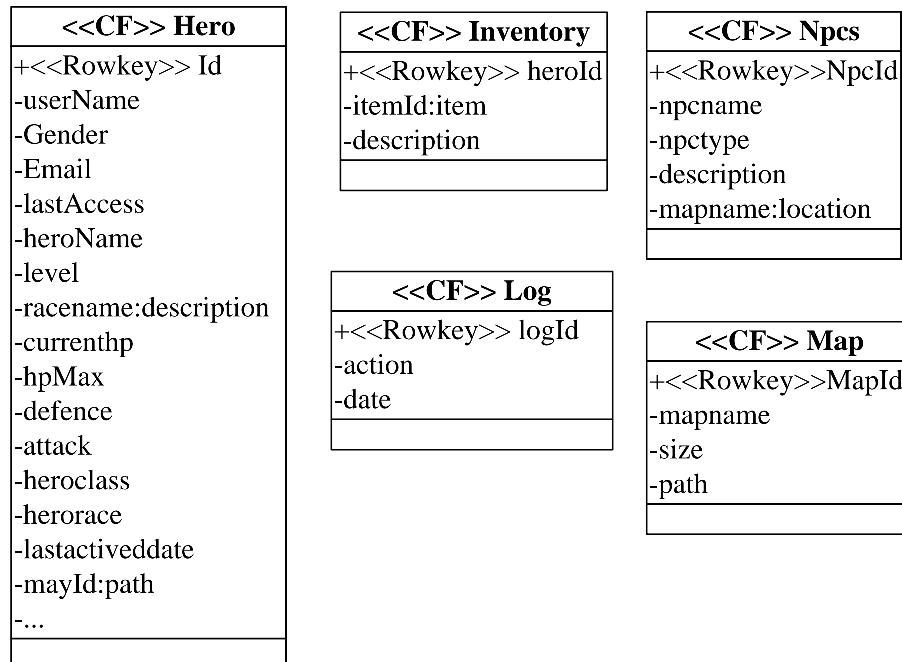

Figure 5.4: Cassandra column family design derived from E-R model

- Get all map: At the very beginning of a game, all the map information should be load into memory once. After this, the user can login into the game world and present at the different maps. This query is a very simple "Rangeslicequery". Scan the whole column family "Map" and load all map information into memory for later usage.

- Get all items of a hero: While the user is playing a game, he will check out her/his inventory and see how many items they have. This query is used to scan the "Inventory" column family by a specified "playerId". All the columns contains the user's properties will be returned.

- Get log data by specifying a date: The user's log data will be constantly recorded and stored in the database. This query is used to get the log data for one specified time granularity.

- Get the skills of a hero: When a user wants to check skills of her/his hero, this query will be used to get all the her/his hero have learned.

- Find a hero for a user by giving email: The user' unique Id is a distributed unique UUID. It is hard for user to remember a regularly UUID, so we allowed the user to use her/his email to login the game. But the UUID are used to uniquely identify one user, so we properly need to find a user's characters by a specified email address.

- Get a hero's basic information by hero's ID: This query are used to get a hero's character information. After user login the game, typically the user will get the chance to select an existed hero or create a new one. If the user has created some heroes before, she/he will see all the characters she/he created and select one of them to continue the game.

After we have analyzed the necessary query patterns, We will explain why they are designed like this and how to implement them with the best practice.

- **Storing and retrieve game player data**

  The user's character data are one of the most important data in an MMORPG world. They need consistency and high availability. Some of the player's data need transaction. For example, if the user's experience reaches to a certain value, the level of the user will increase and the upper limitation of the experience goes up. All the attributes' performance of the user such as attack, defence will also increase. Those values are changed at the same time. Cassandra has no multi-row transaction, but it does support a row-level atomicity as we mentioned before. So in order to keep the user's character information atomic, those data should be stored in one row. Since all the columns in a row are stored sorted and we can use "SliceQuery" to get a range of columns by specifying a starting and ending, all the characters of one user can be represented by a combination of strings. The user's character data such as race, users' basic information and hero would be always read together. So the best practice in Cassandra is to store them together in one single row to achieve an efficient reading. the following shows the hero column family in our prototype.

```
Hero{
    rowkey:0a067127-b915-48fe-a021-f702781c5fec
    { username:player1, email:test@mail.com, race:human, level:12, defence:45 }
    rowkey:8b371834-a285-d583-192b-d837296jd851
    { username:player2, race:night elf, level:23, defence:33 birth:10/1985 }
    ....
}
```

We store all information together. To retrieve a hero created by one user, we can use a “SliceQuery” by specifying in this case “starting” and “ending”, and then the desired information will be returned. If we want to get user’s information by specifying the email, then the value of email column must be set as secondary index.

- **Storing and retrieve Inventory data**

In the column family inventory stores the users’ properties. Every hero’s inventory is stored in one row. To avoid join, the item table is integrated into the inventory. So in Cassandra design, the item column family doesn’t exist. This will cause some redundancy but lead to highly read and write performance. If we want to get a user’s inventory information, we should only do a “QuerySlice” with specifying the user’s id. The column name is the item’s unique id and the value is the item’s information. In our MMORPG prototype we create an item instance with those information. A simple example of column family “Inventory” shows in Figure 5.5

<table>
<tr><td rowspan="2">heroId</td><td>itemId1</td><td>itemid2</td><td>itemid3</td><td rowspan="2">...</td></tr>
<tr><td>A sword|16|500</td><td>A armour|17|450</td><td>payload</td></tr>
<tr><td rowspan="2">heroId</td><td>itemid1</td><td>itemid2</td><td rowspan="2">...</td></tr>
<tr><td>A sword|14|600</td><td>A armour|12|300</td></tr>
<tr><td colspan="3">...</td></tr>
</table>

Figure 5.5: Cassandra column family inventory sample

- **Storing log data**

For storing the log data, we have created a log column family. The game event log data are stored in this column family. As shown in Figure 5.6 on the following page, the row key is ‘ddmmyyhh:eventtype’, and one row represents events within one hour of a day. The MMORPG server will generate a new row in every hour and all events occurring within this hour will be recorded in this row. The column

name stores the time when the event occurred in a finer granularity, such as seconds. The column value stores the payload. Many events could happen within one hour so that a row in the log column family can be very wide. Cassandra's row can hold up to 2 billion columns[Dat13d] so we don't need to worry about the capacity of a row. However, in Cassandra, one row will not be split across nodes and stored together in one node. If we simply use the hour as our row key and keep the event data within an hour together in one row, this will be problematic: Firstly, if we have a super large row with millions of columns, the size of this row can be so large that they cannot host in memory entirely. Second, since the data is handled by only one node in cluster, the entire write request will consequently go to the single node which is holding the row for the current hour, and then this node would be probably a hot spot. The event log data in one MMORPG Cloud be very large, so we have added an event type after the row key so that there will be several rows for different event type within one hour. So the write operations for different event types will go to different nodes in the cluster. By retrieving data, we could use a multi-get for an hour from all of the nodes and merging the results in the application.

0109201323|system	timeuuid1	timeuuid2	timeuuid3	...
	System error	System event	System event2	
0109201323|user	timeuuid1	timeuuid2	...	
	New user created	User quit game		
...				

Figure 5.6: Cassandra column family event log sample

- **Storing other game data**

In our prototype, the column family NPCs and Maps are just like a regular table in relational world. Those column families are stored and retrieved as in RDBMS. However, some of the columns(such as mapId, path) in those column family has been duplicated also in other column families. In contrast to RDBMS, Cassandra cannot update these duplicated columns in different column families. This job has to be done at the application level.

5.3.4 Comparison

Now we have created a RDBMS schema and a Cassandra data schema for our MMORPG prototype. In this subsection we want to make a simple comparison between these two schemas. In RDBMS schema, our design follows 3NF rules. We store Race table, Map

table and Item table separately. The Player table and Hero table use foreign keys to reference to those tables. A design like this can reduce the redundancy in a database. If we want to get the data that the application needs, we can use the powerful SQL to do this job. Additionally, we can also use transaction to perform some item buy/sell operations to ensure the consistency and atomicity. With this RDBMS schema, we can also do some aggregations and group jobs to get the player's statistics, which in Cassandra is hard to achieve. But if we want to add some new features to the game, we had to change the structure of the tables which will be very complicated and inconvenient.

A Cassandra schema design has a flexible structure and we can add new columns in Cassandra column family any time. So the new features or improvement can be easily applied in the database layer. Cassandra can also hold a large number of players' information and linearly scalability in response to the increasing number of players. However, to avoid join operations, we have duplicated the map, race and item table into their related column family. This will cause a reasonable data redundancy and consume more disk space and network traffic. With the Cassandra design, we can only perform the operation we have planned efficiently. However, we cannot use the powerful SQL to query the column family. If we want to get more specific queries we haven't plan up in front, we just need to add new column families in it and this is very in convenient.

5.3.5 Creation and Query of Column Families in Cassandra

There are tools like CLI and CQL available for us to interact with Cassandra. As we have introduced in Chapter 4 on page 25, CQL is a more advanced tool than CLI and it is also recommended by official Cassandra community. However, when we implemented the prototype, CQL was in an unstable status and had no full-featured java libraries. Thus we chose the CLI as our client tool. Before we can use a Cassandra, we must create a keyspace for our prototype and column families. With the syntax shows in Listing 5.2 we can create a keyspace JMMORPG for our prototype.

```
CREATE KEYSPACE JMMORPG
with placement_strategy = 'SimpleStrategy'
and strategy_options = [{replication_factor:1}];
```

Listing 5.2: CLI creating a keyspace

The Listing 5.3 on the next page shows how to use CLI to create a column family.

After we have create the data structure in Cassandra, we can use java and Hector to read or write data into Cassandra cluster.

Hector

Cassandra is implemented in Java, so we prefer to use a Java-based API. The raw Thrift and Hector are available for Java to interact with Cassandra. Thrift is the

```
CREATE COLUMN FAMILY gamePlayers
    WITH comparator = UTF8Type
    AND key_validation_class=UTF8Type
    AND column_metadata = [
{column_name: full_name, validation_class: UTF8Type}
{column_name: email, validation_class: UTF8Type}
{column_name: state, validation_class: UTF8Type}
{column_name: gender, validation_class: UTF8Type}
{column_name: birth_year, validation_class: LongType}
];
```

Listing 5.3: CLI creating a column family

official API supported by Cassandra, which is the most efficient one but lacks of some common enterprise features. Hector is a high level Java client for Cassandra. It is the complete encapsulation of the underlying Thrift API and provides more useful functions. Cassandra is just like the other databases, opening and closing will be very resource consumed. Hector wraps the official API perfectly and provides the connection pool. We have chosen hector as our tool to access Cassandra.

While using Hector in the implementation of an MMORPG, we must consider about the query operations which are support by Hector. Normally, there are three types of basic query method supported by Hector API: ColumnQuery, SliceQuery, and RangeSliceQuery. These methods are the encapsulation of the "get and set" function supported by Thrift. In addition, they are able to provide more powerful functions and high performance.

- ColumnQuery

  ColumnQuery is one of the basic query functions used to retrieve a single column by specifying a rowkey. Its function is similar with the SQL query "Select column name from table where rowkey = X". A simple Java code sample shows in Listing 5.4 on the facing page

- SliceQuery

  SliceQuery is used to get a slice of columns. This class has a function called "setRange", which accepts a start column name & an end column name to specify the starting and ending names of the column slice. A Boolean variable in "setRange" indicates whether the retuned value should be in reversed order. There is also an integer variable to specify how many columns should be returned. A sample code from our prototype shows in Listing 5.5 on the next page

  This is an example in our MMORPG prototype. With this function we can get a user by specifying a rowkey. In line 11 we have passed two null in the setRange method which means we want to get all the columns of this user in the stored order.

```
1  public int getMoney(String rowkey) {
2          //create a instance of ColumnQuery
3          ColumnQuery<String, String, String> columnQuery =
4          HFactory    .createStringColumnQuery(keyspaceOperator);
5          //set a column family for this ColumnQuery and specify
6          //the column name which will be retrieved
7          columnQuery.setColumnFamily(ColumnFamilyName).setKey(rowkey)
8                  .setName("money");
9          QueryResult<HColumn<String, String>> result
10         = columnQuery.execute();
11         money = Integer.valueOf(result.get().getValue());
12         return money;
13 }
```

Listing 5.4: Using a ColumnQuery to get money of a user

```
1      public GamePlayer getPlayerByRowKey(String rowKey) {
2          //create a instance of SliceQuery
3          SliceQuery<String, String, String> query =
4          HFactory.createSliceQuery(keyspaceOperator,
5          stringSerializer, stringSerializer,stringSerializer);
6          //set the rowkey for this query
7          query.setKey(rowKey)
8          //set column family for this query
9          query.setColumnFamily(ColumnFamilyName);
10         //set range for this query, and return maximal 100 columns.
11         query.setRange(null, null, false, 100);
12         QueryResult<ColumnSlice<String, String>> queryResult =
13         query.execute();
14         return queryResulToGamePlayer(queryResult);
15     }
16 }
```

Listing 5.5: A sample of using a SliceQuety

- RangeSliceQuery

  RangeSliceQuery is normally used when we have lower or upper rowkey's bounds and use them to find all entities within this range. After Cassandra version 0.8, it also allows user to specify a where clause on the column which is defined as a secondary index. The equivalent SQL query would be: "Select * from gamePlayers where username = X ".

The 11 line in 5.6 on the following page accept two parameters, we have assigned two null to this method. That means we want to scan the whole column family to find the specific user.

```
1      public void getPlayerByLoginName(String Loginname) {
2          //create a instance of RangeSliceQuery
3          RangeSlicesQuery<String, String, String> rangeSliceQuery =
4          HFactory.createRangeSlicesQuery(keyspaceOperator,
5          stringSerializer,stringSerializer, stringSerializer);
6          //set the column family for this query
7          rangeSliceQuery.setColumnFamily(ColumnFamilyName);
8          rangeSliceQuery.setRange(null, null, false, Integer.MAX_VALUE);
9          //set the rows should be returned
10         rangeSliceQuery.setRowCount(row_count);
11         rangeSliceQuery.setKeys(null, null);
12         rangeSliceQuery.addEqualsExpression("username", Loginname);
13         QueryResult<OrderedRows<String, String, String>> result
14         = rangeSliceQuery.execute();
15         ......
16     }
```

Listing 5.6: A sample of using a RangeSliceQuety

5.3.6 Deployment of Cassandra Cluster

In this section we will illustrate the installation and configuration of our Cassandra cluster in virtual machines. Cassandra is available in the Internet. We have downloaded the version 1.2.5 and the extracted ZIP file to create a folder with the name "apache-cassandra-1.2.5". This folder contains several directories. The CLI and CQL tools are in the bin folder and the configuration files are in the conf folder. In order to set up a Cassandra cluster instance, some parameters in "cassandra.yaml" file must be appropriate configured. The following are several important parameters related to our Cassandra cluster.

- cluster_name: This is the name of the cluster and mainly used to ensure every node can find the right cluster and prevents machines from joining in another cluster.
- listen_address: This is the address to bind to and tell other Cassandra nodes to connect to. This will always be the local IP address of a node.
- rpc_address and rpc_port: This address and port are used to bind the Thrift RPC service to. If all interfaces need to be listened by Thrift, here must be specified "0,0,0,0,"
- seeds: '192.168.144.14,192.168.144.16'; This parameter is used to initiate a new node. It can tell the new node, which wants to join in the cluster, to which node it should connect. Once a node is joined into a cluster, all nodes will use gossip protocol to talk to each other.

After necessary configuration, we can start the five nodes cluster on the virtual machines. In order to manage the nodes in Cassandra cluster with a unified platform

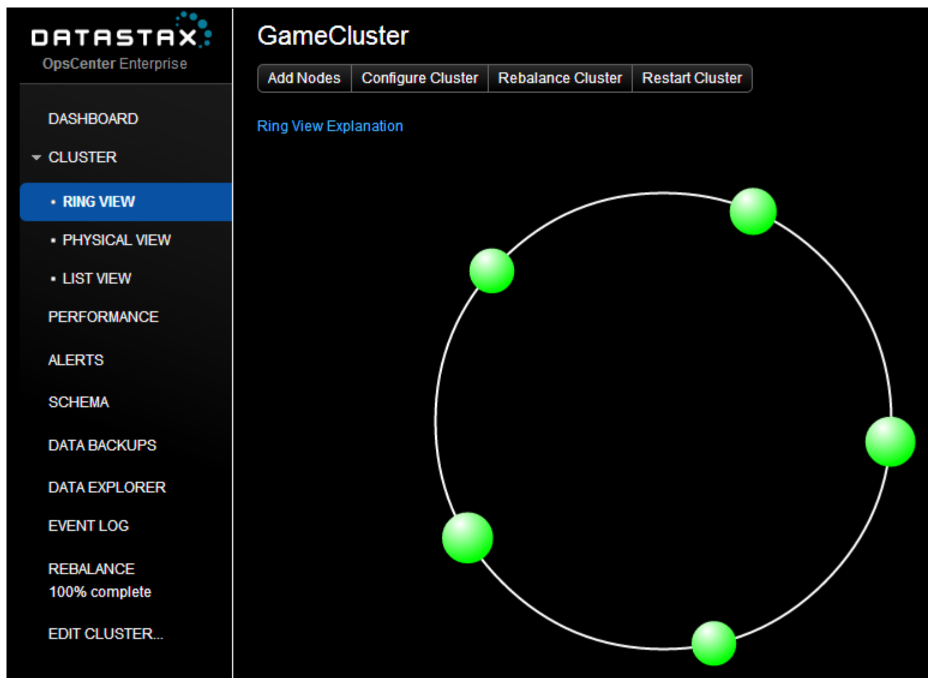

Figure 5.7: Datastax Cassandra OPSCenter

efficiently , there are several web-based administrator interfaces available. Those interfaces allow us to create/delete databases, insert/delete data values even more higher level functions such as monitoring the JMX , CPU, Disk I/O. One example is the OPSCenter, it can be used to connect and interact with the Cassandra cluster and give us an intuitively interface to manage Cassandra cluser as Figure 5.7 shows.

5.4 Game Server Design and Implementation

Now we have the Cassandra cluster running on the virtual machines. In this section We will explain the game server design and its implementation. The game server consists of three layers. The most basic one is the data accessing layer which is used to communicate with the Cassandra Cluster. The communication layer is based on Darkstar project, and the business layer is responsible for handle the commands send by users and dealing with game logic. Figure 5.8 on the following page is a UML java class diagram, which shows some of the most important Java classes in our server side.

5.4.1 Darkstar

Darkstar was an open-source MMOG middleware solution which is dedicated to help the programmer to develop a robust server and client with little effort[9]. It was an open-source Sun project. After Sun was acquised by Oracle, the project has been shut down and it is not be support by the new company Oracle any more. This project has

[9]http://en.wikipedia.org/wiki/Project_Darkstar

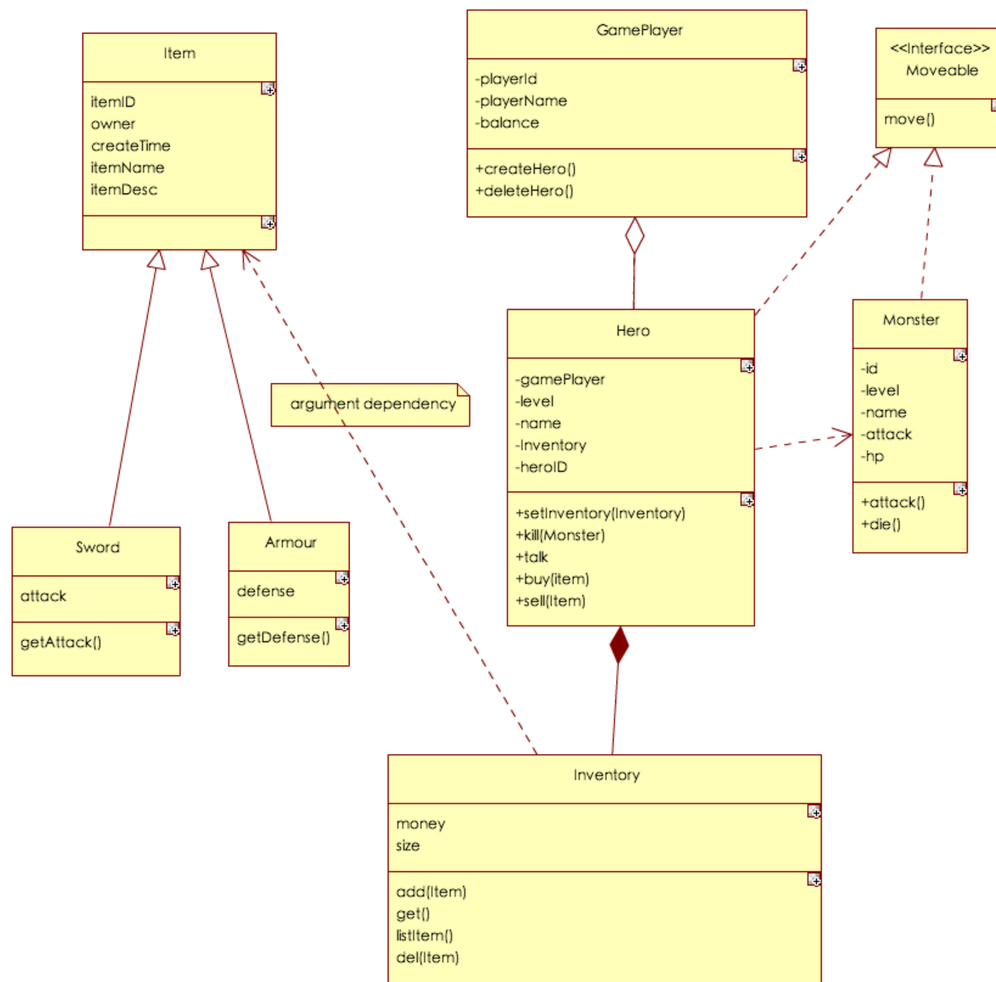

Figure 5.8: Class diagram

been made renamed "RedDwarf Server"[BW09]. However the latest version of Darkstar can fullfill our requirements, so we have built our MMORPG with this project. Darkstar provides an convenient functions library which helps developers to deal with the challenging aspects of networked game development[Bur07]. With the help of Darkstar, we can focus on the game logic and database design of the testbed. Network challenges will be solved by the Darkstar project.

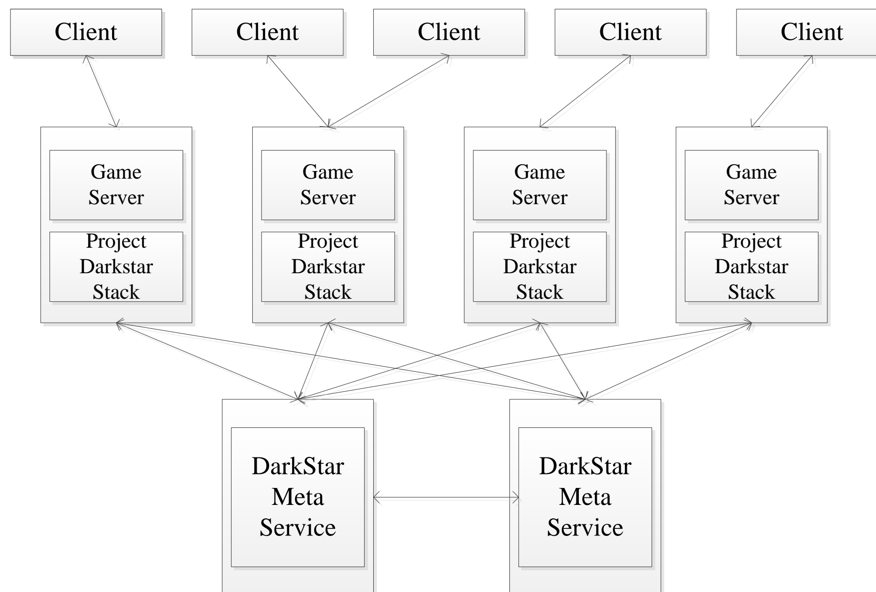

Figure 5.9: DarkStar architecture adapted from [Bur07]

The basic architecture of a Darkstar server shown in Figure 5.9. The implementation of a Darkstar project called Project Darkstar Server (PDS), The instance of a PDS can either start a new network or join one that is already running. The client and the server implementation is user created program written with the Projcet Darkstar API. The project stacks on the server side and several meta-service nodes handle traffic between each node in the server stack.

The core components of Darkstar API are shown on Figure 5.10 on the following page. In a Darkstar application, there are three standard managers. The instance of those managers can be acquired from the "AppContext".

- **Task Manager**
  The Task Manager schedules and creates the single Task in the server implementation. Task is the smallest executable job in PDS and it runs concurrently. However, the programmer can treat Task as single-thread and the data within a Task do not need to be synchronized. The PDS hides the complex multi-thread implementation. The execution is also race-proof and deadlock-proof. A single Task can only keep running up to 100 ms in PDS.

- **Data Manager**
  The Java object, which needs to be persist in the game world, is called "Managed Object" in a PDS. The Data Manager is used to create and access the "Managed Object".

- **Channel Manager**
  A PDS application uses "Channel Manager" to create and control the channels. Channels are used to communicate between clients and servers.

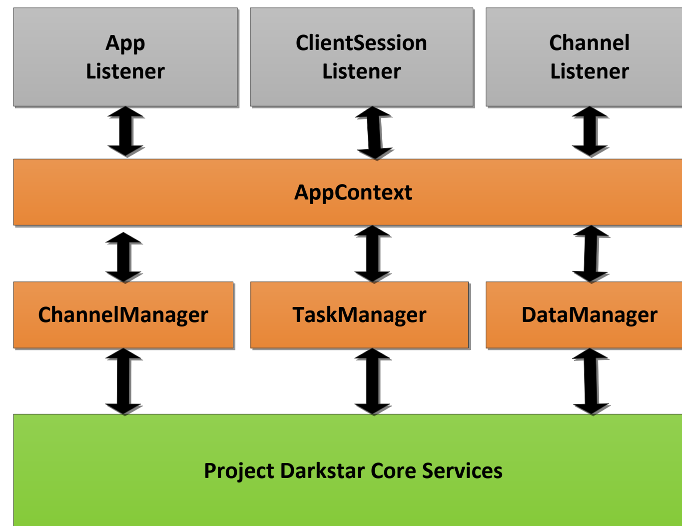

Figure 5.10: DarkStar core component adapted from [Bur07]

5.4.2 Data Structures

To transfer the objects between client and server, we need to define some standard data structures. We will use a class *GamePlayer* to hold all information about the user as Listing 5.7 shows.

```
1
2  public class GamePlayer implements Serializable {
3  //this class used to define data transfer object
4      string uuidString;
5      string userName;
6      string birth;
7      string email;
8      string lastActiveIp;
9      string lastActiceDate;
10     string registDate;
11     string heroClass;
12     int raceId;
13     string heroRace;
14     int currHp;
```

```
15     int MaxHp;
16     int currExp;
17     int MaxExp;
18     int strength;
19     int attack;
20     int defense;
21     int mapId;
22     int classId;
23 }
```

Listing 5.7: Data structure game player

A client sends commands to the server. We have created a *Command* class to represent the communication between client and server. The structure of a command is shown in Listing 5.8. *CommandContext* is a message context send from the user. We define 3 types of commands: the first one is user commands which is sent by the user; the second one is system administrative command, which is sent by game server; the third one is information, which is used to broadcast the hero's position and game state.

```
1
2 public class Command implements Serializable {
3     public string commandContext;
4     public int commandtype;
5     public string sender;
6 }
```

Listing 5.8: Data structure command

Those structures above are only used to hold the basic information and for convenience in the application. An example sequence diagram for the most basic game logic is described in the Figure 5.11 on the next page.

We use Cassandra as the underlying database, which is written in Java. The Cassandra client interface is platform independent. As we mentioned, we are using Hector to interact with Cassandra cluster. The server side is implemented with a java project. The detail package we can find in Figure 5.12 on page 59.

- *game.cassandra.dao*: it contains common Data Access Objects(DAO). We have applied the traditional DAO layer for isolating the business layer and data accessing layer. Every class is an implementation of data access interface. For instance the class *CassandraDAOGamePlayer* help us to access the Column Family Hero in Cassandra

- *game.cassandra.data*: the basic data structure locates in this package, such as game user and command.

- *game.darkstar.network*: this is the basic networking framework. It is the implementation of Darkstar and is responsible for essential networking packet sending and receiving.

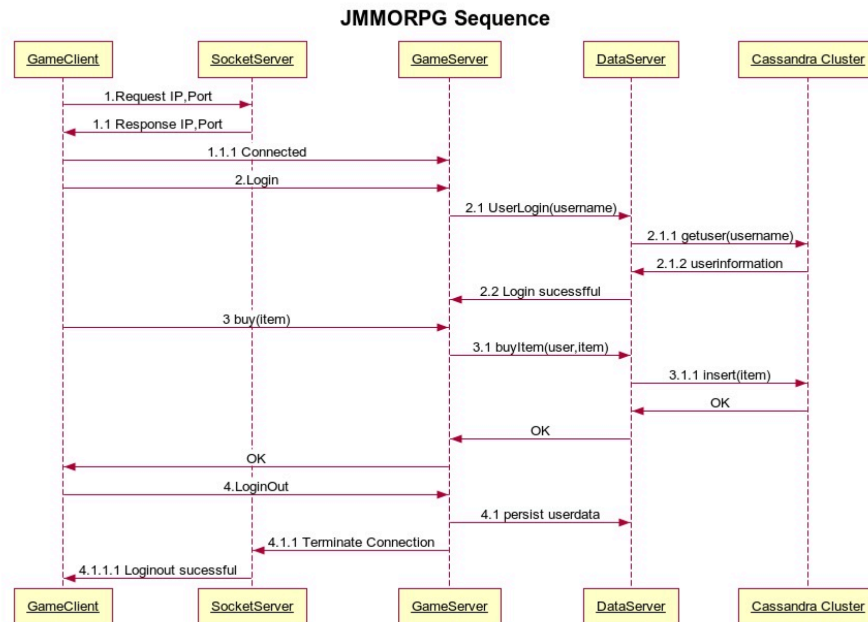

Figure 5.11: Sequence diagram

- *game.cassandra.Factorys*: it contains some factory classes, such as *Gameplayer-Factory*, *ItemFactory* which are used to generate the random basic data for testing purpose.
- *game.cassandra.gamestates* : it contains classes that stores some game logic used information and the some game logic.
- *game.darkstar.task*: in this package, we have implemented some tasks which are performed periodically.
- *game.login.authenticator*: this package contains classes which are responsible for the authenticate job.

5.5 Game Client Implementation

5.5.1 JMMORPG

JMMORPG is an open-source project available on "sourceforge"[10]. It is a very simple Java game server and game client running on a RDBMS database. We have implemented a new game server with Cassandra database and borrow some ideas and the client GUI from this project. The client side elements such as hero figures, maps and so on are adopt from this project. The reader can find this project on "sourceforge"

[10]http://www.sourceforge.net/

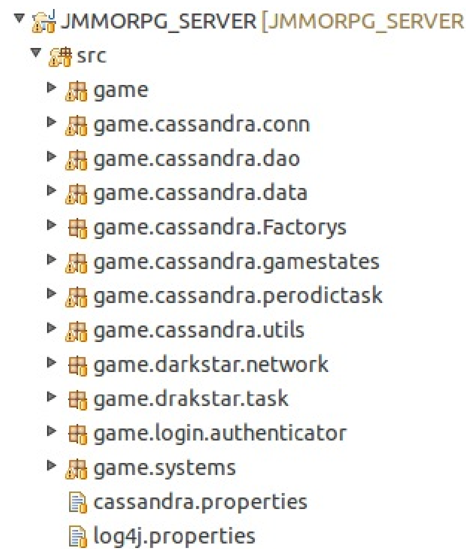

Figure 5.12: Server class implementation

5.5.2 Client Libraries

The Client has the basic network function and it must be able to receive the commands that broadcast by the game server and visualize the game item to the users. To simplify the client rendering, we have adopted a lot of existed Java libraries to help our programming. In this section, we are not intent to introduce the implementation of game clients in detail, but only explain some core java libraries that we are using.

- **LWJGL**: its full name is "Lightweight Java Game Library". This library provides Java developers the ability to get access to high performance cross-platform libraries such as OpenGL[LWJ13]. Java is not particularly good at game client development, but with the help of this library, we can get resources that are unavailable or poorly implemented on existing Java platform.

- **Artemis**: it is an open source high performance Entity System framework for games[ART13]. We use this library to manage entities in our Prototype. It provides a new concept by separating the functions of entities to simplify game design. The core idea of this design is to separate data, logic and view. This framework requires a little different thinking in game design, for example, the entities in a game world are no longer self-contained object but consist of data state components, and the system processes entities based on the their aspects.

In Figure 5.13 on the next page shows a client screen-shoot of our prototype. Five users have logged to the game and they are talking in the game world. We have only one map in the whole game world, More features will be added into the game in the future. The testbed is running well on top of Cassandra and we will discuss the evaluation results based on some database performance charts in the next chapter.

Figure 5.13: A screen shot of our prototype(client side)

6. Evaluation

6.1 Experimental Environment

In previous chapters we have introduced the basic concepts of Cassandra and the implementation of our prototype. Now we are going to introduce a series of experiments, which have been done through the experimental prototype, and analysis experimental results. The experimental environment is as illustrated in the previous chapter. There are eight virtual machines available for running our prototype. Three of them are deployed as game servers and the others are used as Cassandra cluster nodes. The client sides are running on separate PCs. Those virtual machines are located in an Intranet environment. For the safety reasons, all the applications from internet (such as game client) can only get access to the virtual machines through a VPN server (such as game server, Cassandra cluster). There could be a potential problem that the Virtual Private Network (VPN) server could be a bottleneck of our prototype, because all the communication between game server and client must forward by VPN server. However, this problem haven't appeared in our prototype under the current data accessing pressure, for example, when we increased the number of game clients up to 1500, there is no obviously latency of the VPN server.

6.1.1 Experimental Clients

Command-Line Client Side

Normally, we need to run thousands of game clients on one computer, so that a single game client instance will not consume too much system resources during the test, otherwise a single computer cannot afford so many instances of game clients. For our performance test we developed simple command-line client side, which has all the functions of the GUI clients, expect of specific game pictures rendering. We can send a command to perform the operations in the GUI client. For instances, "buy item" can be sent for item buying and this item information will be persisted into the Inventory column family.

Multi-Processes and Multi-Threads

If we want to simulate the situation of multi-player accessing a game server, we have normally two choices. One is to use process to represent a user and another is to use thread. Both of them has advantages and disadvantages. If hundreds of processes are stared simultaneously in one computer, they will consume lots of resources (CPU, memories) of this computer, and waste lots of time until they all start. We tried to start 500 processes in one computer and the computer crash with the increasing number of started processes. If 500 threads are running in one computer to represent game players, those threads will share the resources of their parent process, but it is not the case of the real application. Multi-processes with multi-thread has the potential to compensate between process and thread and reach to a balance performance. We have adapted a multi-processes and multi-thread way to perform the pressure test. For example, we can run 5 process and each of them will generate 100 threads each of which represents a player. In addition, all of our experimental results are based on the same network and hardware/software conditions, which means, all of the results have the same divergence and can present the real scalability tendencies.

6.1.2 Configuration of Cassandra

This evaluation focuses on the performance of our prototype in the case of multi-player concurrent accessing. To avoid the effect of replicas to the reading and writing performance of Cassandra cluster, we set the replication factor of the keyspace as one during the whole evaluation. That means, the cluster has only one replica of keyspace "CLOUDRDBMS" and the read/write operation will return success immediately once one node of the cluster has responded to them. Because the data in Cassandra are partitioned as the calculated token values of the row key, we need to set the token value in the configuration file "cassandra.yaml" of each Cassandra instance manually in different experiments in order to reach the data balance on each node. Even though, in some cases we still can monitor the hotspots which means one or more nodes are accessed too frequently in a short time from the Java Management Extensions (JMX). When hotspot is detected, the ongoing experiment will be carried out again.

6.1.3 Methodology

Our evaluation is realized by the simplified Command-Line client and the game server. We use python script to invoke the Java client class. Every Java client is able to generate 100 threads, each of which represents one player. The python script can start many Java clients processes, that means, each Java client represents for 100 players. We can start at most 5 client processes simultaneously because a single game server is only able to afford 500 users at same time currently. After those threads are started, each of them will send 500 commands to the game server simultaneously, Theses threads will terminate after sending all commands. In our experiments, we tried to simulate the real operations of a game player during the game. We classify the commands into writing and reading. The Cassandra write operations include: "buy item", "player register",

"generate hero", "hero information modify", "write log". Data will be written in column families "Inventory", "Heroes", "Users", and "Log" respectively. The Cassandra read operations include "get player", "get inventory", "get hero's information", and "read log". Data will be read from above mentioned column families. The server will record the response time of each operation in Cassandra. After each experiment, the response time will be aggregated and processed, and then recorded in a log file. In order to simplify the statistics, the reading and writing commands are sent separately from clients to game server. That means, clients send a large number of random writing or reading commands, the response time of those commands will be recorded separately. The purpose of doing so is to investigate more clearly the differences of Cassandra reading and writing performance.

6.1.4 Acquisition and Expectation of Experimental Results

The acquisition of the experimental data is based on recording the Cassandra's response time to each command from clients firstly, and then the mean value of these response time will be calculated as reading and writing response time of one single experiment. In order to investigate the distribution of response time of data accessing, we set 200 clients and 1000 commands send by each client. Afterwards the average response time of each command of the 200 clients are calculated and plotted.

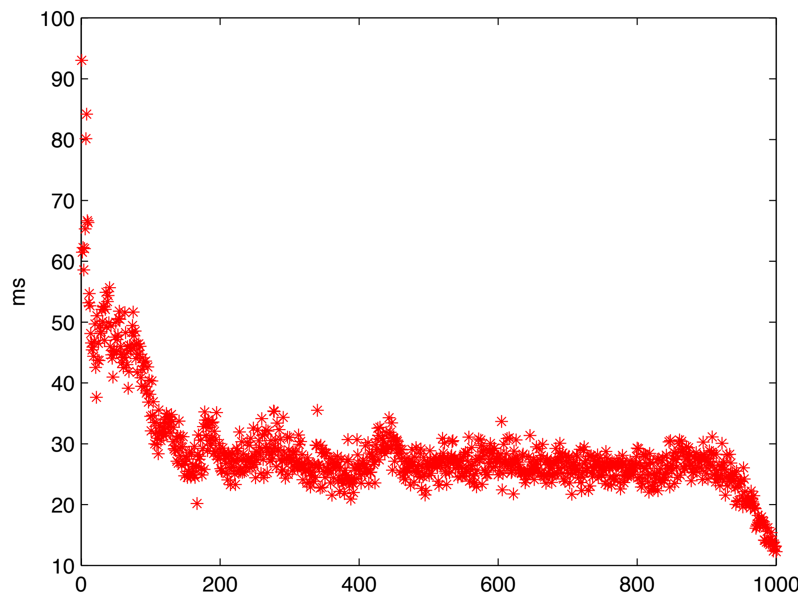

Figure 6.1: Average response time of 200 clients

As can be seen from Figure 6.1, the response time of beginning commands are relative high, which has a negative affect on experimental results analysis, so they are ignored. The reason for this phenomena could be establishing database connection and preparation of resources is necessary at the beginning time. Because the game server handles

concurrent user in different velocities, there might be part of players quit by finishing their sending job earlier than the others. Correspondingly the server has less pressure during the ending period and the response time of data accessing commands is low. This part of results are ignored either. Our experiments are aimed to survey the response time variations with the increasing number of client sides in cases of different nodes in Cassandra cluster, and in further to investigate the performance of a Cassandra based MMORPG.

We expect to see that, Cassandra shows high reading and writing performance in MMORPG environment. In addition, we hope that our prototype is able to support more game clients through increasing the number of game server as well as Cassandra nodes. Normally, the requirement of latency of MMORPG smoothly running is less than 200ms [CHL06]. When the Cassandra met the reading/writing bottleneck, we could released the database pressure by increasing nodes into the Cassandra cluster.

6.2 Experiment Results and Discussion

6.2.1 Performance of the Prototype

In order to investigate the maximum number of players that our prototype can support, we fixed the number of nodes to 5 in the Cassandra cluster, and increased the number of players as well as game server. Three groups of experiments have been done:

- First Group

  The first group uses one game server and runs concurrently 100, 200, 300, 400 ,500 players on client side respectively. Every player send 500 reading/writing commands at same time and the mean response time is calculated as shown in Figure 6.2 on the next page. When we have 600 players, there will be many "time-out" appeared on game server, which cause the fail of command sending. So we can conclude that, the maximum number of players in case of single game server is 500. From Figure 6.2 on the facing page we can also find that the response time of reading and writing is under 15ms, which means 500 players put little pressure on 5 nodes Cassandra cluster in our prototype.

- Second Group

  We use one more game server during the second group of experiments, so the player's number also increases to 200, 400, 600, 800, 1000. The number of commands sent by every play is same as the first group (500 random read/write commands). The responds time of current players are shown in Figure 6.3 on the next page.

- Third Group

  We use three game servers in the third group, correspondingly the player's number are 300, 600, 900, 1200, 1500. The responds time of current players are shown in Figure 6.4 on page 66

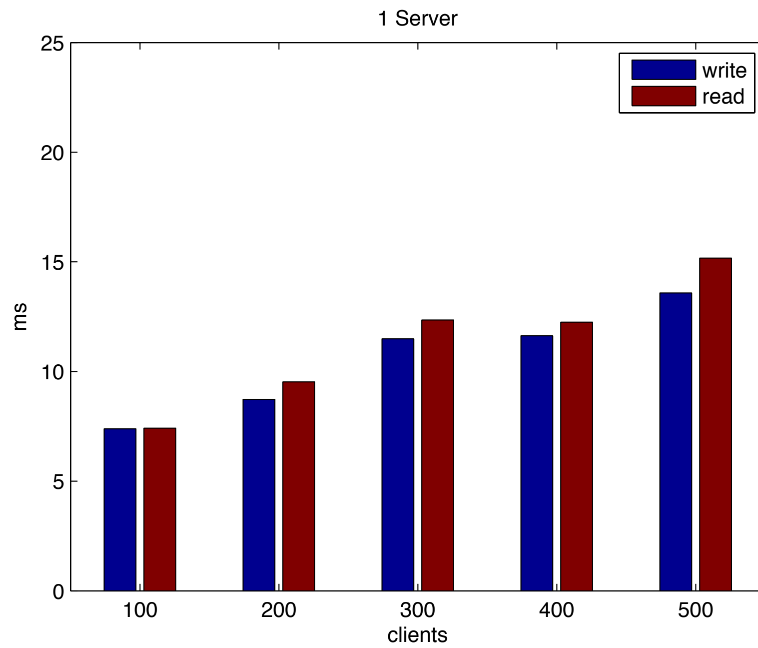

Figure 6.2: Reading/Writing response time for group 1

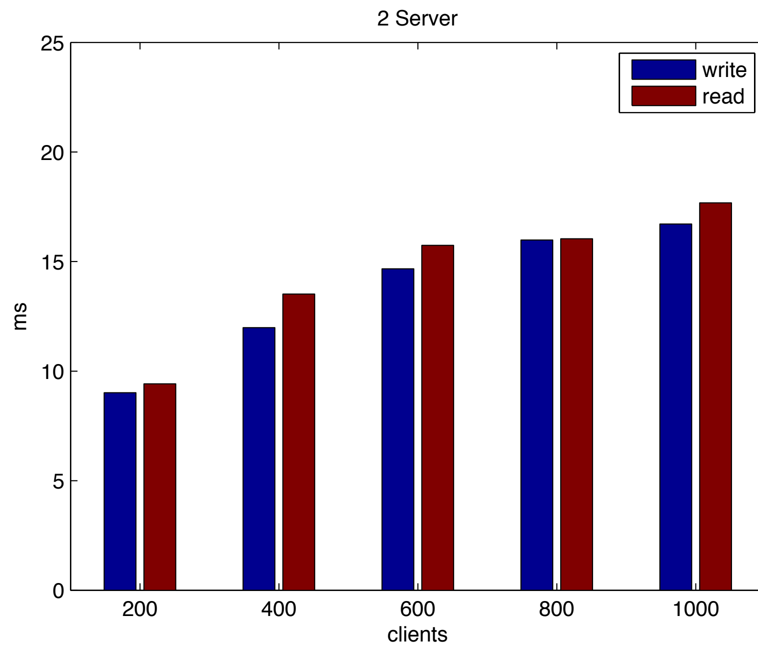

Figure 6.3: Reading/Writing response time for group 2

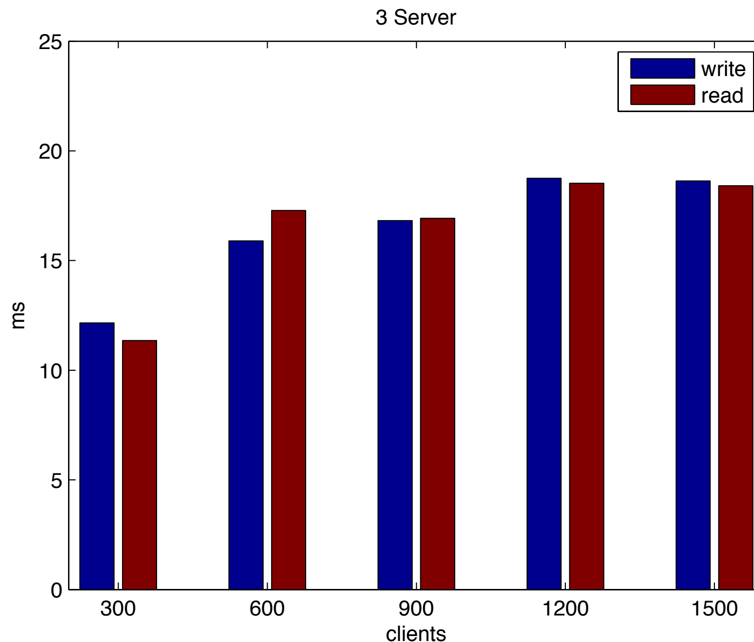

Figure 6.4: Reading/Writing response time for group 3

Limited by our game server and experimental infrastructure, the maximum number of player is set to 1500. From these three figures, we can find that although the response time of Cassandra cluster fluctuates with changes in the number of game server and player, it is always well below the upper limit of 200 ms. Therefore, we believe that our prototype with 5 nodes Cassandra cluster is able to support at least 1500 players easily.

6.2.2 Scalability of the Prototype

The scalability of a game includes two aspects: the scalability of game server and database. From the experiments in the last subsection, we can also find that with the constant adding game sever, the maximum number of players increases linearly from 100 to 1500, which proves that the game server in our prototype has scalability. In the following, we will evaluate the scalability of Cassandra cluster in MMORPG environment.

We carried out 5 groups of experiments to evaluate the scalability of database. The maximum number of concurrent players is set to 1500, which is obtained from our previous experiments. The number of nodes of Cassandra cluster is set from 1 to 5. The number of game server is fixed to 3. Each of the 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 clients sent 500 reading or writing commands. In another word, Cassandra cluster needs to handle 150,000, 300,000, 450,000, 6000,000, 750,000 commands in turn. The corresponding response

time of each command is recorded and afterward the mean response time is calculated. According to the analysis above, the first 1000 commands with relative longer response time and and the last 1000 with relatively lower response time are ignored. To get more precise experimental data, every subgroup is done twice and the average value of them is plotted as the final result.

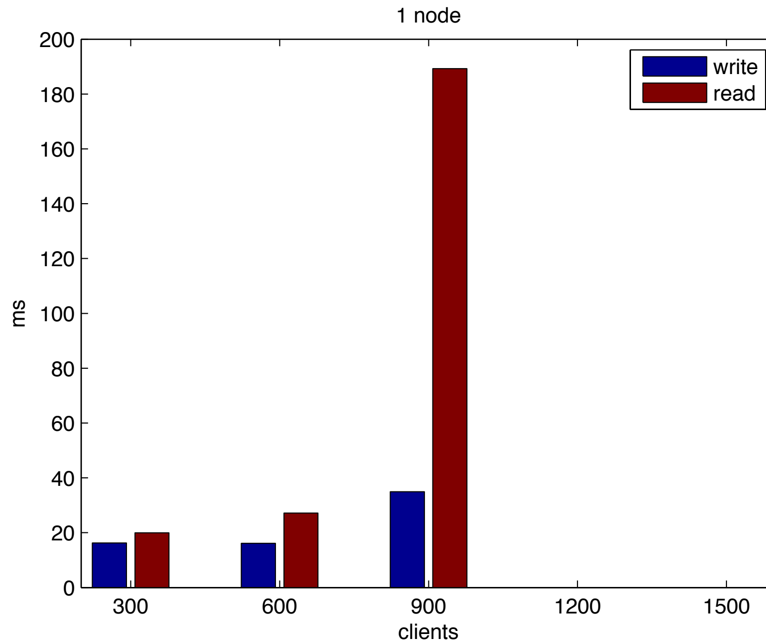

Figure 6.5: Group 1: Performances of 1 nodes Cassandra

From Figure 6.5 we can see that, the best performance of 1 node Cassandra cluster is got under 300 and 600 clients. When the number of clients reach to 900, The response time of read operation increases sharply over 180ms. If we started 1200 clients, Cassandra cluster will not respond the write and read request. Many clients report a connection time out exception. So we can conclude that one node Cassandra can only support up to 900 clients under our experimental environment.

Figure 6.6 on the following page shows that the maximum number of clients reach to 1200, when there are two nodes in the Cassandra cluster. In case of 1500 concurrent players, the problem of "request out of time" will appeared again. So we can say that, 2 nodes Cassandra can support at least 1200 clients.

Figure 6.7 on the next page Figure 6.8 on page 69 Figure 6.9 on page 69 show that, when the number of nodes in Cassandra cluster is more than 3, our prototype is capable to support at least 1500 concurrent players.

In order to observe the differences of 5 groups' results, we plot the reading and writing response time in Figure 6.10 on page 71 and Figure 6.12 on page 72. Because the read response time of 1 node Cassandra at 600 clients is unexpected out of the general range,

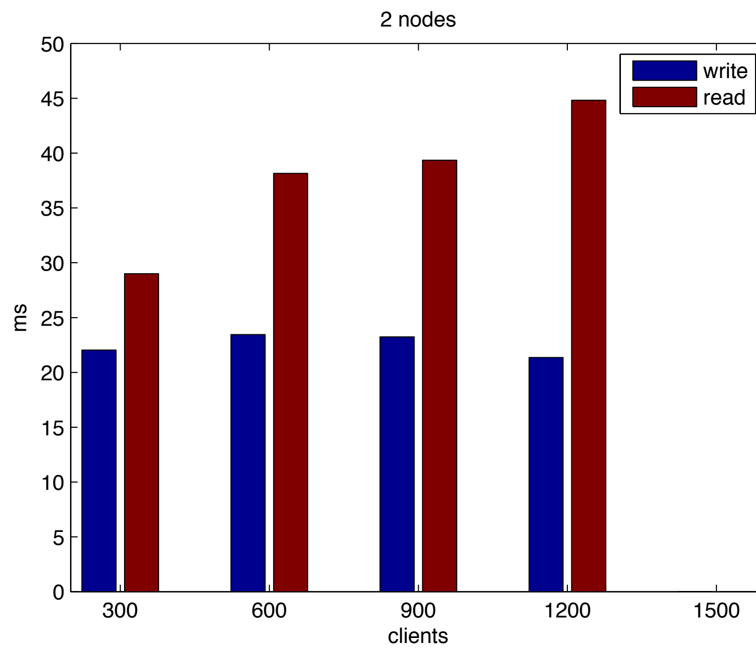

Figure 6.6: Group 2: Performances of 2 nodes Cassandra

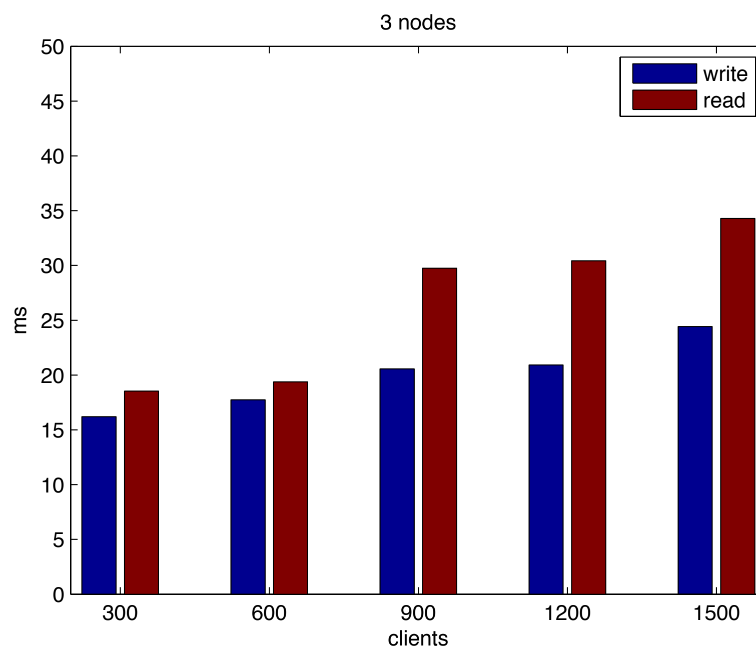

Figure 6.7: Group 3: Performances of 3 nodes Cassandra

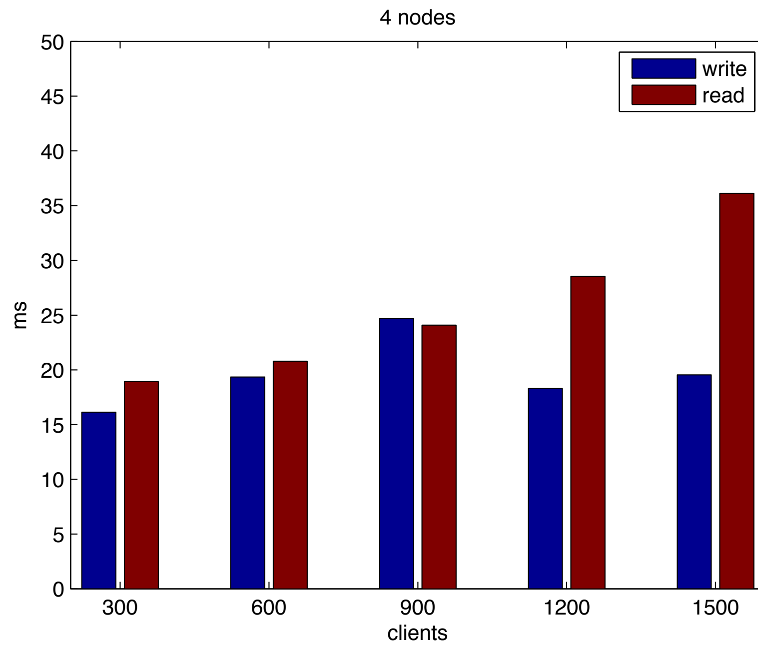

Figure 6.8: Group 4: Performances of 4 nodes Cassandra

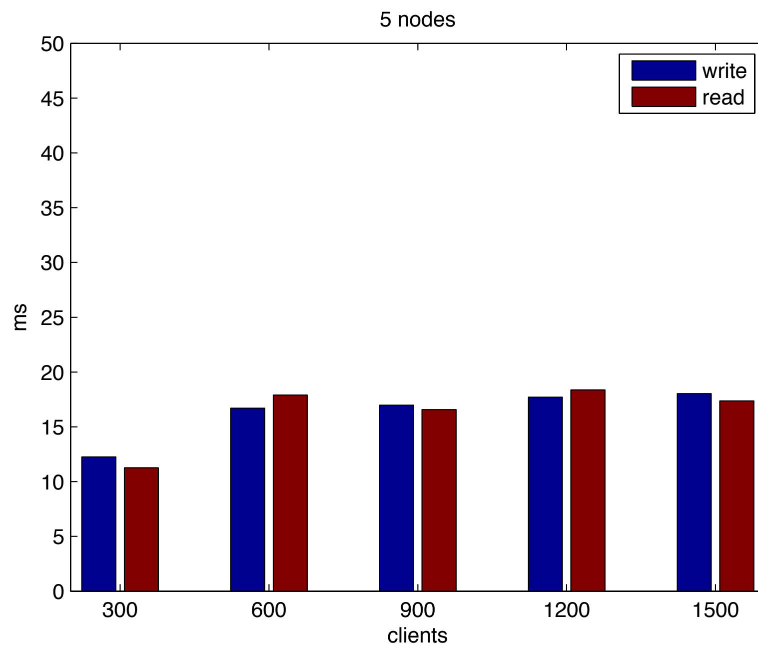

Figure 6.9: Group 5: Performances of 5 nodes Cassandra

so we set the maximal of vertical axis as 50ms. In order to see the detail tendency of the others. The modified figure is shown in Figure 6.11 on the facing page.

According to the analysis of the experimental results, we can find the following tendency:

- With the increasing number of nodes in Cassandra cluster, the number of concurrent players that our prototype can support is also increasing and the reading and writing performance is constantly improved.
- When the number of nodes reach to 5, the performances of Cassandra cluster is the best in all range of clients' number, both of reading and writing response time with 5 nodes Cassandra cluster are about 15ms. With increasing number of clients in 5 nodes Cassandra, there is no obviously variation of reading and writing response time. Evidently, the performance of 5 nodes Cassandra is stable in our prototype. 5 nodes cluster is obviously the best one;
- The performances of 3 nodes and 4 nodes are similar. Normally, the performance of 4 nodes Cassandra should be better than 3 nodes Cassandra. However, our experimental results show that, the reading response time at 1500 users and writing response time at 900 users of 4 nodes cluster are longer. The reason of that could be network latency or the game server logic problems. But the performance of 4 nodes are still generally slightly improved;
- The performance of 3 nodes cluster is generally better than 2 nodes. 3 nodes cluster presents 20ms writing and 33ms reading response time for 1500 concurrent users. Undoubtedly, 3 nodes Cassandra is capable for the maximal number(1500) of game users that our prototype game servers can support;
- The performance of 2 nodes Cassandra is slightly worse than 1 node in case of 300 or 600 clients. The reason could be that the advantage of multi-node Cassandra cluster compare with single node Cassandra cluster is not outstanding when the current players are relatively less. In the meantime, the communication between nodes also consumes some time since the data are distributed on many nodes, which can also induce the phenomena in Figure 6.10 on the next page. However, when the number of current players go up, the performance of single node will decrease dramatically, while the performance of multi-nodes keeps stable.

Base on the analysis above, we can conclude that Cassandra can meet the performance demand of MMORPG in general. The more nodes Cassandra has, the more concurrent players Cassandra can support. With the increasing number of players, the reading performance of Cassandra will be improved a lot, the writing performance stays relatively stable.

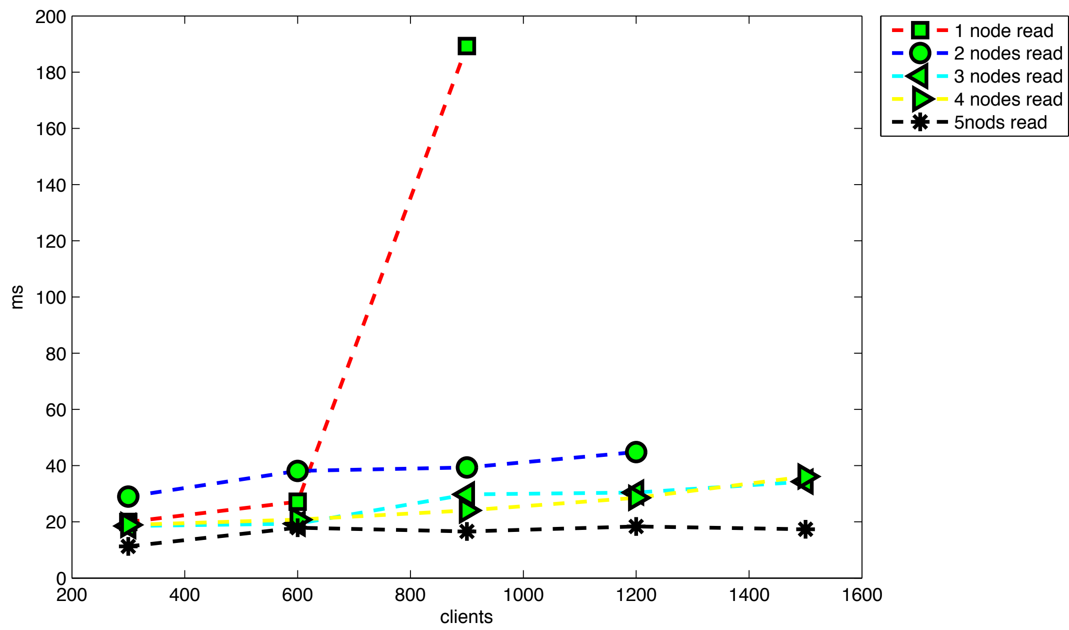

Figure 6.10: Average reading time of 5 groups

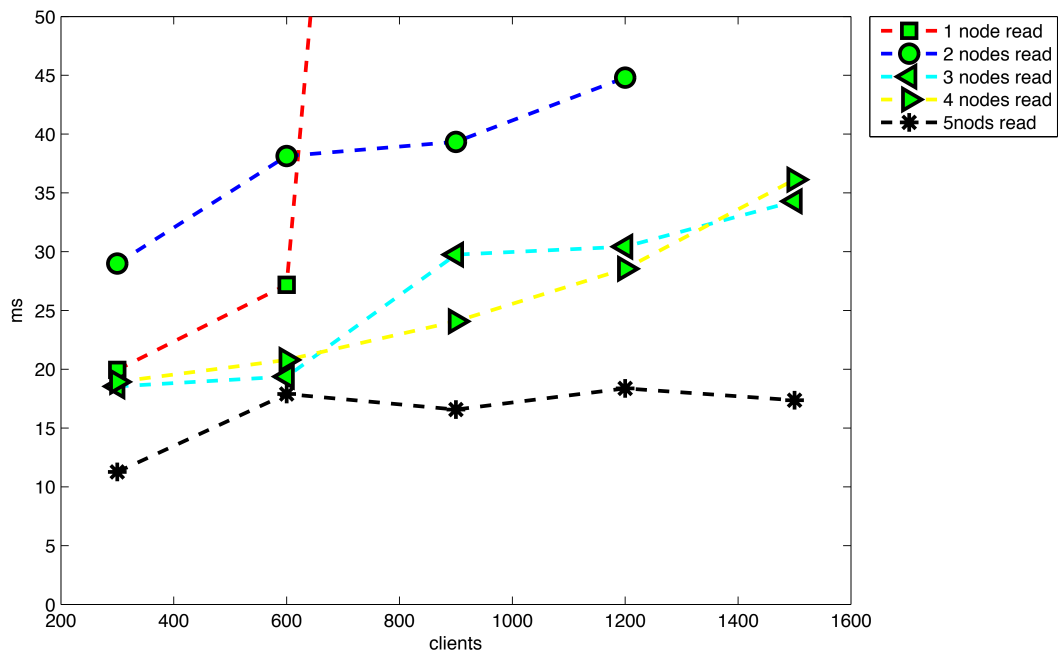

Figure 6.11: Average reading time of 5 groups(modified)

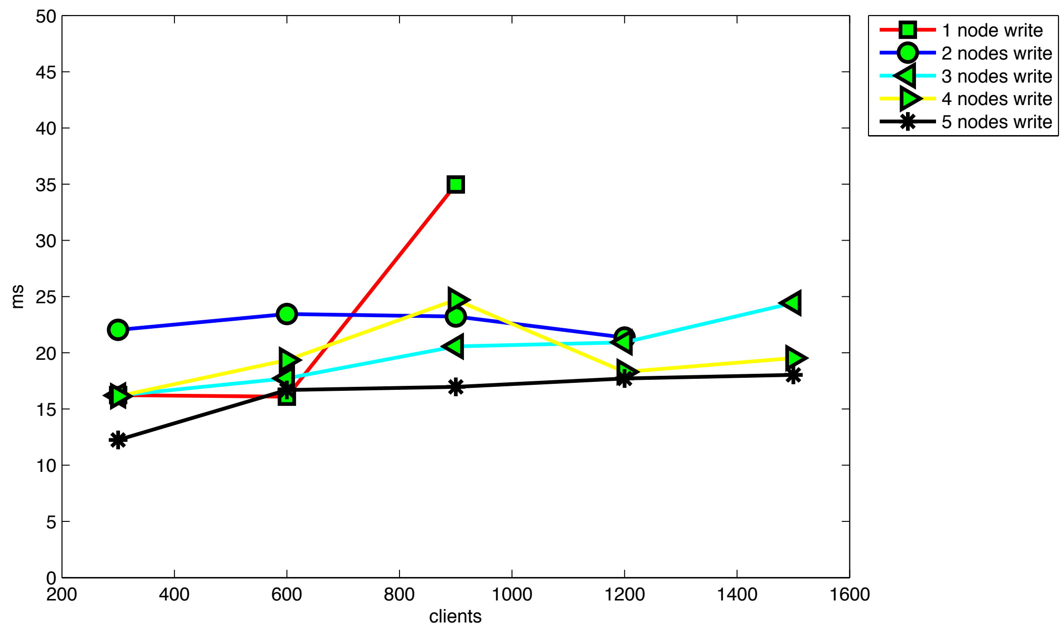

Figure 6.12: Average writing time of 5 groups

7. Conclusion and Future Work

Conclusion

As the favourite type of online game, MMORPG is rapidly developing during the recent years. With the increasing number of players and the accumulation of user data, game developers have to face two issues: the performance of concurrent data accessing and the scalability in game database. The traditional architecture of MMORPG applies a RDBMS to manage data, which will be a bottleneck of scalability of the game. The appearance of Cloud database provides an alternative to overcome the data management limitation. This thesis work started from the characteristics of MMORPG storage requirement and Cloud database, investigated the possibility of applying Cloud database to MMORPG and evaluated the Cloud-based MMORPG.

In this work, we proposed to use Cassandra to manage data in MMORPG. In practice, we found that, the design of a Cassandra data schema must rely on the query pattern. In other words, through increasing data redundancy, Cassandra stores all the data that might be queried together in one column family in order to avoid join operation and improve read performance. For this reason, we have to consider all queries our prototype might perform at the beginning of database schema design. However, Cassandra is not able to perform complex query like aggregation and group by in SQL. In addition, Cassandra doesn't support transactions cross rows. We believe that those two issues have to be addressed in the application level.

According to the demands of the experiment, we implemented a game prototype, our prototype includes a five nodes Cassandra cluster and three game servers. The experiment results reveal that, our prototype can support at least 1500 concurrent players, and can meet the reading and writing performance requirements of MMORPG. Although with the increasing number of players, the performance of the prototype will decrease, but increasing nodes in the the Cassandra cluster and number of game servers is the most promising way to improve this situation. Furthermore, the scalability of this prototype is verified.

Future Work

There are still several factors that limit the performance and scalability of our prototype: (1) Cassandra is still a new born database, so there are a lot of deficiencies (e.g., lack of statistics and management tools) that need to be improved. (2) Cassandra goes though a dozen of version from 1.2.0 to 2.0 during my thesis writing, so the technologies and methods applied in this work could lag behind the latest version of Cassandra; (3)Because this prototype is firstly established, the logic and algorithms of client side and server side need to be optimized. For these reason, the future work should concentrate on the following aspects:

- Using more advanced monitoring method: During our experiments, the operation response time of some subgroups is unexpected long. The reason could be the clients command stock on game servers, or VPN server. Up to now, Cassandra has no perfect monitor to indicate the throughput. Additionally, our prototype also has no function to monitor the activity of online users. Therefore, we cannot verify why this happens and more advanced monitoring method should be add into the prototype in the future.

- The optimization of the prototype: Hector API is applied for the data access layer of our prototype, but Hector is gradually outmoding by other APIs such as "DataStax Java Driver"[1]. So a better API for data accessing is necessary for a better performance data access layer in our prototype.

- Enrichment of prototype functionality: The query pattern of the prototype needs to be improved to get better performance, and new functions will be added to the prototype in the future.

- Comparison experiment: In this work, we proposed to use Cassandra to replace RDBMS in MMORPGs. However, some horizontally comparative experiments between Cassandra and RDBMS as well as another Cloud database need to be done to prove our points. Hence, in the future, we plan to compare the performance and scalability of Cassandra with that of MySQL and HBase in our experimental environment.

[1]DataStax Java Driverhttps://github.com/datastax/java-driver

Bibliography

[17113] 17173. Global mmo market will be 14.9 billion big in 2013 - report. Website, August 2013. Available online at http://2p.com/831716_1/Global-MMO-Market-Will-be-149-Billion-Big-in-2013---Report.htm; visited on August 14th,2013. (cited on Page 1)

[Aba09] DJ Abadi. Data Management in the Cloud: Limitations and Opportunities. IEEE Data Eng. Bull., pages 1–10, 2009. (cited on Page 3 and 13)

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. Communications of the ACM, 53(4):50–58, 2010. (cited on Page 13)

[ART13] ARTEMIS. Artemis entity system framework. Website, July 2013. Available online at http://gamadu.com/artemis/; visited on Juli 19th,2013. (cited on Page 59)

[AT06] Marios Assiotis and Velin Tzanov. A distributed architecture for mmorpg. In Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games, page 4. ACM, 2006. (cited on Page 2 and 7)

[Bar04] Richard A Bartle. Designing virtual worlds. New Riders, 2004. (cited on Page 5)

[Bar10] Daniel Bartholomew. Sql vs. nosql. Linux Journal, 2010(195):4, 2010. (cited on Page 21)

[BBD+08] HAYES Brian, Thomas Brunschwiler, Heinz Dill, Hanspeter Christ, Babak Falsafi, Markus Fischer, Stella Gatziu Grivas, Claudio Giovanoli, Roger Eric Gisi, Reto Gutmann, et al. Cloud computing. Communications of the ACM, 51(7):9–11, 2008. (cited on Page 13)

[BBG10] Rajkumar Buyya, James Broberg, and Andrzej M Goscinski. Cloud computing: Principles and paradigms, volume 87. Wiley. com, 2010. (cited on Page 2 and 3)

[Bok10] Remco Boksebeld. The impact of cloud computing on enterprise architecture and project success. Master's thesis, HOGESCHOOL UTRECHT, Netherlands, November 2010. (cited on Page vii and 14)

[Bre00] Eric A Brewer. Towards robust distributed systems. In PODC, page 7, 2000. (cited on Page 19)

[Bur07] Brendan Burns. Darkstar: The java game server. O'Reilly, 2007. (cited on Page viii, 6, 55, and 56)

[BW09] Tim Blackman and Jim Waldo. Scalable data storage in project darkstar. Technical report, Sun Microsystems, Inc., 2009. (cited on Page 55)

[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), 26(2):4, 2008. (cited on Page 6, 16, and 25)

[CHL06] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. Game traffic analysis: an mmorpg perspective. Computer Networks, 50(16):3002–3023, 2006. (cited on Page 7 and 64)

[CKSW02] Sergio Caltagirone, Matthew Keys, Bryan Schlief, 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 6)

[Cod85] Edgar F Codd. Does your dbms run by the rules? Computer World, 21, 1985. (cited on Page 11)

[Cod01] Edgar F Codd. A relational model of data for large shared data banks. In Pioneers and Their Contributions to Software Engineering, pages 61–98. Springer, 2001. (cited on Page 8)

[Dat13a] DataStax. About data consistency in cassandra. Website, September 2013. Available online at http://www.datastax.com/docs/1.0/dml/data_consistency; visited on September 1st,2013. (cited on Page 30 and 32)

[Dat13b] DataStax. About transactions and concurrency control. Website, August 2013. Available online at http://www.datastax.com/documentation/cassandra/1.2/webhelp/index.htmlcassandra/dml/dml_about_transactions_c.htmlconcept_ds_j1p_m5x_zj; visited on August 30th,2013. (cited on Page 30 and 31)

[Dat13c] DataStax. Cassandra client apis. Website, September 2013. Available online at http://www.datastax.com/docs/1.0/dml/about_clients; visited on September 3th,2013. (cited on Page 33 and 34)

[Dat13d] Datastax. Cassandra wiki. Website, August 2013. Available online at http://wiki.apache.org/cassandra/CassandraLimitations; visited on Aug 18th,2013. (cited on Page 48)

[Dat13e] DataStax. Generating tokens. Website, September 2013. Available online at http://www.datastax.com/docs/1.0/cluster_architecture/replication; visited on September 8th,2013. (cited on Page vii, 29, 32, and 33)

[Dat13f] DataStax. Generating tokens. Website, September 2013. Available online at http://www.datastax.com/docs/1.1/initialize/token_generation; visited on September 8th,2013. (cited on Page 28)

[Dat13g] DataStax. Row level isolation. Website, September 2013. Available online at http://www.datastax.com/dev/blog/row-level-isolation; visited on September 8th,2013. (cited on Page 31)

[DGK+09] Alan Demers, Johannes Gehrke, Christoph Koch, Ben Sowell, and Walker White. Database research in computer games. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pages 1011–1014. ACM, 2009. (cited on Page 2)

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In SOSP, volume 7, pages 205–220, 2007. (cited on Page 25)

[DSWM13] Ziqiang Diao, Eike Schallehn, Shuo Wang, and Siba Mohammad. Cloud data management for online games: Potentials and open issues. Datenbank-Spektrum, 2013. (cited on Page vii, ix, 2, 3, 6, 7, 8, 23, and 24)

[DWC10] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: Issues and challenges. In Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on, pages 27–33. Ieee, 2010. (cited on Page 3)

[Eba13] Ebay. Cassandra data modeling best practices, part 1. Website, September 2013. Available online at http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/; visited on September 11th,2013. (cited on Page 45)

[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, pages 19–24. ACM, 2007. (cited on Page 1 and 2)

[Fea10] Dietrich Featherston. Cassandra: Principles and application. University of Illinois, 7, 2010. (cited on Page 28)

[FRS05] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games, pages 1–9. ACM, 2005. (cited on Page 7)

[GK08] Galen Gruman and Eric Knorr. What cloud computing really means. InfoWorld, 37, 2008. (cited on Page 13)

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2):51–59, 2002. (cited on Page vii and 19)

[Hew10] Eben Hewitt. Cassandra: The definitive guide. O'Reilly, 2010. (cited on Page vii, 8, 20, 26, 27, 30, 43, and 44)

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In Pervasive computing and applications (ICPCA), 2011 6th international conference on, pages 363–366. IEEE, 2011. (cited on Page 25)

[HYC04] Gao Huang, Meng Ye, and Long Cheng. Modeling system performance in mmorpg. In Global Telecommunications Conference Workshops, 2004. GlobeCom Workshops 2004. IEEE, pages 512–518. IEEE, 2004. (cited on Page 6)

[int13] Gameindusty international. China's online game market to hit 11.9$ billion this year. Website, August 2013. Available online at http://www.gamesindustry.biz/articles/2013-05-02-chinas-online-game-market-to-hit-USD11-9-billion-this-year; visited on August 14th,2013. (cited on Page 1)

[KLXH04] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, volume 1. IEEE, 2004. (cited on Page 2)

[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 23 and 25)

[LWJ13] LWJGL. Lightweight java game library. Website, July 2013. Available online at http://www.lwjgl.org/; visited on Juli 19th,2013. (cited on Page 59)

[MCO10] Vladimir Mateljan, D Cisic, and D Ogrizovic. Cloud database-as-a-service (daas)-roi. In MIPRO, 2010 Proceedings of the 33rd International Convention, pages 1185–1188. IEEE, 2010. (cited on Page ix, 16, and 22)

[MT05] Einar Mykletun and Gene Tsudik. Incorporating a secure coprocessor in the database-as-a-service model. In Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005, pages 7–pp. IEEE, 2005. (cited on Page 16)

[Muh11] Yousaf Muhammad. Evaluation and Implementation of Distributed NoSQL Database for MMO Gaming Environment. PhD thesis, Uppsala University, 2011. (cited on Page 3)

[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 High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, pages 1–12. IEEE, 2008. (cited on Page 7)

[NPI11] Vlad Nae, Radu Prodan, and Alexandru Iosup. Massively multiplayer online game hosting on cloud resources. Cloud Computing: Principles and Paradigms, pages 491–509, 2011. (cited on Page 7)

[ÖV96] M Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. ACM Computing Surveys (CSUR), 28(1):125–128, 1996. (cited on Page 11)

[Pri08] Dan Pritchett. Base: An acid alternative. Queue, 6(3):48–55, 2008. (cited on Page 19, 20, and 21)

[PS09] Hui Peng and Yanli Sun. The network externality and game-playing time characteristics in different types of mmorpgs. In Wireless Communications, Networking and Mobile Computing, 2009. WiCom'09. 5th International Conference on, pages 1–4. IEEE, 2009. (cited on Page 1)

[Ram09] Raghu Ramakrishnan. Data management in the cloud. In Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on, pages 5–5. IEEE, 2009. (cited on Page 3)

[Spr12] Mark Spreeuwenberg. Cloud Computing. Master thesis, Radboud Univerty, 2012. (cited on Page vii and 15)

[Sto10] Michael Stonebraker. Sql databases v. nosql databases. Commun. ACM, 53(4):10–11, April 2010. (cited on Page 17)

[SULS09] Marc Seeger and S Ultra-Large-Sites. Key-value stores: a practical overview. Computer Science and Media, 2009. (cited on Page 23)

[TB11] Bodgan George Tudorica and Cristian Bucur. A comparison between several nosql databases with comments and notes. In Roedunet International Conference (RoEduNet), 2011 10th, pages 1–5. IEEE, 2011. (cited on Page 17, 18, and 23)

[TèV11] M Tamer èOzsu and Patrick Valduriez. Principles of distributed database systems. Springer, 2011. (cited on Page 11)

[Var08a] Jinesh Varia. Cloud architectures. White Paper of Amazon, jineshvaria. s3. amazonaws. ..., 2008. (cited on Page 14)

[Var08b] Jinesh Varia. Cloud architectures. White Paper of Amazon, jineshvaria. s3. amazonaws. com/public/cloudarchitectures-varia. pdf, 2008. (cited on Page 16)

[Vog09] Werner Vogels. Eventually consistent. Communications of the ACM, 52(1):40–44, 2009. (cited on Page 18 and 22)

[VRMCL08] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. ACM SIGCOMM Computer Communication Review, 39(1):50–55, 2008. (cited on Page 13)

[VVE09] Toby Velte, Anthony Velte, and Robert Elsenpeter. Cloud computing, a practical approach. McGraw-Hill, Inc., 2009. (cited on Page 15)

[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 vii, 6, and 10)

[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, pages 53–58. ACM, 2008. (cited on Page 2, 6, and 10)

[ZPC11] Wei Zhou, Guillaume Pierre, and Chi-Hung Chi. Cloudtps: Scalable transactions for web applications in the cloud. Services Computing, IEEE Transactions, 5(4), 2011. (cited on Page 18)

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

Magdeburg, den 14. Oktober. 2013