摘 要

随着因特网应用和计算机技术的飞速发展，数据库逐渐成为信息系统的核心部分并广泛应用于企业、金融机构、政府及国防等各个领域。其中，分布式关系型数据库以其低成本、高可靠性等特点成为当前数据库理论与应用领域的研究热点。

通过对分布式理论、关系型数据库理论及相关技术的学习和研究，本文基于Java语言实现了一个分布式的关系型数据库JSQL。JSQL主要包含五大模块，网络模块接受前端应用程序的连接请求，对客服端进行认证和授权，并对连接进行管理，可实现基于Mysql的通信协议，这样便于Mysql的用户方便的迁移到本系统上来；Sql的解析和执行模块接受客户端的SQL请求，然后解析和执行数据库存储的调用，返回执行结果；审计模块是存储和分析所有对数据库的更改情况，以可视化的方式向用户展示；数据库引擎模块利用了orientdb开源的数据库引擎，实现可靠的分布式存储；分布式模块利用hazlcast实现了数据库集群。基于以上功能模块，ＪＳＱＬ具有高可用性，可扩展性，负载均衡等特性，同时从数据库底层考虑了数据库安全审计需求，加入了数据审计图形化界面显示审计结果。

论文对系统进行了功能和性能测试。功能测试结果表明，系统在功能上符合分布式数据库的基本要求，审计系统的功能也达到本论文的要求。论文通过对性能测试结果进行分析，认为系统的性能基本达到本论文的要求。但是本系统对复制SQL语句的支持还不是很完善，最后提出了改进的方案。

关键词：分布式数据库，mysql，安全审计，OLTP，NOSQL

ABSTRACT

Database, operating system and compiler and called the three systems, can be said that the cornerstone of the entire computer software. Which is closer to the application layer database, is a lot of business support. This field after decades of development, There are new developments. From the beginning of the hierarchy database and relational database, to the recent hot Nosql database, and then to the recent Google Spanner and F1 as the representative of the NewSql database.

In the Internet age, the storage and access of massive data becomes the bottleneck of system design and use. For mass data processing, King, divided into two types: online transaction processing (OLTP) and online analytical processing (OLAP).

Relational database is based on the relational model of the database, which by means of aggregation algebra and other mathematical concepts and methods to deal with the database data. Which is the most popular mysql, mysql is an open source relational database, the advantage lies in the open source code, any business and individuals can according to their own needs to modify the source code mysql.

NoSQL database, called Not Only SQL, meaning that when the relational database is used when the relational database, not applicable There is no need to use relational database is not necessary, you can consider the use of more appropriate data storage.

Oracle, mysql and other traditional relational database is very mature and has been large-scale commercial, why use NoSQL database? mainly With the development of the Internet, the amount of data is growing, the performance requirements are getting higher and higher, the traditional database of congenital defects, namely stand-alone (single Library) performance bottlenecks, and difficult to expand. This is a stand-alone single library bottleneck, but difficult to expand, naturally unable to meet the growing mass of data storage And its performance requirements, so there will be a variety of different NoSQL products.

Although in the cloud computing era, the traditional database there are congenital defects, but NoSQL database can not be replaced, NoSQL can only For the traditional data supplement can not be replaced, so to avoid the shortcomings of traditional databases is the current era of large data must be resolved.

In order to solve these problems, such as mysql and other relational database, this article describes how to design and implement a compatible mysql protocol distributed database. He can automatically find the distributed database cluster nodes, Automatically allocate data to support massive data storage. Taking into account the increasingly important security of the database, from time to time the occurrence of database administrators or other attackers malicious changes in the database data, So the database developed by the database from the bottom of the database to join the audit function.

The main work of this paper is as follows:

Based on the realization of java language compatible mysql communication protocol database;

The realization of the database cluster, to achieve the database of high availability, scalability, load balancing and other characteristics;

From the bottom of the database to consider the database security, joined the data audit and other features.

Keywords: mysql, java, mysql, java, mysql

目 录

第一章 绪论

1.1 研究背景和研究意义

互联网诞生以来在全球迅速蔓延。 2017年我国工业和信息化部最新发布的通信业经济运行情况显示， 2月末，我国移动电话用户总数达到13.3亿户，移动互联网用户总数达到11.2亿户，使用手机上网的用户数接近10.6亿户。互联网用户数量还有很大的增长空间，特别是亚洲人口众多的发展中国家。同时，智能手机的革命发展通过移动互联网大大提升用户体验，使移动互联网迅速发展。伴随着互联网的发展出现了各种基于互联网的应用服务，从传统媒体门户网站到BBS以及近年来社会媒体的兴起，电子商务的巨大发展，还有各种各样的移动互联网应用程序。

随着互联网和互联网应用的发展，数据存储的需求不断增长。 IDC报告显示，预计到2020年全球数据总量将超过40ZB，这一数据量是2011年的22倍。在过去几年，全球的数据量以每年百分之58的速度增长，在未来这个速度会更快。如果按照现在存储容量每年百分之40的增长速度计算，到2018年需要存储的数据量甚至会大于存储设备的总容量。

未来是“大数据”时代，这么大的存储需求，给数据存储技术的发展带来了很大的压力。有很多应用基于互联网提供的各种服务正在进入井喷时代的发展，而这些应用，背景的很大一部分面临同样的问题：怎么样以尽可能廉价的方式实现大规模数据存储和查询。如搜索服务，需要存储页面而分析，微博等社交网络需求的实时用户来表达查询的观点，在线旅游需要玩家的信息和操作来存储和查询，银行需要用户帐户信息和用户用于实时存储和查询。这种应用所面临的挑战总结为大数据可用性和可靠性要求高，部分应用需要实时查询响应和高并发性和数据一致性要求。在这样的环境下，分布式数据库近年来也取得了飞速发展。分布式数据库是一种数据库技术和网络技术结合的产品，相对于单节点数据库，分布式数据库容量和可用性具有很大的优势，因此能够更好的应对大规模和超大规模的数据存储需求的应用。分布式数据库系统通常使用大量廉价，独立的计算机系统构建，经济和性能上，可以满足互联网应用程序的对数据大小，可用性和性能的需求。

1.2 国内外研究历史与现状

数据库技术的发展始于20世纪60年代。在没有数据库的情况下，使用计算机存储数据的用户（主要是财务科研单位）以操作系统中的文件的形式存储数据。随着业务的发展，应用程序变得越来越复杂，人们开始需要开发管理数据在通用软件，这促成了数据库的诞生。 20世纪60年代以来，人们探索数据模型实现数据库，后来开发出三大数据库模型：层次数据模型，网格数据模型，关系数据模型，后来也出现在对象数据模型中。其中，可以使用关系数据模型严格的数学理论来描述数据库的组织和操作，具有简单灵活，数据库独立性高的特点特征。从20世纪70年代到90年代，关系数据库理论成熟并得到广泛应用，这个里程碑是1974年，IBM的圣荷西，加利福尼亚研究实验室的D.D. Chamberlin和Ray Boyce开发了SEQUEL结结构化查询语言，后来在1980年更名为SQL。 SQL是数据库中的标准数据查询语言，在1986年，由美国国家标准学会规范SQL，就这样成为了数据库系统的标准语言。 SQL包含三个部分：数据定义语言，数据操作语言，数据控制语言。 SQL是一种全面的通用关系数据语言，可以当它是一种高级的非程序语言，它允许用户在高级数据结构中工作。使用SQL 用户无需知道数据的具体存储方式。在SQL中一个简单的语句实现效果，使用其他编程语言需要很大一部分程序才能实现。另外，SQL通过使用这种语言允许用户掌握这种语言就可以使用这种语言来操作任何一个标准数据库产品。到20世纪90年代，关系数据库标准几乎适用于任何数据存储需求的应用程序。

分布式数据库系统是数据库系统技术与网络技术的结合。分布式数据库系统（DDBS）的研究始于20世纪70年代。但是，分布式数据库的理论与应用成为一个热门话题，是20世纪90年代的事情。 90年代，互联网网络出现爆炸式增长，同时各种应用程序对存储的需求与日俱增。在这样的环境下，分布式数据库系统的研究成为了热门话题，人们探索分布式数据库系统理论和关键技术，并快速应用理论实践，开发了各种商业应用价值的数据库产品。 2002年，Eric Brewer提出了引导分布式数据库研究的重要理论，并且后来证明是正确的，这个理论就是CAP定理，CAP定理的核心观点是：在分布式计算系统中，不可能同时满足以下三点：一致性，可用性，分区容忍性。 CAP理论认为分布式数据库产品的设计必须介于三者之间，其中至少有两个可以满足，不可能同时满足三个。根据CAP理论的指导，有学者认为，基于传统的关系数据库构建分布式数据库，很难满足当前大型数据存储需求的各类互联网应用，如搜索引擎，社交网络等。由于传统的关系数据库非常重视数据一致性，在传统的关系数据库中，交易的四点必须保证要求：ACID（Atomicity，一致性，隔离，持久性）。有部分人认为，根据CAP理论，在

满足强大的一致性之后，也希望获得互联网应用的高可用性性是非常困难的。因此，在二十世纪九十年代末和二十一世纪初，互联网应用大幅增长出现了一些人认为是革命性的分布式数据库概念：NoSQL（not only SQL）。 NoSQL和传统的关系数据库非常的不同，对于一个概念，NoSQL的显着特点是：非关系型，分布式，不提供ACID数据库设计模式。 NoSQL不支持复杂的关系操作，不提供对交易一致性的支持。由于摆脱了必须满足支持一致性的功能要求，根据NoSQL概念设计产品生成的高可扩展性高并发高可用性性支持得到了很大的改善，各种类似的开源或商业的NoSQL产品出现了，并且被大量和被各种互联网应用程序所使用。这些NoSQL产品被应用到新的产品，如博客，论坛，微博等其他社交服务。这些应用程序具有高度可扩展性，高可用性，并发性。高可用性这些功能很重要，因为这些应用压力主要来自大规模数据存储，大量并发访问及时响应， NoSQL拒绝一致性支持是合理的，因为像社交网络应用程序一样用户丢失数据库的后果不会太严重，毕竟，这不会导致用户遭受巨大的损失，如经济利益。作为分布式数据库，NoSQL在过去两年爆发式增长，均受益于互联网高速发展，也是各种应用到“大数据”的发展趋势，CAP理论再一次证明了它的正确性。

毕竟，NoSQL放弃了一致性的支持，这些产品可以应用到请求的一致性不高在应用上，随着互联网的快速发展和“大数据”时代的到来，那些需要关系操作，需要高级一致性并且面临大规模数据存储查询要求的应用，比如电子商务，这样的应用程序不能使用NoSQL产品。所以，目前的分布式数据库开发方向，根据CAP理论，一个是尽可能的减少一致性，提高可用性，另一个方向是需要尝试确保操作之间的关系，一致性的支持，同时利用分布式技术的优势，提供尽量高性能的服务。

1.3 论文的主要工作

通过对分布式理论、关系型数据库理论及相关技术的学习和研究，论文基于Java语言实现了一个分布式的关系型数据库JSQL ，jsql是一个兼容mysqk通信协议的分布式数据库。论文的主要工作包括：

1.利用orientdb开源项目设计和实现了分布式数据库的本地存储。

2.实现了mysql通信协议，使得可以通过mysql客服端来连接jsql分布式数据库，方便mysql用户迁移到本数据库系统。

3.实现了对sql语句的解析和执行。

4.实现了数据库的分布式架构，使得数据可以存储在多台计算机上面。

5.实现了系统的审计系统，使得系统的安全性得到增强。

1.4 本论文的结构安排

第一章作为本论文的绪论，首先介绍了论文的选题背景和本论文进行研究的意义。接着阐述了数据库的发展历史和现状。最后一节里介绍了论文的主要工作。

第二章，主要介绍了本论文相关的理论基础和相关的技术，首先给数据库分类，给出数据库相关的概念，然后对分布式数据库相关的技术进行了介绍，主要包括负载均衡技术，数据分片技术以及数据库高可用技术。本章最后给出了mysql的体系结构，作为本系统的参考架构。

第三章，作为论文的需求分析，本章给出了分布式数据库jsql的实现目标，同时也包括对通信协议的实现和编程模式的分析。

第四章，是系统的设计，包括总体设计和模块划分，本章对系统的总体架构和各个模块的详细架构给出了阐述。系统主要包括数据库模块，集群架构，已经数据审计功能模块。

第五章，是关于系统的具体实现，本章分别从数据库模块，集群架构模块，数据库审计模块详细说明了每个模块的实现。接着就每个关键模块的实现进行了讲解。最后对系统主要功能的流程进行讲解。

第六章里，论文对jsq分布式数据库进行测试，包括功能测试和性能测试。

最后部分，是关于致谢和参考资料。

1.5 本章小结

本章首先阐述了论文的背景，并表明本论文的研究对于该领域的发展具有非常重要的意义。然后介绍本课题的国内外研究历史和发展现状。最后介绍了本文的主要工作和章节部分安排。

第二章 理论基础和实例研究

本节介绍数据库有关的理论知识。从单机版数据库系统到分布式系统系统理论，最后讨论了目前几种成熟的分布式的系统。

2.1 单机数据库系统

单机存储引擎就是哈希表、B树等数据结构在机械磁盘、SSD等持久化介质上的实现。单机存储系统是单机存储引擎的一种封装，对外提供文件、键值、表格或者关系模型。单机存储系统的理论来源于关系数据库。数据库将一个或多个操作组成一组，称作事务，事务必须满足原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation) 以及持久性(Durability), 简称为ACID 特性。多个事务并发执行时，数据库的并发控制管理器必须能够保证多个事务的执行结果不能破坏某种约定，如不能出现事务执行到一半的情况，不能读取到未提交的事务，等等。为了保证持久性，对于数据库的每一个变化都要在磁盘上记录日志，当数据库系统突然发生故障，重启后能够恢复到之前一致的状态。

本章首先介绍了计算机的层级存储架构和磁盘等硬件基础知识，接着介绍主流的单机存储引擎。其中，哈希存储引擎是哈希表的持久化实现，B树存储引擎是B树的持久化实现，而LSM 树(Log Structure Merge Tree) 存储引擎采用批量转储技术来避免磁盘随机写入。最后，介绍关系数据库理论基础，包括事务、并发控制、故障恢复、数据压缩等。

2.1.1 硬件基础

数据库系统离不开底层硬件的支持，了解现在计算机的存储系统和各种存储介质的特点有利于对数据库系统的深入理解，也是开发一个数据库系统必须要学习的基础知识。

2.1.1.1 计算机层次存储架构

计算机数据存储（通常称为存储或存储器）是由用于保留数字数据的计算机组件和记录介质组成的技术。它是计算机的核心功能和基础组件。计算机的中央处理单元（CPU）是通过执行计算来操纵数据的。在实践中，几乎所有的计算机都使用一个存储层级，意思就是越靠近CPU的存储速度越快，但价格也相对来说更加的昂贵，离CPU越远的数据的存取速度更慢，但是价格相对来说更便宜，所

以现在所有的计算机基本都是结合各种存储技术来使得存储性价比更高。通常，易失性存储技术（在断电时丢失数据）被称为内存，而较慢的持久性存储技术被称为外存; 但是，当引用持久存储时，有时也会使用“内存”。

在Von Neumann架构中，CPU由两个主要部分组成：控制单元和算术逻辑单元（ALU）。前者控制CPU和存储器之间的数据流，而后者对数据执行算术和逻辑运算。没有大量的存储器，计算机将仅能够执行固定操作并立即输出结果。它必须重新配置以改变其行为。这对于诸如桌面计算器，数字信号处理器和其他专用设备的设备是可接受的。冯诺依曼机器在存储操作说明和数据的存储器上有所不同。这样的计算机是更多功能的，因为它们不需要为每个新程序重新配置其硬件，而是可以简单地重新编程新的内存指令; 它们也倾向于更简单的设计，因为相对简单的处理器可以在连续计算之间保持状态以建立复杂的程序结果。大多数现代电脑都是冯诺依曼机器。

通常，存储层次越低，带宽越小，其访问延迟越大。这种传统的存储分区到一级，二级，三级和离线存储也是以每比特成本为指导。在当代使用中，“存储器”通常是半导体存储读写随机存取存储器，通常是DRAM（动态RAM）或其他形式的快速但临时存储。“存储”由存储设备及其介质不能直接通过CPU（二级或三级存储）（通常为硬盘驱动器，光盘驱动器以及其他比RAM慢但非易失性的设备可访问）（掉电时保留内容）。历史上，内存被称为核心内存，主内存，真实存储或内部内存。同时，非易失性存储设备被称为辅助存储器，外部存储器或辅助/外围存储器。按存储设备的性格大概可以分成下面几种：

1.主存储(也称为主存储器或内部存储器)，通常简称为存储器，是CPU可以直接访问的一个存储部件。CPU连续读取存储在那里的指令，并根据需要执行。任何积极操作的数据也以均匀的方式存储在那里。如图所示，传统上主存储器还有两个子层，除主要大容量RAM外还有寄存器和缓存。处理器寄存器位于处理器内部。每个寄存器通常保存一个数据字（通常是32位或64位）。 CPU指令指示算术逻辑单元对该数据执行各种计算或其他操作（或借助它）。寄存器是所有形式的计算机数据存储中最快的。处理器缓存是超快速寄存器和较慢主存储器之间的中间阶段。它是为提高计算机的性能而提出的。在主存储器中主动使用的最主要的信息只是在缓存中复制，这个速度更快，但容量要小得多。另一方面，主存储器速度要慢得多，但存储容量比处理器寄存器大得多。多级分层缓存设置也是常用的 - 主缓存最小，速度最快，位于处理器内部; 二级缓存稍大一点。主存储器通过存储器总线直接或间接地连接到中央处理单元。实际上是两条总线（不在图上）：

地址总线和数据总线。CPU首先通过一个地址总线，一个叫做存储器地址的数字来发送一个数字，指示数据的所需位置。然后使用数据总线读取或写入存储单元中的数据。此外，内存管理单元（MMU）是CPU和RAM之间的一个小型设备，重新计算实际的内存地址，例如提供虚拟内存或其他任务的抽象。由于用于主存储的RAM类型是易失性的（在启动时未初始化），因此仅包含此类存储的计算机将不具有读取指令的源，以启动计算机。因此，使用包含小启动程序（BIOS）的非易失性主存储器来引导计算机，即将较大的程序从非易失性副存储读取到RAM并开始执行。用于此目的的非易失性技术称为ROM，用于只读存储器（术语可能有些混乱，因为大多数ROM类型也可以随机访问）。许多类型的“ROM”不是字面上只读的，因为它们的更新是可能的; 然而，它是缓慢的，并且内存必须被大量擦除，才能被重写。一些嵌入式系统直接从ROM（或类似的）运行程序，因为这样的程序很少改变。标准电脑不会在ROM中存储非基本程序，而是使用大容量的二次存储，这是非易失性的，而不是昂贵的。最近，一些使用中的主存储和二级存储分别是指历史上称为二级存储和三级存储的。

2.辅助存储器（也称为外部存储器或辅助存储器）与主存储器不同之处在于CPU不能直接访问。计算机通常使用其输入/输出通道访问辅助存储，并使用主存储器中的中间区域传输所需的数据。当设备掉电时，辅助存储不会丢失数据 - 它是非易失性的。每单位，通常比主要存储器便宜两个数量级。现代计算机系统通常比主存储器具有二个数量级的二次存储，并且数据在较长时间内被保存。在现代计算机中，通常将硬盘驱动器用作辅助存储。访问存储在硬盘上的给定字节的信息所花费的时间通常是千分之几秒或几毫秒。相比之下，访问存储在随机存取存储器中的给定字节的信息所花费的时间以十亿分之一秒或纳秒来测量。这说明了将固态存储器与旋转磁存储设备区分开的重要访问时间差异：硬盘通常比存储器慢约百万倍。旋转光存储设备，如CD和DVD驱动器，访问时间更长。使用磁盘驱动器，一旦磁盘读/写头达到正确的位置，并且感兴趣的数据在其下方旋转，轨道上的后续数据将非常快地访问。为了减少搜索时间和旋转延迟，数据传输到大块连续块中的磁盘或从磁盘传输。当数据驻留在磁盘上时，阻止访问隐藏延迟提供了设计高效外部存储器算法的机会。磁盘上的顺序或块访问比随机访问快几个数量级，并且已经开发了许多复杂的范例来设计基于顺序和块访问的有效算法。减少I / O瓶颈的另一种方法是并行使用多个磁盘，以增加主存储器和辅助存储器之间的带宽。辅助存储技术的一些其他示例是闪存

（例如USB闪存驱动器或密钥），软盘，磁带，纸带，冲孔卡，独立RAM磁盘和Iomega Zip驱动器。辅助存储器通常根据文件系统格式进行格式化，这提供了将数据组织到文件和目录中所必需的抽象，还提供描述特定文件的所有者的附加信息（称为元数据），访问时间，访问权限，和其他信息。大多数计算机操作系统使用虚拟内存的概念，允许利用比系统中物理上可用的更多的主存储容量。当主内存填满时，系统将最少使用的块（页）移动到辅助存储设备（交换文件或页面文件），稍后在需要时重新检索。由于较慢的次要存储需要更多的这些检索，所以整个系统性能的降低越多。

3.三次存储，一个大型磁带库，磁带盒放在前面的架子上，一个机器人手臂在后面移动。图书馆的可见高度约为180厘米。三级存储或三级存储器提供第三级存储。通常，它涉及一种机器人机构，其将根据系统的要求将可移动大容量存储介质安装（插入）和卸载到存储设备中; 此数据在使用前经常复制到二级存储器中。它主要用于归档很少访问的信息，因为它比二级存储慢得多（例如5-60秒对1-10毫秒）。这主要用于非常大的数据存储，无人操作者访问。典型的例子包括磁带库和光学自动点播机。当计算机需要从三级存储中读取信息时，它将首先查阅目录数据库以确定哪个磁带或光盘包含信息。接下来，计算机将指示机器人手臂取出介质并将其放在驱动器中。当计算机读完信息时，机器人手臂将介质返回到库中的位置。三级存储也被称为近线存储，因为它是“靠近在线”。在线，近线和离线存储之间的正式区别是：在线存储立即可用于I / O。近线存储不能立即可用，但可以在没有人为干预的情况下快速上线。离线存储不能立即可用，需要进行人工干预才能上线。例如，永久旋转的硬盘驱动器是在线存储，而自动旋转的驱动器（例如大量空闲磁盘（MAID））则是近线存储。像磁带库一样可以自动加载的磁带盒等可移动媒体是近线存储，而必须手动加载的磁带盒是离线存储的。离线存储是在不受处理单元控制的介质或设备上的计算机数据存储。[7]介质被记录，通常在二级或三级存储设备中，然后被物理地去除或断开。在计算机再次访问之前，必须由操作人员插入或连接。与三级存储不同，如果没有人工交互，则无法访问。离线存储用于传输信息，因为分离的介质可以容易地物理传输。此外，如果灾难（例如火灾）会破坏原始数据，则远程位置的介质可能不受影响，从而实现灾难恢复。离线存储增加了一般信息安全性，因为它在计算机上是物理上不可访问的，并且数据机密性或完整性不会受到基于计算机的攻击技术的影响。此外，如果存储用于存档目的的信息很少被访问，离线存储比三级存储便宜。在现

代个人计算机中，大多数二级和三级存储介质也用于离线存储。光盘和闪存设备是最受欢迎的，并且可移动硬盘驱动器的程度要小得多。在企业用途中，磁带占主导地位。较早的示例是软盘，Zip磁盘或打孔卡。

2.1.1.2 磁盘介绍

磁盘存储[?]（有时也称为驱动器存储器）是最常用的存储介质，其通过对一个或多个旋转盘的表面层的各种电子，磁，光学或机械变化记录数据。磁盘驱动器是实现这样的存储机构的装置。值得注意的是包含不可移动磁盘的硬盘驱动器（HDD），软盘驱动器（FDD）及其可移动软盘以及各种光盘驱动器（ODD）和相关联的光盘介质。。

数字磁盘驱动器是块存储设备。每个磁盘分为逻辑块（扇区的集合）。使用逻辑块地址（LBA）对块进行寻址。从磁盘的读取或写入发生在块的粒度。最初磁盘容量相当低，并且已经通过多种方式之一进行了改进。机械设计和制造的改进允许更小更精确的头，这意味着可以在每个磁盘上存储更多的轨迹。数据压缩方法的进步允许将更多信息存储在每个单独的扇区中。驱动器将数据存储到汽缸，磁头和扇区上。扇区单元是要存储在硬盘驱动器中的最小数据量，每个文件将分配许多扇区单元。 CD中最小的实体称为帧，由33个字节组成，包含六个完整的立体声采样。其余九个字节由八个CIRC纠错字节和一个用于控制和显示的子代码字节组成。信息从计算机处理器发送到BIOS，进入控制数据传输的芯片。然后通过多线连接器将其发送到硬盘驱动器。一旦数据被接收到驱动器的电路板上，它们将被转换并压缩成单个驱动器可用于存储到磁盘本身上的格式。然后将数据传送到控制对驱动器的访问的电路板上的芯片。驱动器被分成存储在一个内部盘的一个侧面上的数据扇区。内部具有两个磁盘的HDD通常将在所有四个表面上存储数据。驱动器上的硬件告诉执行器臂要进入相关轨道的位置，然后将压缩的信息发送到头部，以改变驱动器上每个字节的光学或磁性的物理属性，从而存储信息。文件不以线性方式存储，而是以最快的方式进行检索。

磁盘在驱动器内部发生两种不同的运动。一个是设备内盘的旋转。另一种是当磁道在磁道之间移动时磁头横跨磁盘的磁头移动运动。有两种类型的磁盘旋转方式：恒定的线速度（主要用于光学存储）根据头部的位置改变光盘的旋转速度，恒定的角速度（用于HDD，标准FDD，几个光盘系统和乙烯基音频记录）以一个恒定的速度旋转介质，无论头部位于何处。轨道定位也遵循磁盘存储设备的两种不同的方法。存储设备专注于保存计算机数据，例如HDD，FDD，Iomega zip驱动器，使用同心轨道存储数据。在顺序读取或写入操作期间，在驱动器访问轨道中

的所有扇区之后，它将头部重新定位到下一个轨道。这将导致设备和计算机之间的数据流的瞬间延迟。相比之下，光学音频和视频光盘使用单个螺旋轨道从盘上的最内侧开始并连续流向外边缘。当读取或写入数据时，不需要停止数据流来切换轨迹。这与乙烯基记录类似，除了乙烯基记录在外边缘开始并向中心旋转。

磁盘存取时间由下面几个时间决定：

- 寻道时间
- 旋转延迟
- 命令处理时间
- 结算时间

其中寻道时间和旋转时间占的比例最大，实现存储引擎一定要尽量减少寻道时间和旋转时间。

2.1.2 单机存储引擎

存储引擎是存储系统的发动机，直接决定了存储系统能够提供的性能和功能。存储系统的基本功能包括：增、删、读、改，其中，读取操作又分为随机读取和顺序扫描。哈希存储引擎是哈希表的持久化实现，支持增、删、改，以及随机读取操作，但不支持顺序扫描，对应的存储系统为键值(Key-Value) 存储系统； B 树(B-Tree) 存储引擎是 B 树的持久化实现，不仅支持单条记录的增、删、读、改操作，还支持顺序扫描，对应的存储系统是关系数据库。当然，键值系统也可以通过B树存储引擎实现； LSM 树 (Log-Structured Merge Tree) 存储引擎和B 树存储引擎一样，支待增、删、改、随机读取以及顺序扫描。它通过批抵转储技术规避磁盘随机写入问题，广泛应用于互联网的后台存储系统，例如Google Bigtable 、Google LevelDB 以及Facebook 开源的Cassandra 系统。本节分别以Bitcask 、MySQL InnoDB 以及Google LevelDB 系统为例介绍这三种存储引擎。

2.1.2.1 哈希存储引擎

在计算中，散列表（hash map）是一种实现关联数组抽象数据类型的数据结构，这是一种将键映射到值的结构。散列表使用散列函数将索引计算到可以找到所需值的桶或槽的数组中。理想情况下，哈希函数将每个密钥分配给唯一的桶，但是大多数散列表设计采用不完美的散列函数，这可能导致哈希冲突，其中散列函数为多个密钥生成相同的索引。这种碰撞必须以某种方式适应。在一个精确的散列表中，每个查找的平均成本（指令数）与存储在表中的元素数量无关。许多散列表设计还允许以键值对的任意插入和删除。在许多情况下，散列表比搜索树

或任何其他表查找结构更有效。因此，它们广泛应用于多种计算机软件，特别是关联阵列，数据库索引，高速缓存和集合。

哈希的想法是将数据（关键字/值）分配到一个数组的桶中。给定一个关键字，该算法计算一个索引，建议可以在哪里找到条目：

$$index = f(y,\ array\_size)$$

通常这通过两个步骤完成：

$$hash = hashfunc\Phi key\Psi$$

$$index = \frac{hash}{array\_size}$$

在该方法中，散列与数组大小无关，然后使用模运算符将其缩小为一个索引（一个0和$array_size - 1$之间的数字）。在阵列大小为2的幂的情况下，余数运算减小到位运算，这提高了速度，但是可能会增加不良散列函数的问题。

良好的散列函数和实现算法对于良好的散列表性能至关重要，但可能难以实现。一个基本要求是该函数应该提供散列值的均匀分布。不均匀分布会增加碰撞次数和解决问题的成本。统一性有时难以通过设计来确定，但是可以使用统计测试对经验进行评估，例如，用于离散均匀分布的Pearson的卡方检验。只有在应用程序中出现的表格大小，才需要统一分发。特别地，如果使用精确加倍的动态调整大小和表格尺寸s的一半，那么只有当s是2的幂时，散列函数才需要是均匀的。这里，索引可以被计算为散列函数的一些位的范围。在另一方面，一些散列算法更喜欢有小号是一个素数。所述的模数操作可以提供一些额外的混合; 这对于散列函数很差很有用。对于开放寻址方案，散列函数还应避免聚类，将两个或多个密钥映射到连续的时隙。这样的聚类可能导致查找成本飞涨，即使负载因子低，并且冲突不频繁。声称流行的乘法散列[3]具有特别差的聚类行为。认为加密散列函数可以通过模减或通过位屏蔽来为任何表格尺寸s提供良好的散列函数。如果恶意用户试图通过提交设计为在服务器的哈希表中生成大量冲突的请求来破坏网络服务，那么它们也可能是适当的。然而，也可以通过更便宜的方法（例如对数据应用秘密盐或使用通用散列函数）来避免破坏风险）。加密散列函数的缺点在于计算速度通常较慢，这意味着在不需要任何均匀的情况下，非加密散列函数可能更为可取。

散列表与其他表数据结构的主要优点是速度。当条目数量大时，这个优点更加明显。当可以预先预测最大条目数时，散列表特别有效，因此可以将桶阵列分配一次为最佳大小，而不会调整大小。如果一组键值对是固定的并且提前知道

（因此不允许插入和删除），则可以通过仔细选择散列函数，桶表大小和内部数据结构来减少平均查找成本。特别地，人们可以设计出无碰撞或甚至完美的散列函数。在这种情况下，键不需要存储在表中。尽管散列表上的操作平均需要恒定时间，但是良好的散列函数的成本可以显着高于用于顺序列表或搜索树的查找算法的内部循环。因此，当条目数非常少时，哈希表无效。（但是，在某些情况下，可以通过与密钥一起保存哈希值来缓解计算散列函数的高成本。）对于某些字符串处理应用程序，如拼写检查，散列表可能比尝试，有限自动机或Judy阵列的效率更低。另外，如果没有太多可能的存储密钥，也就是说，如果每个密钥可以由足够少的位表示，那么代替哈希表，可以直接使用密钥作为数组中的索引的价值观。请注意，在这种情况下没有冲突。存储在哈希表中的条目可以有效地枚举（以每个条目的固定成本），但只能以某些伪随机顺序。因此，没有有效的方法来定位密钥最接近给定密钥的条目。以特定顺序列出所有n个条目通常需要一个单独的排序步骤，其成本与每个条目的$\log(n)$成比例。相比之下，有序搜索树的查找和插入成本与$\log(n)$成比例，但是允许以大约相同的成本找到最近的密钥，并且以每个条目的固定成本排序所有条目的枚举。如果没有存储密钥（因为散列函数是无冲突的），则可能没有简单的方法来枚举在任何给定时刻存在于表中的键。虽然每个操作的平均成本是恒定的并且相当小，但单次操作的成本可能相当高。特别地，如果哈希表使用动态调整大小，插入或删除操作可能偶尔需要与入口数成正比。这可能是实时或交互式应用程序中的严重缺点。哈希表通常表现出较差的参考位置，即要访问的数据在内存中随机分布。因为哈希表导致访问模式跳转，这可能会触发导致长时间延迟的微处理器高速缓存未命中。如果表格相对较小并且键是紧凑的，那么使用线性搜索搜索的数组的紧凑数据结构可能会更快。最佳性能点因系统而异。当有许多碰撞时，散列表变得相当低效。虽然非常不均匀的散列分布极有可能不可能出现，但是具有散列函数知识的恶意对手可能能够通过引起过度的冲突来提供信息到产生最坏情况行为的散列，导致非常差的性能。

解决冲突最简单和最常用的方法是链地址法，链地址法的基本思想是，为每个Hash值建立一个单链表，当发生冲突时，将记录插入到链表中。设有8个元素a,b,c,d,e,f,g,h ，采用某种哈希函数得到的地址分别为：0，2，4，1，0，8，7，2，当哈希表长度为 10 时，采用链地址法解决冲突的哈希表如图2-1所示。

2.1.2.2 B树存储引擎

B+树[?]的结构如图2-2所示。B+树是一个N进制树具有可变的孩子节点。B+树由根，内部节点和叶子节点组成。根可以是叶子或具有两个或更多个孩子的节点。

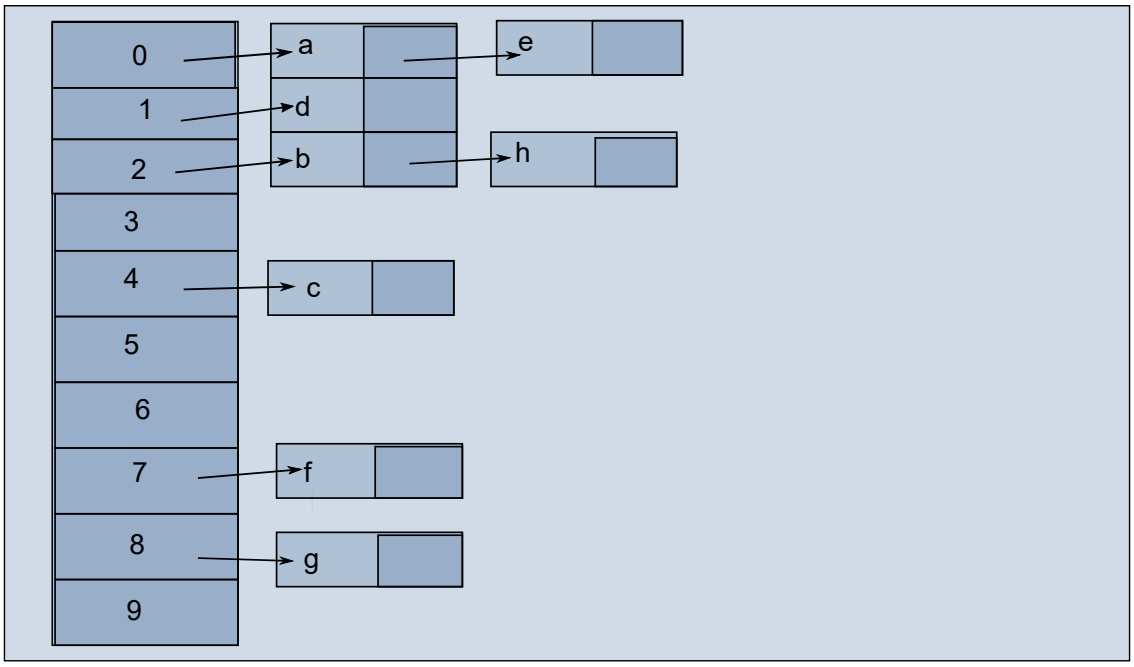


图 2-1 哈希表

B+树可以被视为B树，其中每个节点仅包含键（不是键值对），并且在底部添加有附加级别的链接叶。 B+树的主要价值在于存储用于在面向块的存储环境（特别是文件系统）中有效检索的数据。这主要是因为与二进制搜索树不同， B+树具有非常高的扇出（在节点中的子节点的指针数，通常在100或更多的数量级），这减少了所需的I/O操作的数量在树中找到一个元素。

B+树中的叶子（最底部的索引块）通常在链表中彼此链接; 这使得范围查询或通过块的（有序）迭代更简单和更有效（尽管即使没有这种添加也可以实现上述上限）。这并不显着增加树上的空间消耗或维护。这说明B+树在B树上的显着优点之一; 在B树中，由于不是所有的键都存在于叶中，所以不能构造这样的有序链表。因此，B+树作为数据库系统索引特别有用，其中数据通常驻留在磁盘上，因为它允许B+树实际提供用于容纳数据本身的有效结构

虽然理论上一次性是不必要的，但实际上索引块通常会占用一些额外的空间（例如，叶片中的链表参考）。具有比存储系统的实际块略大的索引块代表着显着的性能下降; 因此，谨慎的方面是错误的。如果B+树的节点被组织为元素的数组，那么插入或删除元素可能需要相当长的时间，因为数组的一半将需要平均移位。为了克服这个问题，节点内的元素可以被组织在一个二叉树或一个B+树而不是数组中。 B+树也可用于存储在RAM中的数据。在这种情况下，块大小的合理选择将是处理器的高速缓存行的大小。通过使用一些压缩技术可以提高B+树的空间效

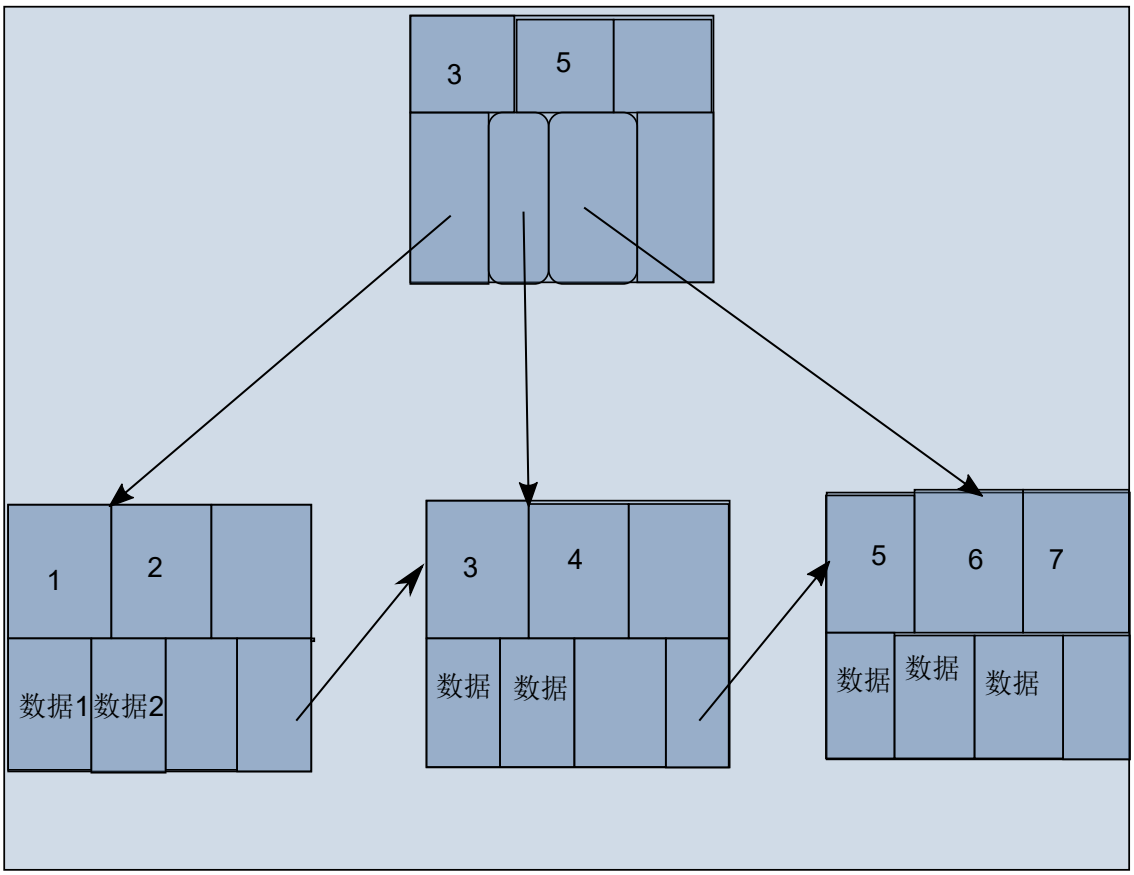


图 2-2 B树结构

率。一种可能性是使用增量编码来压缩存储在每个块中的密钥。对于内部块，可以通过压缩键或指针来实现节省空间。

2.1.2.3 LSM树存储引擎

在计算机科学中，日志结构合并树（或LSM树）[?]是具有性能特征的数据结构，使其具有吸引力，可以提供具有高插入量的文件的索引访问，如事务日志数据。像其他搜索树一样，LSM树保持键值对。LSM树将数据保存在两个或多个单独的结构中，每个结构都针对其各自的底层存储介质进行了优化; 数据在两个结构之间有效地批量同步。

2.1.3 数据模型

如果说存储引擎相当于存储系统的发动机，那么，数据模型就是存储系统的外壳。. 存储系统的数据模型主要包括三类：文件、关系以及随着NoSQL 技术流行起来的键值模型。传统的文件系统和关系数据库系统分别采用文件和关系模型。关系模型描述能力强，产业链完整，是存储系统的业界标准。然而，随着应用在可扩展性、高并发以及性能上提出越来越高的要求，大而全的关系数据库有时显得力不从心，因此，产生了一些新的数据模型，比如键值模型，关系弱化的表格模型，等等。

2.1.3.1 文件模型

文件模型就是提供给用户文件操作接口，用户可以把任何数据存储在文件当中，然后操作文件的接口来获取数据。文件模型相当于其他模型的优点是它的广泛性，任何类型的数据都是二进制的，所以都可以存储在文件里面。

在计算中，使用文件系统或文件系统来控制如何存储和检索数据。没有文件系统，放置在存储介质中的信息将是一大堆数据，无法确定一条信息在哪里停止，下一个开始。通过将数据分成几个部分并给每个部分一个名称，信息很容易被隔离和识别。以名称为基础的纸质信息系统的名称，每组数据称为“ 文件”。用于管理信息组及其名称的结构和逻辑规则称为“文件系统”。有许多不同种类的文件系统。每个人具有不同的结构和逻辑，速度，灵活性，安全性，大小等特点。一些文件系统被设计用于特定应用。例如，ISO 9660文件系统专门为光盘设计。

文件系统可以用于使用不同种类的媒体的许多不同类型的存储设备。目前使用的最常见的存储设备是硬盘驱动器。使用的其他种类的介质包括闪存，磁带和光盘。在某些情况下，例如使用tmpfs，计算机的主存储器（随机存取存储器，RAM）用于创建临时文件系统以供短期使用。一些文件系统在本地数据存储设备

上使用 ; 其他人通过网络协议提供文件访问（例如，NFS，[2] SMB或9P客户端）。一些文件系统是“虚拟的”，这意味着所提供的“文件”（称为虚拟文件）是根据请求（例如procfs）计算的，或仅仅是映射到用作后备存储的不同文件系统。文件系统管理对文件内容和元数据的访问关于这些文件。负责安排存储空间; 关于物理存储介质的可靠性，效率和调优是重要的设计考虑。

2.1.3.2 关系模型

用于数据库管理的关系模型是基于谓词逻辑和集合论的一种数据模型，广泛被使用于关系型数据库之中。最早于1969年由埃德加·科德提出。

关系模型的基本假定是所有数据都表示为数学上的关系，就是说n个集合的笛卡儿积的一个子集，有关这种数据的推理通过二值（就是说没有NULL）的谓词逻辑来进行，这意味着对每个命题都没有两种可能的赋值：要么是真要么是假。数据通过关系演算和关系代数的一种方式来操作。关系模型是采用二维表格结构表达实体类型及实体间联系的数据模型. 关系模型允许设计者通过数据库规范化的提炼，去建立一个信息的一致性的模型。访问计划和其他实现与操作细节由DBMS引擎来处理，而不应该反映在逻辑模型中。这与SQL DBMS普遍的实践是对立的，在它们那里性能调整经常需要改变逻辑模型。

基本的关系建造块是域或者叫数据类型。元组是属性的有序多重集（multiset），属性是域和值的有序对。关系变量（relvar）是域和名字的有序对（序偶）的集合，它充当关系的表头（header）。关系是元组的集合。尽管这些关系概念是数学上的定义的，它们可以宽松的映射到传统数据库概念上。表是关系的公认的可视表示；元组类似于行的概念。关系模型的基本原理是信息原理：所有信息都表示为关系中的数据值。所以，关系变量在设计时刻是相互无关联的；反而，设计者在多个关系变量中使用相同的域，如果一个属性依赖于另一个属性，则通过参照完整性来强制这种依赖性。

关系模型中操作数据的语言是SQL， SQL最初作为关系数据库的标准语言而提出，而在实际上总是违背它。所以SQL DBMS实际上不是真正的RDBMS，并且当前ISO SQL标准不提及关系模型或者使用关系术语或概念。

2.1.3.3 键值模型

键值存储，或键-值数据库，是数据存储模式设计用于存储，检索和管理关联数组，一个数据结构更通常今天被称为字典或散列。词典包含对象或记录的集合，这些对象或记录又包含许多不同的字段，每个都包含数据。使用唯一标识记录的密钥存储和检索这些记录，并用于快速查找数据库中的数据。

键值存储器与更好的关系数据库（RDB）以非常不同的方式工作。 RDB将数据库中的数据结构预定义为一系列包含具有明确定义的数据类型的字段的表。将数据类型暴露给数据库程序允许它应用一些优化。相比之下，键值系统将数据视为单个不透明集合，每个记录可能具有不同的字段。这提供了相当大的灵活性，更接近现代概念，如面向对象编程。因为可选值不像大多数RDB中的占位符表示，所以键值存储通常使用的存储空间更少存储相同的数据库，这可能导致某些工作负载的性能提升。性能，缺乏标准化等问题将关键价值体系限制在利基多年的使用中，但2010年以后云计算的迅速发展已经导致了复兴，成为更广泛的NoSQL运动的一部分。一些图形数据库也是内部的键值存储器，将记录之间的关系（指针）的概念添加为第一类数据类型。

键值存储可以使用从最终一致性到可序列化的一致性模型。有些支持键的排序。有的维护内存中的数据（RAM），而其他则使用固态驱动器或旋转盘。根据DB-Engines排名，Redis是截至2015年8月份最受欢迎的键值数据库实施。键值数据库的另一个例子是Oracle NoSQL数据库。 Oracle NoSQL数据库为应用程序开发人员提供了一个键值范例。每个实体（记录）是一组键值对。键有多个组件，指定为有序列表。主要标识标识实体，并由密钥的主要组成部分组成。后续组件称为次密钥。该组织类似于文件系统中的目录路径规范（例如/Major /minor1 /minor2 ）。键值对的“值”部分只是一个未解释的任意长度的字节串。 Unix系统提供dbm（数据库管理器），它是由Ken Thompson最初编写的一个库。还移植到Windows操作系统平台，通过编程语言（如Perl for Win32）提供。 dbm通过使用单个键（主键）来管理任意数据的关联数组。现代实现包括ndbm，sdbm和gNU dbm。

2.1.3.4 SQL与NoSQL

NoSQL（最初指“非SQL”或“非关系”）数据库提供了一种机制存储和检索是在比所使用的表格关系的其他手段建模的数据的关系数据库。从60年代后期这样的数据库已经存在，但并没有获得“的NoSQL”的绰号，直到人气在早期的二十一世纪的激增，由需求引发的Web 2.0公司如Facebook的，谷歌和亚马逊.com。

NoSQL数据库越来越多地用于大数据和实时Web应用程序。 NoSQL系统有时也被称为“不仅SQL”，强调他们可能支持类似SQL的查询语言。这种方法的动机包括：简单的设计，更简单的“横向”缩放到机器集群（这是关系数据库的一个问题），和更好的控制可用性。NoSQL数据库使用的数据结构（例如键值，宽列，图形或文档）与关系数据库中默认使用的数据结构不同，使NoSQL中的某

些操作更快。给定的NoSQL数据库的特殊适用性取决于它必须解决的问题。有时NoSQL数据库使用的数据结构也被视为比关系数据库表更“灵活”。许多NoSQL存储有利于可用性，分区容限和速度，从而破坏了一致性（在CAP定理的意义上）。更多地采用NoSQL存储的障碍包括使用低级查询语言（而不是SQL，例如缺乏在表中执行临时连接的能力），缺乏标准化接口以及之前对现有关系的巨大投入数据库。大多数NoSQL商店缺乏真正的ACID交易，尽管几个数据库，如MarkLogic，Aerospike，FairCom c-treeACE，Google Spanner（虽然技术上是NewSQL数据库），SymasLMDB和OrientDB使他们成为他们设计的核心。相反，大多数NoSQL数据库提供了“最终一致性”的概念，其中数据库更改“最终”（通常在几毫秒内）传播到所有节点，因此数据查询可能不会立即返回更新的数据，也可能导致读取不准确的数据，这是一个叫做陈旧读取的问题。另外，一些NoSQL系统可能会出现丢失的写入和其他形式的数据丢失。幸运的是，一些NoSQL系统提供了诸如预写日志记录等概念，以避免数据丢失。对于分布式事务处理跨多个数据库，数据一致性是一个更大的挑战，对于NoSQL和关系数据库都是困难的。即使当前的关系数据库“不允许引用完整性约束来跨越数据库”。几个系统都维护了ACID事务和X / Open XA标准，用于分布式事务处理。 Nosql有键值数据库和，图形数据库和文档数据库。

键值（KV）存储使用关联数组（也称为映射或字典）作为其基本数据模型。在该模型中，数据被表示为键值对的集合，使得每个可能的键在集合中最多出现一次。键值模型是最简单的非平凡数据模型之一，更丰富的数据模型通常作为其扩展来实现。键值模型可以扩展到以词典顺序维护键的离散有序模型。这种扩展在计算上是强大的，因为它可以有效地检索选择性关键范围。键值存储可以使用从最终一致性到可序列化的一致性模型。一些数据库支持键的排序。存在各种硬件实现，一些用户将数据保存在存储器（RAM）中，而其他用户使用固态驱动器或旋转盘。示例包括ArangoDB，InfinityDB，Oracle NoSQL数据库，Redis和dbm。

文档数据库的核心概念是文档的概念。虽然每个面向文档的数据库实现在此定义的细节上有所不同，但一般来说，它们都假定文档以某些标准格式或编码封装和编码数据（或信息）。使用的编码包括XML，YAML和JSON以及像BSON这样的二进制形式。文档通过表示该文档的唯一键在数据库中进行处理。面向文档的数据库的其他定义特征之一是，除了键值存储执行的关键查找之外，数据库还提供了一种基于其内容检索文档的API或查询语言。不同的实现提供组织和或分组文档的不同方式：

●集合

- 标签
- 不可见元数据
- 目录层次结构
- 对象

与关系数据库相比，例如，可以将集合视为与记录类似的表和文档。但它们不同：表中的每个记录具有相同的字段序列，而集合中的文档可能具有完全不同的字段。

图形数据库被设计用于数据存储关系数据，其关系被良好地表示为由与它们之间的有限数量的关系互连的元素组成的图形。数据类型可以是社会关系，公共交通链路，路线图或网络拓扑。

2.1.4 事务与并发控制

2.1.4.1 事务

事务象征内执行工作单元的数据库管理系统中独立于其他事务的连贯和可靠的方式（或类似系统）针对数据库，并进行处理。一个事务通常代表数据库中的任何变化。数据库环境中的事务有两个主要目的：为了提供可靠的工作单元，允许从故障中正确恢复，并保持数据库一致，即使在系统故障的情况下，执行停止（完全或部分），数据库上的许多操作仍未完成，状态不清楚。在同时访问数据库的程序之间提供隔离。如果没有提供这种隔离，程序的结果可能是错误的。根据定义，数据库事务必须是原子的，一致的，隔离的和持久的。数据库从业者通常使用缩写ACID来引用数据库事务的这些属性。交易提供了“全或无”的命题，指出在数据库中执行的每个工作单元必须完整或完全没有任何影响。此外，系统必须将每个事务与其他事务隔离开来，结果必须符合数据库中的现有约束，并且成功完成的事务必须写入持久存储。

将数据完整性视为最重要的数据库和其他数据存储通常包括处理事务以维护数据完整性的能力。单个事务由一个或多个独立的工作单元组成，每个读取和/或写入数据库或其他数据存储的信息。当发生这种情况时，确保所有此类处理使数据库或数据存储处于一致状态时，通常很重要。双重会计系统的例子通常说明交易的概念。在双重会计中，每笔借记都需要记录相关信贷。如果有人写了一张100美元的支票来购买杂货，那么一个交易双重会计系统必须记录以下两个条目以涵盖单个交易：借记 100到杂货费用帐户支付100美元到支票账户事务系统将使两个条目都通过，或者两个条目都将失败。通过将多个条目的记录作为原子事务单元处理，系统维护所记录数据的完整性。换句话说，没有人会遇到记录借记的情况，但没有相关的信用记录，反之亦然。

2.1.4.2 并发控制

在信息技术和计算机科学领域，特别是在计算机编程，操作系统，多处理器和数据库领域，并发控制可以确保并发操作的正确结果，同时尽快获得这些结果。计算机系统，软件和硬件都由模块或组件组成。每个组件被设计为正确地操作，即遵守或符合某些一致性规则。当通过消息传递或通过共享访问的数据（在存储器或存储器中）同时进行操作的组件时，某个组件的一致性可能被另一个组件违反。并发控制的一般领域提供了规则，方法，设计方法和理论以保持组件在交互时并发运行的一致性，从而保持整个系统的一致性和正确性。将并发控制引入系统意味着应用操作约束，这通常会导致某些性能下降。操作一致性和正确性应尽可能高效地实现，而不会将性能降低到合理的水平以下。与更简单的顺序算法相比，并发控制可能需要大量额外的复杂性和并发算法的开销。例如，并发控制失败可能导致数据损坏，从而导致读取或写入操作受损。

在并发控制数据库管理系统，其它事务对象和相关的分布式应用程序（例如，网格计算和云计算）确保数据库事务被执行同时在不违反各数据库的数据完整性。因此，并发控制是任何系统中的正确性的基本要素，其中两个数据库事务或更多的数据库事务或多个时间重叠执行，可以访问相同的数据，例如实际上在任何通用数据库系统中。因此，自20世纪70年代初出现数据库系统以来，已经积累了大量相关研究。上述参考文献中提出了一个完善的数据库系统并发控制理论：序列化理论，可以有效地设计和分析并发控制方法和机制。中提出了一种用于抽象数据类型的原子事务并发控制的替代理论），下面没有使用。这个理论更加精细，复杂，范围更广，在数据库文献中比以前的经典理论更少使用。每个理论都有其利弊，重点和见解。在一定程度上它们是互补的，它们的合并可能是有用的。为了确保正确性，一个DBMS通常可以保证只有序列化的交易时间表中产生，除非串行化是故意轻松提高性能，但只有在应用程序的正确性不受损害的情况。为了在失败（中止）事务（由于许多原因总是可能发生）的情况下维护正确性，计划也需要具有可恢复性（从abort）属性。DB MS还保证不会导致提交的事务的影响，并且不会中止任何影响（回滚）交易保留在相关数据库中。整体交易表征通常由以下ACID规则汇总。随着数据库已经成为分布，或在分布式环境中进行合作需要（例如，联邦数据库在90年代初，和云计算目前），并发控制机制有效配置受到格外的关注。

数据库事务（或原子事务）的概念已经发展，以便能够在故障环境中对易于理解的数据库系统行为进行任何处理，并且恢复从崩溃到一个很好理解的数据库状态。数据库事务是一个工作单元，通常在数据库中封装了多个操作（例如，读

取数据库对象，写入，获取锁定等），数据库以及其他系统中支持的抽象。每个交易在该事务中包含哪些程序/代码执行（由事务的程序员通过特殊事务命令确定）方面具有明确的界限。每个数据库事务遵守以下规则（通过在数据库系统中的支持;即，数据库系统旨在保证其运行的事务）：

- 原子性，当事务完成（分别提交或中止）时，所有操作或其任何操作的效果都保留。换句话说，对于外部世界，出现一个承诺的事务（通过其对数据库的影响）是不可分割的（原子的），并且中止的事务根本不影响数据库。所有的操作都是完成交易或没有任何其他操作。
- 一致性 - 每个事务必须使数据库保持一致（正确）状态，即维护数据库的预定完整性规则（数据库对象之间的约束）。事务必须将数据库从一个一致的状态转变为另一个一致状态（但是，交易的程序员有责任确保事务本身是正确的，即正确地执行它打算执行的操作（从应用程序的角度来看视图），而预定义的完整性规则由DBMS实施）。因此，由于数据库只能由事务正常更改，所以数据库的所有状态都是一致的。
- 隔离 - 交易不能相互干扰（作为其执行的最终结果）。此外，通常（根据并发控制方法），不完整的事务的影响对于另一个事务来说甚至不可见。提供隔离是并发控制的主要目标。
- 耐久性 - 成功（提交）事务的影响必须通过崩溃（通常通过将事务的效果及其提交事件记录在非易失性存储器中）来持续存在。

原子交易的概念在这几年中已经扩展到实际实现工作流类型并且不是原子的商业交易。然而，这种增强的事务通常将原子事务用作组件。

如果交易被执行的顺序，即按顺序在时间没有重叠，没有事务并发存在。然而，如果以不受控制的方式允许具有交织操作的并发事务，则可能会发生一些意外的不期望的结果，例如：丢失的更新问题：第二个事务在第一个并发事务写入的第一个值的顶部写入一个数据项（数据）的第二个值，第一个值将丢失到并行运行的其他事务，按其优先级，读取第一个值。读取错误值的事务以不正确的结果结尾。脏读取问题：事务读取由稍后中止的事务写入的值。该值在中止时从数据库中消失，并且不应该被任何事务读取（“脏读”）。阅读交易以不正确的结果结束。不正确的摘要问题：当一个事务对重复数据项的所有实例的值进行摘要时，第二个事务会更新该数据项的某些实例。所得到的摘要并不反映两个事务之间的任何（通常正确的）正确性优先顺序的正确结果（如果一个在另一个之前执行），而是根据更新的时间而确定一些随机结果，以及是否确定更新结果已经包含在摘

要中。大多数高性能事务系统需要并发运行事务以满足其性能要求。因此，没有并发控制，这样的系统既不能提供正确的结果也不能保持数据库的一致性。

存在并发控制的许多方法。它们中的大多数可以在上述主要类别中实现。主要的方法，锁定，通过分配给数据的锁控制对数据的访问。对另一个事务锁定的数据项（数据库对象）的事务的访问可能被阻止（取决于锁类型和访问操作类型），直到锁定释放；串行化，检查计划图中的周期并通过中止来破坏它们；时间戳排序，为事务分配时间戳，并按时间戳顺序控制或检查对数据的访问；承诺排序，控制或检查交易的提交事件的时间顺序与其各自的优先顺序兼容；与上述方法结合使用的其他主要并发控制类型包括：多分支并发控制（MVCC），通过在每次写入对象时生成新版本的数据库对象，并根据调度方式允许事务对每个对象的最后相关版本的读取操作来增加并发性和性能；索引并发控制，将访问操作同步到索引，而不是用户数据。专业方法提供了显着的性能提升。

每个事务都为其访问的数据维护一个私有工作空间，只有在事务提交之后，其更改的数据才会在事务外部变得可见。这种模式在许多情况下提供了不同的并发控制行为，并带来好处。

2.1.5 故障恢复

“数据故障恢复”和“完整性约束”、“并发控制”一样，都是数据库数据保护机制中的一种完整性控制。所有的系统都免不了会发生故障，有可能是硬件失灵，有可能是软件系统崩溃，也有可能是其他外界的原因，比如断电等等。运行的突然中断会使数据库处在一个错误的状态，而且故障排除后没有办法让系统精确地从断点继续执行下去。这就要求DBMS要有一套故障后的数据恢复机构，保证数据库能够回复到一致的、正确地状态去。而“数据故障恢复”正是这样一个机构。

所有的数据恢复的方法都基于数据备份。对于一些相对简单的数据库来说，每隔一段时间做个数据库备份就足够了，但是对于一个繁忙的大型数据库应用系统而言，只有备份是远远不够的，还需要其他方法的配合。恢复机制的核心是保持一个运行日志，记录每个事务的关键操作信息，比如更新操作的数据改前值和改后值。事务顺利执行完毕，称之为提交。发生故障时数据未执行完，恢复时就要滚回事务。滚回就是把做过的更新取消。取消更新的方法就是从日志拿出数据的改前值，写回到数据库里去。提交表示数据库成功进入新的完整状态，滚回意味着把数据库恢复到故障发生前的完整状态。现在举一个银行转帐的例子。从转出账户扣了钱，没来得及写入转入账户就发生故障了。数据库显然处在数据不一

致的状态。恢复时就要从日志上把转出帐户余额的改前值找出来，写回到数据库里面去，转账事务就滚回了。账没转成不要紧，待会儿再转一次，但重要的是数据库能够恢复到故障前的正确状态。这样，数据的完整性才得以保持。在写日志的一刻发生故障，日志的记录并不完整，就不能依靠日志来保持数据库的完整。但是，DB MS有一个控制准则叫做“写日志优先准则”。当数据库执行更新操作时，先把更新信息写进日志里面，然后再更新数据库。所以当写日志系统发生故障时，数据库并没有进行更新，这些数据也就并不需要参与恢复过程。

2.1.5.1 数据库日志

日志在内存里也是有缓存的，这里将其叫做log buffer 。磁盘上的日志文件称为log file。log file一般是追加内容，可以认为是顺序写，顺序写的磁盘IO开销要小于随机写。

Undo日志记录某数据被修改前的值，可以用来在事务失败时进行rollback；Redo日志记录某数据块被修改后的值，可以用来恢复未写入data file的已成功事务更新的数据。下面的示例来自于杨传辉《大数据分布式存储系统原理解析与架构实践》，略作改动。

例如某一事务的事务序号为T1，其对数据X进行修改，设X的原值是5 ，修改后的值为15，那么Undo日志为¡T1, X, 5¿，Redo日志为¡T1, X, 15¿。

也有把undo和redo结合起来的做法，叫做Undo/Redo日志，在这个例子中Undo/Redo日志为¡T1, X, 5, 15¿。当用户生成一个数据库事务时，undo log buffer会记录被修改的数据的原始值，redo会记录被修改的数据的更新后的值。

redo日志应首先持久化在磁盘上，然后事务的操作结果才写入db buffer，（此时，内存中的数据和data file对应的数据不同，我们认为内存中的数据是脏数据），db buffer再选择合适的时机将数据持久化到data file中。这种顺序可以保证在需要故障恢复时恢复最后的修改操作。先持久化日志的策略叫做Write Ahead Log，即预写日志。

在很多系统中，undo日志并非存到日志文件中，而是存放在数据库内部的一个特殊段中。本文中就把这些存储行为都泛化为undo日志存储到undo log file中。

对于某事务T，在log file的记录中必须开始于事务开始标记（比如“start T”），结束于事务结束标记（比如“end T”、”commit T”）。在系统恢复时，如果在log file中某个事务没有事务结束标记，那么需要对这个事务进行undo操作，如果有事务结束标记，则redo。在db buffer中的内容写入磁盘数据库文件之前，应当把log buffer的内容写入磁盘日志文件。

有一个问题，redo log buffer和undo log buffer存储的事务数量是多少，是按照什么规则将日志写入log file？如果存储的事务数量都是1个，也就意味着是将日志立即刷入磁盘，那么数据的一致性很好保证。在执行事T时，突然断电，如果未对磁盘上的redo log file发生追加操作，可以把这个事务T看做未成功。如果redo log file被修改，则认为事务是成功了，重启数据库使用redo log恢复数据到db buffer和data file即可。

如果存储多个的话，其实也挺好解释的。就是db buffer写入data file之前，先把日志写入log file。这种方式可以减少磁盘IO，增加吞吐量。不过，这种方式适用于一致性要求不高的场合。因为如果出现断电等系统故障，log buffer、db buffer中的完成的事务会丢失。以转账为例，如果用户的转账事务在这种情况下丢失了，这意味着在系统恢复后用户需要重新转账。

checkpoint是为了定期将db buffer的内容刷新到data file。当遇到内存不足、 db buffer已满等情况时，需要将db buffer中的内容/部分内容（特别是脏数据）转储到data file中。在转储时，会记录checkpoint发生的”时刻“。在故障回复时候，只需要redo/undo最近的一次checkpoint之后的操作。

在日志文件中的操作记录应该具有幂等性。幂等性，就是说同一个操作执行多次和执行一次，结果是一样的。例如fi$5*1=5*1*1*1$，所以对5的乘1操作具有幂等性。日志文件在故障恢复中，可能会回放多次（比如第一次回放到一半时系统断电了，不得不再重新回放），如果操作记录不满足幂等性，会造成数据错误。

2.2 分布式系统

2.2.1 基本概念

分布式系统（distributed system）是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。因此，网络和分布式系统之间的区别更多的在于高层软件（特别是操作系统），而不是硬件。内聚性是指每一个数据库分布节点高度自治，有本地的数据库管理系统。透明性是指每一个数据库分布节点对用户的应用来说都是透明的，看不出是本地还是远程。在分布式数据库系统中，用户感觉不到数据是分布的，即用户不须知道关系是否分割、有无副本、数据存于哪个站点以及事务在哪个站点上执行等。

2.2.1.1 一致性

在计算机科学中，一致性模型用于分布式系统，如分布式共享内存系统或分布式数据存储（如文件系统，数据库，乐观复制系统或Web缓存）。如果对内存的操作遵循特定的规则，则该系统被称为支持给定的模型。数据一致性模型规定了程序员和系统之间的合同，其中系统保证如果程序员遵循规则，则内存将是一致的，并且内存操作的结果将是可预测的。这不同于缓存一致性，缓存或无缓存的系统中发生的问题是数据相对于所有处理器的一致性。Coherence不处理这一连贯性，因为一致性处理维护一个全局顺序，其中只对单个位置写入一个变量，者所有处理器都看到一个变量。一致性处理与所有处理器相关的多个位置的操作顺序。高级语言（如C ++和Java）通过将内存操作转换为低级操作，以保留内存语义的方式部分维护合同。为了保持合同，编译器可以重新排序一些存储器指令，并且库调用例如pthread_mutex_lock()封装所需的同步。一般来说，通过模型检验验证顺序一致性，即使对于有限状态的高速缓存一致性协议也是一无可估的。一致性模型定义更新的明显顺序和可见性的规则，它是一个具有权衡的连续体。

有两种方法来定义和分类一致性模型; 问题和观点。问题：问题方法描述了定义流程如何发布操作的限制。视图：View方法，它定义可以对进程可见的操作顺序。例如，一致性模型可以定义一个进程不允许发出一个操作，直到所有以前发出的操作完成。不同的一致性模型执行不同的条件如果一个模型需要该模型的所有条件和更多，则一个一致性模型可以被认为比另一个更强。换句话说，具有较少约束的模型被认为是较弱的一致性模型。这些模型定义了硬件需要如何布局和高层次，程序员必须编写代码。所选模型还会影响编译器如何重新排序指令。通常，如果指令之间的控制依赖关系以及如果对同一位置的写入进行排序，则编译器可以根据需要重新排序。然而，使用下面描述的模型，有些可能允许在加载之前的写入被重新排序，而有些可能不会。

严格的一致性，严格的一致性是最强的一致性模型。在这种模式下，所有处理器都需要立即看到任何处理器对变量的写入。严格的模型图和非严格的模型图描述了时间约束 - 即时。可以更好地理解，因为存在全局时钟，其中每个写入应在该时钟周期结束时反映在所有处理器高速缓存中。下一个操作必须在下一个时钟周期内进行。

的顺序一致性模型比严格一致性较弱的存储器模型。对变量的写入不一定要立即被看到，然而，所有处理器必须以相同的顺序看待不同处理器对变量的写入。如Lamport（1979）所定义的，如果“任何执行的结果与所有处理器的操作以某种顺序执行的相同，并且每个处理器的操作出现在此处，则满足顺序一致性按其程

序指定的顺序排列。应保持每个处理器内的程序顺序和处理器之间的操作顺序排序。为了保持处理器之间的顺序执行顺序，所有操作必须看起来相对于每个其他处理器即时或原子地执行。这些操作只需要“出现”才能完成，因为物理上不可能立即发送信息。例如，一旦总线发布了信息，就可以保证所有的处理器都能在同一时刻看到信息。因此，将信息传递到总线完成了对所有处理器的执行，并且似乎已被执行。具有互连网络的无缓存体系结构或高速缓存体系结构不是瞬时的，可能包含处理器和存储器之间的缓慢路径。这些慢路可能导致顺序不一致，因为一些存储器比其他存储器更快地接收广播数据。顺序一致性可以产生非确定性结果。这是因为在程序的不同运行期间，处理器之间的顺序操作顺序可能不同。所有内存操作都需要在程序顺序中进行。线性化（也称为原子一致性）可以被定义为与实时约束的顺序一致性。

因果一致性是通过将事件分类为与因果相关的事件和不是因果关系的事件而将顺序一致性的弱化模型。它定义了只有与因果关系的写入操作需要按照所有进程的相同顺序进行查看。该模型放松了处理器对并发写入的顺序一致性以及与因果关联的写入。如果执行第二次写入的处理器刚刚读取第一次写入，写入变量的两个写可能会依赖于对任何变量的先前写入。这两个写入可能是由相同的处理器或不同的处理器完成的。如顺序一致性，读取不需要立即反映更改，但是，它们需要顺序地反映变量的所有更改。

为了保持数据的一致性，并获得可处理器系统，其中每个处理器都有自己的内存，导出了处理器一致性模型。[5]所有处理器都必须按照一个处理器完成的写入顺序和他们看到不同处理器对相同位置的写入（保持一致性）的顺序保持一致。然而，当写入由不同的处理器到不同的位置时，它们不需要一致。

2.2.2 数据分布

分布式系统如何拆解输入数据，将数据分发到不同的机器中。下面将介绍几种不同的数据分布方式。

2.2.2.1 哈希分布

哈希方式是最常见的数据分布方式,其方法是按照数据的某一特征计算哈希值, 并将哈希值与机器中的机器建立映射关系,从而将不同哈希值的数据分布到不同的机器上。所谓数据特征可以是key-value 系统中的 key,也可以是其他与应用业务逻辑相关的值。

只要哈希散列性比较好，数据就能均匀到分发到不同机器中。同时,需要管理的元信息很少,只需要知道哈希函数和模（一般是机器总数）。但是有个明显的缺

点,扩展性很差。如果我想把集群规模扩大,可能所有的数据需要被重新迁移。工程中,扩展哈希分布数据的系统时,往往使得集群规模成倍扩展,按照数据重新计算哈希,这样原本一台机器上的数据只需迁移一半到另一台对应的机器上即可完成扩展。针对哈希方式扩展性差的问题,一种思路是不再简单的将哈希值与机器做除法取模映射,而是将对应关系作为元数据由专门的元数据服务器管理。访问数据时,首先计算哈希值并查询元数据服务器,获得该哈希值对应的机器。同时,哈希值取模个数往往大于机器个数,这样同一台机器上需要负责多个哈希取模的余数。不过，需要管理的元数据就多了。另外，如果作为哈希函数的key的某个值出现了严重不均，就容易出现“数据倾斜”。比如以用户ID作为特征值，偏偏用户ID＝1的数据特别多，这样就悲剧了。

2.2.2.2 顺序分布

按数据范围分布是另一个常见的数据分布式, 将数据按特征值的值域范围划分为不同的区间,使得集群中每台(组)服务器处理不同区间的数据。对于上面哈希方式某个用户数据特别多我们就可以通过采用数据范围分布解决,动态划分范围空间,实现负载均衡（类似B树）。按数据范围分布数据需要记录所有的数据分布情况。一般的,往往需要使用专门的服务器在内存中维护数据分布信息, 称这种数据的分布信息为一种元数据。实际工程中,一般也不按照某一维度划分数据范围,而是使用全部数据划分范围, 从而避免数据倾斜的问题。使用范围分布数据的方式的最大优点就是可以灵活的根据数据量的具体情况拆分原有数据区间, 拆分后的数据区间可以迁移到其他机器,一旦需要集群完成负载均衡时, 与哈希方式相比非常灵活。另外,当集群需要扩容时,可以随意添加机器,而不限为倍增的方式, 只需将原机器上的部分数据分区迁移到新加入的机器上就可以完成集群扩容。而缺点就是元数据可能会成为瓶颈。

2.2.2.3 一致性哈希

一致性哈希的基本方式是使用一个哈希函数计算数据或数据特征的哈希值,令该哈希函数的输出值域为一个封闭的环,即哈希函数输出的最大值是最小值的前序。将节点随机分布到这个环上,每节点负责处理从自己开始顺时针至下一个节点的全部哈希值域上的数据。

一致性哈希的优点在于可以任意动态添加、删除节点,每次添加、删除一个节点仅影响一致性哈希环上相邻的节点。但也有很明显的缺点,随机分布节点的方式使得很难均匀的分布哈希值域, 尤其在动态增加节点后,即使原先的分布均匀也很

难保证继续均匀,由此带来的另一个较为严重的缺点是 ,当一个节点异常时,该节点的压力全部转移到相邻的一个节点,当加入一个新节点时只能为一个相邻节点分摊力。为此，一种改进是引入“虚节点”的概念。系统初始时就创建许多虚节点, 虚节点的个数一般远大于未来集群中机器的个数, 将虚节点均匀分布到一致性哈希值域环上,其功能与基本一致性哈希算法中的节点相同。为每个节点分配若干虚节点。操作数据时,首先通过数据的哈希值在环上找到对应的虚节点,进而查找元数据找到对应的真实节点。这样,一旦某个节点不可用,该节点将使得多个虚节点不可用,从而使得多个相邻的真实节点负载失效节点的压里。同理,一旦加入一个新节点,可以分配多个虚节点,从而使得新节点可以负载多个原有节点的压力,从全局看,较容易实现扩容时的负载均衡。

一般,我们可以把哈希分布数据方式与按数据量分布数据方式组合使用。按用户 id 的哈希值分数据,当某个用户 id 的数据量特别大时, 该用户的数据始终落在某一台机器上。此时,引入按数据量分布数据的方式 ,统计用户的数据量, 并按某一阈值将用户的数据切为多个均匀的数据段,将这些数据段分布到集群中去。由于大部分用户的数据量不会超过阈值, 所以元数据中仅仅保存超过阈值的用户的数据段分布信息,从而可以控制元数据的规模。

2.2.2.4 负载均衡

在计算中，负载平衡可以改善多个计算资源（如计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器）之间的工作负载分配。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间，并避免任何单一资源的过载。使用负载均衡而不是单个组件的多个组件可以通过冗余来增加可靠性和可用性。负载平衡通常涉及专用软件或硬件，如多层交换机或域名系统服务器进程。负载平衡与通道绑定不同之处在于负载平衡在网络套接字（OSI模型层4）的基础上划分网络接口之间的流量，而通道绑定意味着在每个数据包（OSI模型层3）或数据链路（OSI模型层2）的基础上，使用最短路径桥接协议。

轮询法是负载均衡中最常用的算法，它容易理解也容易实现。轮询法是指负载均衡服务器（load balancer）将客户端请求按顺序轮流分配到后端服务器上，以达到负载均衡的目的。假设现在有6个客户端请求，2台后端服务器。当第一个请求到达负载均衡服务器时，负载均衡服务器会将这个请求分派到后端服务器1；当第二个请求到害时，负载均衡服务器会将这个请求分派到后端服务器2。然后第三个请求到达，由于只有两台后端服务器，故请求3会被分派到后端服务器1。

加权轮询法, 简单的轮询法并不考虑后端机器的性能和负载差异。给性能高、负载低的机器配置较高的权重，让其处理较多的请求；而性能低、负载高的机

器，配置较低的权重，让其处理较少的请求。加权轮询法可以很好地处理这一问题，它将请求顺序且按照权重分派到后端服务器。假设有6个客户端请求，2台后端服务器。后端服务器1被赋予权值5，后端服务器2被赋予赋予权值1。这样一来，客户端请求1，2，3，4，5都被分派到服务器1处理；客户端请求6被分派到服务器2处理。接下来，请求7，8，9，10，11被分派到服务器1，请求12被分派到服务器2，依次类推。

最小连接数法,即使后端机器的性能和负载一样，不同客户端请求复杂度不一样导致处理时间也不一样。最小连接数法根据后端服务器当前的连接数情况，动态地选取其中积压连接数最小的一台服务器来处理当前的请求，尽可能提高后端服务器的利用效率，合理地将请求分流到每一台服务器。为什么根据连接数可以合理地利用服务器处理请求呢？考虑一个客户端请求的处理逻辑较复杂，需要服务器的处理时间较长，由于客户端需要等待服务器的响应，故需要保持与服务器的连接，这样一来，客户端就需要与服务器保持较长时间的连接。

随机法,随机法也很简单，就是随机选择一台后端服务器进行请求的处理。由于每次服务器被挑中的概率都一样，客户端的请求可以被均匀地分派到所有的后端服务器上。

源地址哈希法,源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。如果后端服务器是一缓存系统，当后端服务器增加或者减少时，采用简单的哈希取模的方法，会使得命中率大大降低，这个问题可以采用一致性哈希的方法来解决。

2.2.3 数据复制

2.2.3.1 复制的概述

复制几乎是构成分布式系统，尤其是分布式存储和分布式数据库的关键所在，那么本文就来综合谈论下复制技术。

简单说复制本身可以分为同步复制和异步复制，两者的区别在于前者需要等待所有副本返回写入确认，而后者只需要一个返回确认即可。从用途上，复制可以分为两类，一类用于确保不同副本的表现行为一致，另一类则用于允许不同副本之间的数据差异，先来看看前者。有若干种手段用于确保不同副本之间的状态一致。

第一种叫主从复制。主从之间可以是异步复制，也可以是同步复制。例如MySQL，在默认情况下采用异步复制，异步复制容易引起数据丢失，比如主从结构中，主节点的写入请求还没有复制到从节点就挂了，当从节点被选为新的主节点之后，在这之前写入没有同步的数据就会被丢失。即便采用了同步复制，也只能提供相对较弱的基本保障，考虑如下情形：主接收写入请求然后发到从节点，从节点写入成功后并发送确认给主，如果此时主节点正准备发送确认信息给客户端时挂了，那么客户端就会认为提交失败，可是从节点已经提交成功了，如果这是从节点被提升为主，那么就出现问题了。

在主从复制结构里，异步复制相比同步复制具备更高的吞吐量和更低延迟，因此，结合同步和异步复制是一个常见选项。

比如Kafka，根据它的声称，这是一个CA系统，也就是同时达到数据一致和高可用。Kafka的复制设计同时包含异步复制和同步复制。

另一个例子来自数据库高可用设计，如下是Joyent的PostgreSQL高可用方案manatee，一个主节点带一个同步复制的从节点，以及若干异步复制从节点，当主节点挂了之后，同步复制节点被选举为主节点，异步复制节点选举一个提升为同步复制，而此前挂掉的节点恢复之后首先加入异步复制集群。

第三种则引入分区一致性算法Paxos，又叫复制状态机。复制状态机在数据库开发的很多领域都可以遇到，比如Google Megastore，针对不同分区的每次提交采用复制状态机来确保每个分区的全局事务提交时序； Google Spanner在单分区内也采用了类似的设计。复制状态机主要用于满足两点需求：客户端在面对任何一个副本时都具备完全一致的访问行为；每个副本在执行请求时都需要按照完全一致的顺序来进行。

2.2.3.2 一致性和可用性

在理论计算机科学中，CAP定理，又被称作布鲁尔定理（Brewer's theorem），它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

1.一致性（Consistency)（等同于所有节点访问同一份最新的数据副本）

2.可用性（Availability）（对数据更新具备高可用性）

3.容忍网络分区（Partition tolerance）（以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

根据定理，分布式系统只能满足三项中的两项而不可能满足全部三项。理解CAP理论的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了C性质。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了A性质。除非两个节点可以互相通信，才能既保证C又保证A，这又会导致丧失P性质。

CAP理论在互联网界有着广泛的知名度，知识稍微宽泛一点的工程师都会把其作为衡量系统设计的准则。大家都非常清楚地理解了CAP：任何分布式系统在可用性、一致性、分区容错性方面，不能兼得，最多只能得其二，因此，任何分布式系统的设计只是在三者中的不同取舍而已。

高可用、数据一致是很多系统设计的目标，但是分区又是不可避免的事情：

- CA without P：如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但其实分区不是你想不想的问题，而是始终会存在，因此CA的系统更多的是允许分区后各子系统依然保持CA。
- CP without A：如果不要求A（可用），相当于每个请求都需要在Server之间强一致，而P（分区）会导致同步时间无限延长，如此CP也是可以保证的。很多传统的数据库分布式事务都属于这种模式。
- AP wihtout C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

对于web2.0网站来说，关系数据库的很多主要特性却往往无用武之地。数据库事务一致性需求：很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求并不高。允许实现最终一致性。数据库的写实时性和读实时性需求：对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。对复杂的SQL查询，特别是多表关联查询的需求：任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。

CAP理论经常在不同方面被人误解，对于可用性和一致性的作用范围的误解尤为严重，可能造成不希望看到的结果。如果用户根本获取不到服务，那么其实谈不上C和A之间做取舍，除非把一部分服务放在客户端上运行，即所谓的无连接

操作或称离线模式7。离线模式正变得越来越重要。HTML5的一些特性，特别是客户端持久化存储特性，将会促进离线操作的发展。支持离线模式的系统通常会在C和A中选择A，那么就不得不在长时间处于分区状态后进行恢复。

“一致性的作用范围”其实反映了这样一种观念，即在一定的边界内状态是一致的，但超出了边界就无从谈起。比如在一个主分区内可以保证完备的一致性和可用性，而在分区外服务是不可用的。Paxos算法和原子性多播（atomic multi-cast）系统一般符合这样的场景。像Google的一般做法是将主分区归属在单一个数据中心里面，然后交给Paxos算法去解决跨区域的问题，一方面保证全局协商一致（global consensus）如Chubby，一方面实现高可用的持久性存储如Megastore。

分区期间，独立且能自我保证一致性的节点子集合可以继续执行操作，只是无法保证全局范围的不变性约束不受破坏。数据分片（sharding）就是这样的例子，设计师预先将数据划分到不同的分区节点，分区期间单个数据分片多半可以继续操作。相反，如果被分区的是内在关系密切的状态，或者有某些全局性的不变性约束非保持不可，那么最好的情况是只有分区一侧可以进行操作，最坏情况是操作完全不能进行。

“三选二”的时候取CA而舍P是否合理？已经有研究者指出了其中的要害——怎样才算“舍P”含义并不明确。设计师可以选择不要分区吗？哪怕原来选了CA，当分区出现的时候，你也只能回头重新在C和A之间再选一次。我们最好从概率的角度去理解：选择CA意味着我们假定，分区出现的可能性要比其他的系统性错误（如自然灾难、并发故障）低很多。

这种观点在实际中很有意义，因为某些故障组合可能导致同时丢掉C和A，所以说CAP三个性质都是一个度的问题。实践中，大部分团体认为（位于单一地点的）数据中心内部是没有分区的，因此在单一数据中心之内可以选择CA；CAP理论出现之前，系统都默认这样的设计思路，包括传统数据库在内。然而就算可能性不高，单一数据中心完全有可能出现分区的情况，一旦出现就会动摇以CA为取向的设计基础。最后，考虑到跨区域时出现的高延迟，在数据一致性上让步来换取更好性能的做法相对比较常见。

CAP还有一个方面很多人认识不清，那就是放弃一致性其实有隐藏负担，即需要明确了解系统中存在的不变性约束。满足一致性的系统有一种保持其不变性约束的自然倾向，即便设计师不清楚系统中所有的不变性约束，相当一部分合理的不变性约束会自动地维持下去。相反，当设计师选择可用性的时候，因为需要在分区结束后恢复被破坏的不变性约束，显然必须将各种不变性约束一一列举出

来，可想而知这件工作很有挑战又很容易犯错。放弃一致性为什么难，其核心还是“并发更新问题”，跟多线程编程比顺序编程难的原因是一样的。

2.2.4 分布式协议

2.2.4.1 两阶段提交协议

在计算机网络以及数据库领域内，二阶段提交是指，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。通常，二阶段提交也被称为是一种协议。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为协调者的组件来统一掌控所有节点(称作参与者)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。需要注意的是，二阶段提交(英文缩写：2PC)不应该与并发控制中的二阶段锁(英文缩写：2PL)进行混淆。

二阶段提交算法的成立基于以下假设：该分布式系统中，存在一个节点作为协调者(Coordinator)，其他节点作为参与者(Cohorts)。且节点之间可以进行网络通信。所有节点都采用预写式日志，且日志被写入后即被保持在可靠的存储设备上，即使节点损坏不会导致日志数据的消失。所有节点不会永久性损坏，即使损坏后仍然可以恢复。

第一阶段(提交请求阶段)，协调者节点向所有参与者节点询问是否可以执行提交操作，并开始等待各参与者节点的响应。参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。有时候，第一阶段也被称作投票阶段，即各参与者投票是否要继续接下来的提交操作。

第二阶段(提交执行阶段)，当协调者节点从所有参与者节点获得的相应消息都为”同意”时：协调者节点向所有参与者节点发出”正式提交”的请求。参与者节点正式完成操作，并释放在整个事务期间内占用的资源。参与者节点向协调者节点发送”完成”消息。协调者节点收到所有参与者节点反馈的”完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为”终止”，或者协调者节点在第一阶段的询问超时之前无法获取所参与者节点的响应消息时：协调者节点向所有参与者节点发出”回滚操作”的请求。参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。参与者节点向协调者节点发送”回滚完成”消息。协调者节点收到所有参与者节点反馈的”回滚完成”消息后，取消事务。有时候，第二阶段也被称作完成阶段，因为无论结果怎样，协调者都必须在此阶段结束当前事务。

2.2.4.2 Paxos协议

在过去十年里，Paxos基本成为了分布式领域内一致性协议的代名词。Google的粗粒度锁服务Chubby的设计开发者Burrows曾经说过：“所有一致性协议本质上要么是Paxos要么是其变体”。 Paxos是几乎所有相关课程必讲内容以及很多其它一致性协议的起点，Paxos的提出者LeslieLamport也因其对分布式系统的杰出理论贡献获得了2013年图灵奖。

除了其基础性和重要性之外，Paxos也一直以难以理解闻名。Lamport最初的论文晦涩难懂，很少有人能够透彻理解其精髓，直到之后一系列试图以简明清晰讲解Paxos机制的论文发表后此现象才得以缓解。由此带来的一个副作用是：在根据Paxos原理构造实际可用系统时有一定程度的困难，很多声明基于Paxos原理构造的系统在实现时往往会引入开发者自己的理解，这造成的后果是尽管Paxos在理论上可以证明其正确性，但是实现时经过改造的一致性协议并不能保证这一点。Burrows也曾说过：“Paxos算法描述和真实实现之间存在巨大鸿沟，所以最终系统很可能是基于一个未经证明的一致性协议”。

本节讲述Paxos协议，首先介绍副本状态机模型，之后介绍Paxos的一些基本概念，然后描述Paxos协议本身内容。其实从协议内容本身来看很好理解其运作机制，好像体会不到其难理解性， Paxos的难理解性在于是什么因素导致协议以此种方式呈现以及其正确性证明过程而非最终协议内容本身。

2.3 分布式数据库

2.3.1 分布式数据库概述

分布式数据库是一个数据库，其中存储设备没有全部连接到一个共同的处理器。它可以存储在位于相同物理位置的多台计算机中; 或者可以分散在互连计算机的网络上。与其中处理器紧密耦合并构成单个数据库系统的并行系统不同，分布式数据库系统由松散耦合的站点组成，共享没有物理组件。

2.3.2 数据库中间件

目前数据库中间件有很多，基本这些中间件在下都有了解和使用，各种中间件优缺点及使用场景也都有些心的。

传统的架构模式就是应用连接数据库直接对数据进行访问，这种架构特点就是简单方便。但是随着目前数据量不断的增大我们就遇到了问题: 单个表数据量太大；单个库数据量太大；单台数据量服务器压力很大；读写速度遇到瓶颈。当面临以上问题时，我们会想到的第一种解决方式就是向上扩展(SCALE UP) 。简单来说就是不断增加硬件性能。这种方式只能暂时解决问题，当业务量不断增长时还是解决不了问题。特别是淘宝，facebook，youtube这种业务成线性，甚至指数级上升的情况

此时我们不得不依赖于第二种方式：水平扩展。直接增加机器，把数据库放到不同服务器上，在应用到数据库之间加一个proxy进行路由，这样就可以解决上面的问题了。很多人都会把中间件认为是读写分离，其实读写分离只是中间件可以提供的一种功能，最主要的功能还是在于他可以分库分表。

2.3.2.1 架构

分布式中间一般有3部分组成，第一部分接受前端的连接，第2部分处理sql语句，然后分发到后端不同的数据库，第三个部分就是连接后端不同的数据库，并且做连接池的管理。

2.3.2.2 扩容

当数据量越来越大的时候，单单的增加机器的容量已经不能满足应用的需求了，这个时候我们就需要用分布式的架构，用多台计算机同时来服务前端的连接。

2.3.2.3 讨论

虽然现在分布式中间件很多，大多数都是基于Mysql的中间件，在一些简单的场景来说，这样的方法是可以达到一定的要求的。但是在扩容的容易性上面，还需要更加改进。

2.3.3 分布式数据库实例研究

当今几乎每个大的公司都在在分布式存储系统，了解这些广泛应用的系统，有助于实现自己的数据库。

2.3.3.1 Microsoft SQL Azure

SQL Azure是Azure存储平台的逻辑数据库，物理数据库仍然是SQL Server。一个物理的SQL Server被分成多个逻辑分片(partition)，每一个分片成为一个SQL Azure实例，在分布式系统中也经常被称作子表(tablet)。和大多数分布式存储系统一样，SQL Azure的数据存储三个副本，同一个时刻一个副本为Primary，提供读写服务，其它副本为Secondary，可以提供最终一致性的读服务。每一个SQL Azure实例的允许的最大数据量可以为1GB或者5GB(Web Edition)， 10GB, 20GB, 30GB, 40GB或者50GB(Business Edition)。由于限制了子表最大数据量，Azure存储平台内部不支持子表分裂。，与大多数Web系统架构类似，Azure存储平台大致可以分为四层，从上到下分别为：

- Client Layer：将用户的请求转化为Azure内部的TDS格式流
- Services Layer：相当于网关，相当于普通Web系统的逻辑层
- Platform Layer：存储节点集群，相当于普通Web系统的数据库层
- Infrastructure Layer：硬件和操作系统。Azure使用的硬件为普通PC机

2.3.3.2 Google Spanner

Spanner 是Google的全球级的分布式数据库 (Globally-Distributed Database) 。Spanner的扩展性达到了令人咋舌的全球级，可以扩展到数百万的机器，数已百计的数据中心，上万亿的行。更给力的是，除了夸张的扩展性之外，他还能同时通过同步复制和多版本来满足外部一致性，可用性也是很好的。冲破CAP的枷锁，在三者之间完美平衡

持外部一致的事务。Spanner能做到这些，离不开一个用GPS和原子钟实现的时间API。这个API能将数据中心之间的时间同步精确到10ms以内。因此有几个给力的功能：无锁读事务，原子schema修改，读历史数据无block。

2.3.3.3 OceanBase

OceanBase 是阿里集团研发的可扩展的关系数据库，实现了数千亿条记录、数百 TB 数据上的跨行跨表事务，截止到2012 年8 月，支待了收藏夹、直通车报表、天猫评价等OLTP 和OLAP 在线业务，线上数据批已经超过一于亿条。从模块划分的角度看， OceanBase 可以划分为四个模块：主控服务器RootServer 、更新服务器UpdateServer 、基线数据服务器ChunkServer 以及合并服务器MergeServer。OceanBase 系统内部按照时间线将数据划分为基线数据和增最数据，基线数据是

只读的，所有的修改更新到增噩数据中，系统内部通过合并操作定期将增位数据融合到基线数据中。

OceanBase 支持部署多个机房，每个机房部署一个包含RootServer 、Merge Server 、 ChunkServer 以及UpdateServer 的完整OceanBase 集群，每个集群由各自的RootServer 负责数据划分、负载均衡、集群服务器管理等操作，集群之间数据同步通过主集群的主UpdateServer 往备集群同步增揖更新操作日志实现。客户端配置了多个集群的 RootServer 地址列表，使用者可以设置每个集群的流拯分配比例，客户端根据这个比例将读写操作发往不同的集群。

2.4 本章小结

本章主要介绍了本论文相关的关键的理论和技术，包括单机版数据库和分布式数据库方面的理论和技术，还讨论了现今最成熟的几种分布式数据库。

第三章 系统概述与分析

JSQL作为一个分布式数据库，本章讨论了作为一个分布式系统，它的设计目标。系统本身由很多模块组成。每个模块的实现都要用到不同的技术，本章还分析了jsql实现中每个模块所需要的详细技术，对每个模块的功能进行分析。

3.1 系统概述

JSQL的目标是用java语言开发一个和mysql类似的数据库系统。作为一个数据库系统，他的功能大概有如下：

1.数据定义：系统提供数据定义语言DDL，供用户定义数据库的三级模式结构、两级映像以及完整性约束和保密限制等约束。 DDL主要用于建立、修改数据库的库结构。DDL所描述的库结构仅仅给出了数据库的框架，数据库的框架信息被存放在数据字典中。

2.数据操作：系统提供数据操作语言DML ，供用户实现对数据的追加、删除、更新、查询等操作。

3.数据库的运行管理：数据库的运行管理功能是系统的运行控制、管理功能，包括多用户环境下的并发控制、安全性检查和存取限制控制、完整性检查和执行、运行日志的组织管理、事务的管理和自动恢复，即保证事务的原子性。这些功能保证了数据库系统的正常运行。

4.数据组织、存储与管理：系统要分类组织、存储和管理各种数据，包括数据字典、用户数据、存取路径等，需确定以何种文件结构和存取方式在存储级上组织这些数据，如何实现数据之间的联系。数据组织和存储的基本目标是提高存储空间利用率，选择合适的存取方法提高存取效率。

5.数据库的保护：数据库中的数据是信息社会的战略资源，所以数据的保护至关重要。系统对数据库的保护通过4个方面来实现：数据库的恢复、数据库的并发控制、数据库的完整性控制、数据库安全性控制。系统的其他保护功能还有系统缓冲区的管理以及数据存储的某些自适应调节机制等。

6.数据库的维护：这一部分包括数据库的数据载入、转换、转储、数据库的重组合重构以及性能监控等功能，这些功能分别由各个使用程序来完成。

7.通信：系统具有与操作系统的联机处理、分时系统及远程作业输入的相关接口，负责处理数据的传送。对网络环境下的数据库系统，还应该包括系统与网络中其他软件系统的通信功能以及数据库之间的互操作功能。

3.2 网络实现分析

所有的服务软件都需要实现网络模块，这样才能连接客服端的请求。JSQL用java语言开发，主要用到的是java的网络开发模块。

3.2.1 JAVA网络技术

IO的方式通常分为几种，同步阻塞的BIO、同步非阻塞的NIO、异步非阻塞的AIO[?]。

3.2.1.1 BIO

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

3.2.1.2 NIO

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题：在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。

NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这

个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。

HTTP/1.1出现后，有了Http长连接，这样除了超时和指明特定关闭的http header外，这个链接是一直打开的状态的，这样在NIO处理中可以进一步的进化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的. 当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。

3.2.1.3 AIO

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。在JDK1.7中，这部分内容被称作NIO.2，主要在Java.nio.channels包下增加了下面四个异步通道：

1.AsynchronousSocketChannel

2.AsynchronousServerSocketChannel

3.AsynchronousFileChannel

4.AsynchronousDatagramChannel

其中的read/write方法，会返回一个带回调函数的对象，当执行完读取/写入操作后，直接调用回调函数。

Java对BIO、NIO、AIO的支持：

- Java BIO ：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
- Java NIO ：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

●Java AIO(NIO.2)：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO、NIO、AIO适用场景分析:

●BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

●NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

●AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

另外，I/O属于底层操作，需要操作系统支持，并发也需要操作系统的支持，所以性能方面不同操作系统差异会比较明显。

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。在比较这两个模式之前，我们首先的搞明白几个概念，什么是阻塞和非阻塞，什么是同步和异步,同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

3.2.1.4 Netty

Netty的主要目的是建立基于NIO（或可能的NIO.2）的高性能协议服务器，分离和松散耦合网络和业务逻辑组件。它可能会实现一个广为人知的协议，如HTTP或您自己的特定协议。 Netty的线程模型如如3-1所示。

Netty是一个非阻塞框架。与阻塞IO相比，这导致高吞吐量。 Netty使用事件驱动的应用程序范例，因此数据处理的流水线是一系列事件处理程序。事件和处理程序可以与入站和出站数据流相关联。入站事件可以如下：

●通道激活和停用

●阅读操作事件

●异常事件

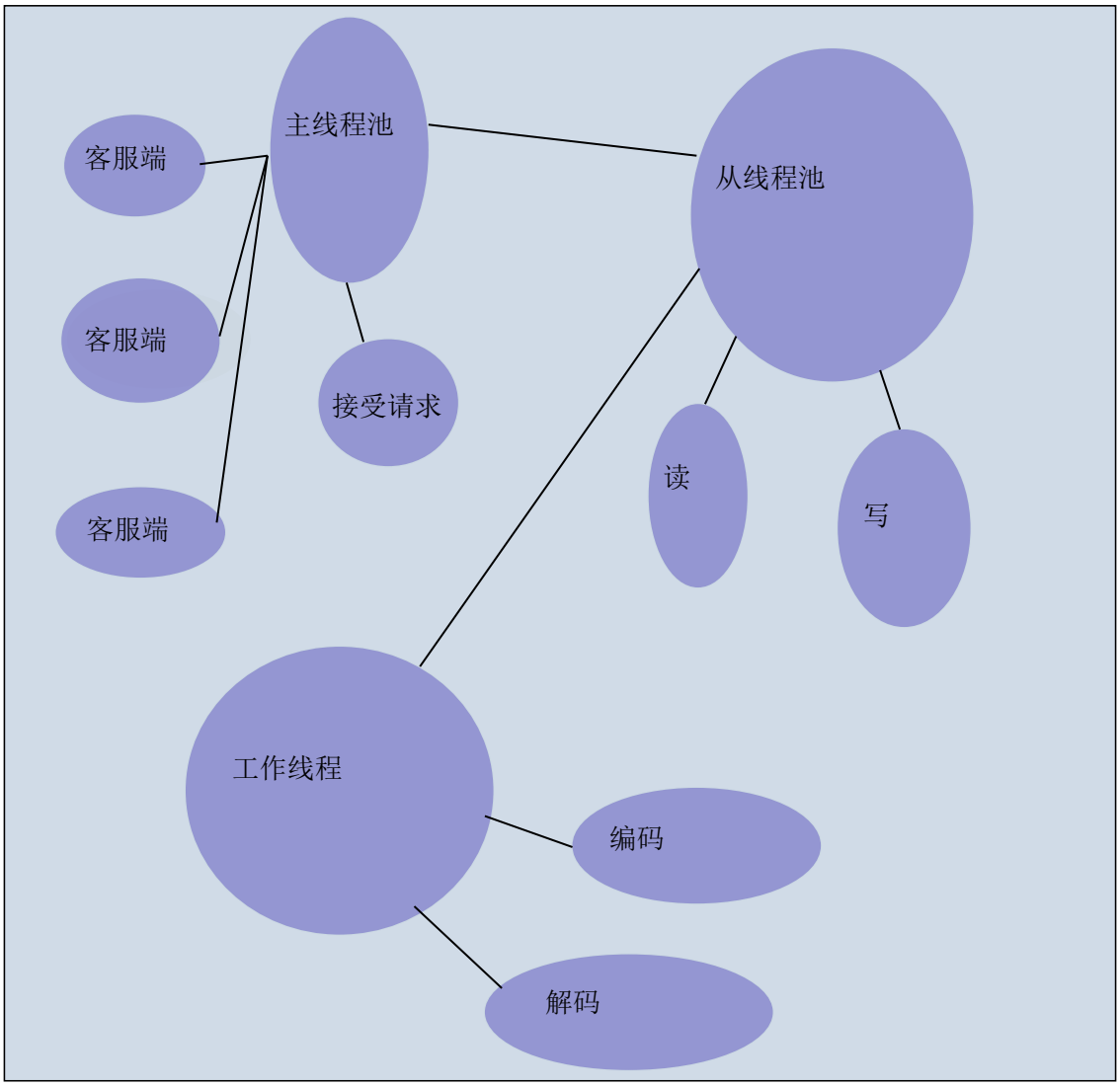


图 3-1 Netty

●用户事件

出站事件更简单，通常与打开/关闭连接和写入/刷新数据有关。

Netty应用程序由几个网络和应用程序逻辑事件及其处理程序组成。通道事件处理程序的基础接口是ChannelHandler及其祖先ChannelOutboundHandler和ChannelInboundHandler。

Netty提供了一个巨大的ChannelHandler实现层次结构。值得注意的只是空的实现的适配器，例如ChannelInboundHandlerAdapter和ChannelOutboundHandlerAdapter。当我们需要处理所有事件的一个子集时，我们可以扩展这些适配器。

此外，还有许多具体协议的实现方式，如HTTP，例如HttpRequestDecoder，HttpResponseEncoder，HttpObjectAggregator。在Netty的Javadoc中熟悉他们是件好事。

当我们使用网络协议时，我们需要执行数据序列化和反序列化。为了这个目的， Netty的介绍的特殊扩展ChannelInboundHandler为解码器，其能够进入的数据进行解码。大多数解码器的基类是ByteToMessageDecoder。对于编码输出数据，Netty具有被称为编码器的ChannelOutboundHandler的扩展。MessageToByteEncoder是大多数编码器实现的基础。我们可以使用编码器和解码器将消息从字节序列转换为Java对象，反之亦然。

因为Netty的这些特效，利用它可以实现一个高性能的网络应用程序，所以本系统的网络模块也是用的Netty来实现的，它直接高并发的客服端访问。

3.3 通信协议分析

每个服务器和客服端通信都要实现自己的通信协议，考虑到mysql使用的广泛性。jsql采用mysql的通信协议。这样就不用自己开发协议了。

Mysql协议的交互过程如图4-3所示。

3.4 SQL实现分析

结构化查询语言是一种特殊目的编程语言，用于数据库中的标准数据查询语言，IBM公司最早使用在其开发的数据库系统中。1986年10月，美国国家标准协会对SQL进行规范后，以此作为关系式数据库管理系统的标准语言（ANSI X3.135-1986）， 1987年得到国际标准组织的支持下成为国际标准。不过各种通行的数据库系统在其实践过程中都对SQL规范作了某些编改和扩充。所以，实际上不同数据库系统之间的SQL不能完全相互通用 , 甚至不同版本间也可能无法互通。

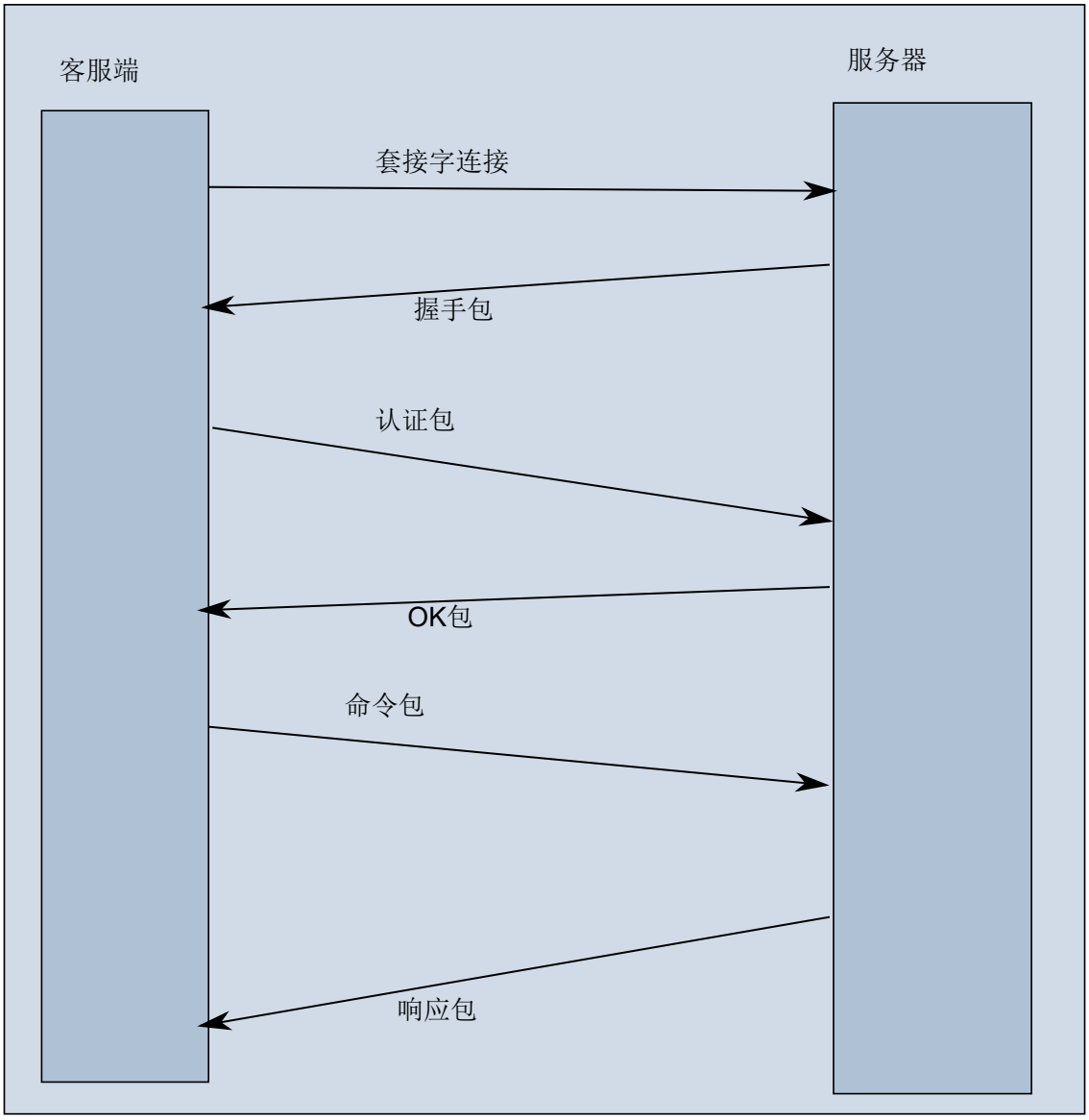


图 3-2 MYSQL协议交互过程

SQL是高级的非过程化编程语言，它允许用户在高层数据结构上工作。它不要求用户指定对数据的存放方法，也不需要用户了解其具体的数据存放方式。而它的界面，能使具有底层结构完全不同的数据库系统和不同数据库之间，使用相同的SQL作为数据的输入与管理。它以记录项目〔records〕的合集（set）〔项集，record set〕作为操纵对象，所有SQL语句接受项集作为输入，回提交的项集作为输出，这种项集特性允许一条SQL语句的输出作为另一条SQL语句的输入，所以SQL语句可以嵌套，这使它拥有极大的灵活性和强大的功能。在多数情况下，在其他编程语言中需要用一大段程序才可实践的一个单独事件，而其在SQL上只需要一个语句就可以被表达出来。这也意味着用SQL可以写出非常复杂的语句，在不特别考虑性能下。 SQL同时也是数据库文件格式的扩展名。 SQL包含四个部分：

1.数据定义语言

2.数据操纵语言

3.数据控制语言

4.事务控制语言

mysql支持的sql语句和标准的sql语句不全一样，它支持下面这几种语句类型：

- 数据定义语句
- 数据操作语句
- 交易和锁定声明
- 复制语句
- 准备的SQL语句语法
- 复合语句语法
- 数据库管理语句
- 效用声明

其中每种语句类型又分为很多种sql语句，所以mysql支持的sql语句非常的多，因为本系统选择了兼容mysql的协议，所以我们也要解析这些不同的sql语句，解析sql语句是一种很麻烦的事情，所以我用了一个成熟的开源的sql解析框架Druid。

Parser 由两部分组成，词法分析和语法分析。当拿到一条形如 select id, name from user 的 SQL 语句后，首先需要解析出每个独立的单词，select，id，name，from，user。这一部分，称为词法分析，也叫作Lexer。通过词法分析后，便要进行语法分析了。经常能听到很多人在调侃自己英文水平很一般时会说： 26个字母我都知道，但是一组合在一起我就不知道是什么意思了。这说明他掌握了词法分

析的技能，却没有掌握语法分析的技能。那么对于 SQL 解析器来说呢，它不仅需要知道每个单词，而且要知道这些单词组合在一起后，表达了什么含义。语法分析的职责就是明确一个语句的语义，表达的是什意思。自然语言和形式语言的一个重要区别是，自然语言的一个语句，可能有多重含义，而形式语言的一个语句，只能有一个语义; 形式语言的语法是人为规定的，有了一定的语法规则，语法解析器就能根据语法规则，解析出一个语句的一个唯一含义。

AST 是 Parser 的产物，语句经过词法分析，语法分析后，它的结构需要以一种计算机能读懂的方式表达出来，最常用的就是抽象语法树。树的概念很接近于一个语句结构的表示，一个语句，我们经常会对它这样看待：它由哪些部分组成？其中一个组成部分又有哪些部分组成？例如一条 select 语句，它由 select 列表、where 子句、排序字段、分组字段等组成，而 select 列表则由一个或多个 select 项组成，where 子句又由一个或者多个 where条件组成。在我们人类的思维中，这种组成结构就是一个总分的逻辑结构，用树来表达，最合适不过。并且对于计算机来说，它显然比人类更擅长处理“树”。

AST 仅仅是语义的表示，但如何对这个语义进行表达，便需要去访问这棵 AST，看它到底表达什么含义。通常遍历语法树，使用 VISITOR 模式去遍历，从根节点开始遍历，一直到最后一个叶子节点，在遍历的过程中，便不断地收集信息到一个上下文中，整个遍历过程完成后，对这棵树所表达的语法含义，已经被保存到上下文了。有时候一次遍历还不够，需要二次遍历。遍历的方式，广度优先的遍历方式是最常见的。

利用Druid可以简化对sql语句的解析过程，同时也让系统更改高效。

3.5 存储引擎分析

存储引擎是存储系统的发动机，直接决定了存储系统能够提供的性能和功能。存储系统的基本功能包括：增、删、读、改，其中，读取操作又分为随机读取和顺序扫描。每种存储引擎底层都基于一种数据结构。比如常用的哈希表结构和B+树结构。本系统用的是一种叫HashTree的数据结构，它的结果如图3-3所示。

哈希树(HashTree)算法就是要提供一种在理论上和实际应用中均能有效地处理冲突的方法。一般的哈希(Hash)算法都是O(1)的，

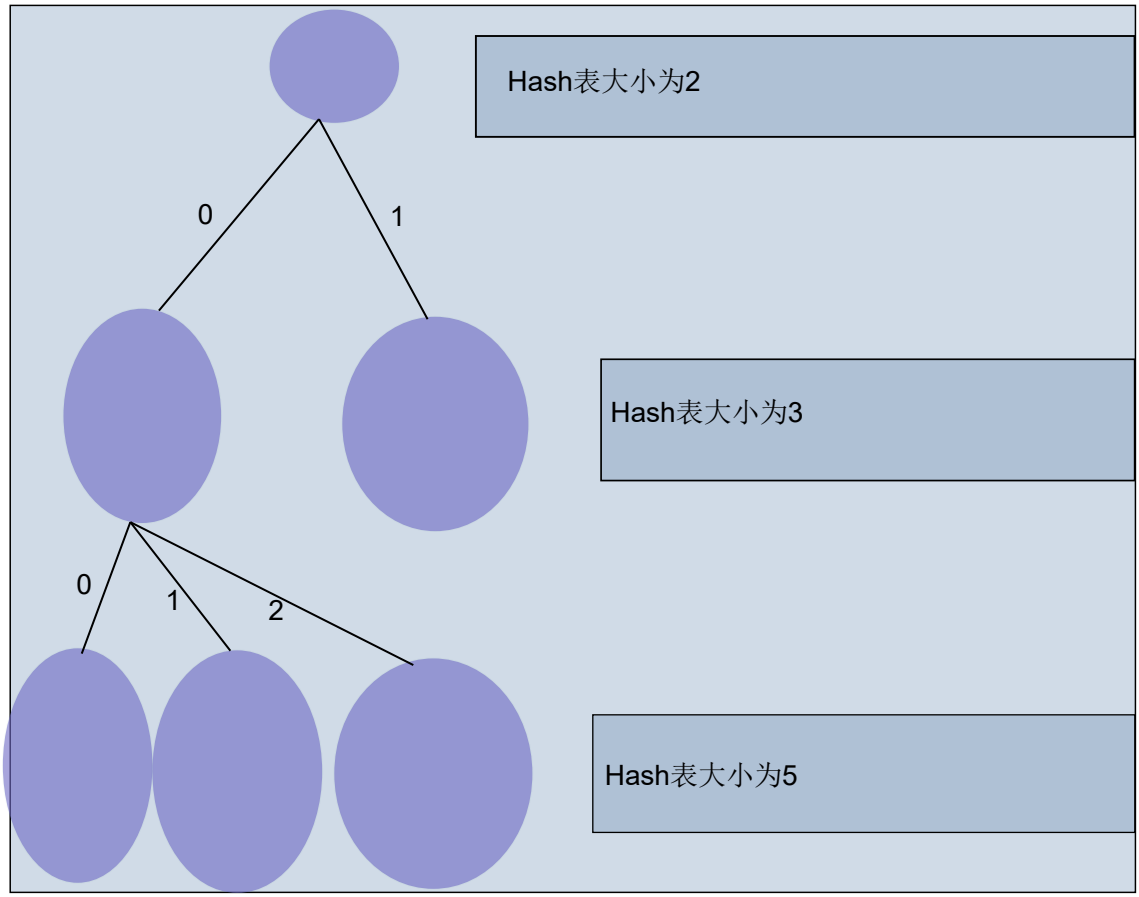


图 3-3 HashTree结构图

3.5.1 哈希树的理论基础

哈希树的理论基础是质数分辨定理，简单地说就是：n个不同的质数可以“分辨”的连续整数的个数和他们的乘积相等。“分辨”就是指这些连续的整数不可能有完全相同的余数序列。

例如：从2起的连续质数，连续10个质数就可以分辨大约$M(10) = 2*3*5*7*11*13*17*19*23*29 = 6464693230$个数，已经超过计算机中常用整数的表达范围。连续100个质数就可以分辨大约$M(100) = 4.711930*10^{219}$。而按照目前的CPU水平，100次取余的整数除法操作几乎不算什么难事。在实际应用中，整体的操作速度往往取决于节点将关键字装载内存的次数和时间。一般来说，装载的时间是由关键字的大小和硬件来决定的；在相同类型关键字和相同硬件条件下，实际的整体操作时间就主要取决于装载的次数。他们之间是一个成正比的关系。

3.5.1.1 优点

结构简单，从哈希树的结构来说，非常的简单。每层节点的子节点个数为连续的质数。子节点可以随时创建。因此哈希树的结构是动态的，也不像某些哈希算法那样需要长时间的初始化过程。哈希树也没有必要为不存在的关键字提前分配空间。需要注意的是哈希树是一个单向增加的结构，即随着所需要存储的数据量增加而增大。即使数据量减少到原来的数量，但是哈希树的总节点数不会减少。这样做的目的是为了避免结构的调整带来的额外消耗。

查找迅速，从算法过程我们可以看出，对于整数，哈希树层级最多能增加到10 。因此最多只需要十次取余和比较操作，就可以知道这个对象是否存在。这个在算法逻辑上决定了哈希树的优越性。一般的树状结构，往往随着层次和层次中节点数的增加而导致更多的比较操作。操作次数可以说无法准确确定上限。而哈希树的查找次数和元素个数没有关系。如果元素的连续关键字总个数在计算机的整数所能表达的最大范围内，那么比较次数就最多不会超过10次，通常低于这个数值。

结构不变，从删除算法中可以看出，哈希树在删除的时候，并不做任何结构调整。这个也是它的一个非常好的优点。常规树结构在增加元素和删除元素的时候都要做一定的结构调整，否则他们将可能退化为链表结构，而导致查找效率的降低。哈希树采取的是一种“见缝插针”的算法，从来不用担心退化的问题，也不必为优化结构而采取额外的操作，因此大大节约了操作时间。

3.5.1.2 缺点

非排序性，哈希树不支持排序，没有顺序特性。如果在此基础上不做任何改进的话并试图通过遍历来实现排序，那么操作效率将远远低于其他类型的数据结构。

关于超长字符串的问题，如果是超长字符串的关键字，该如何处理？若把它们按26进制每一位都转换为数字，则得到的结果太大。我们可以用MD5等消息压缩算法来生成定长的整数。 MD5算法具有以下特点：

1.压缩性：任意长度的数据，算出的MD5值长度都是固定的。

2.容易计算：从原数据计算出MD5值很容易。

3.抗修改性：对原数据进行任何改动，哪怕只修改1个字节，所得到的MD5值都有很大区别。

4.弱抗碰撞：已知原数据和其MD5值，想找到一个具有相同MD5值的数据（即伪造数据）是非常困难的。

5.强抗碰撞：想找到两个不同的数据，使它们具有相同的MD5值，是非常困难的。

对于超长字符串，我们可以用MD5算法生成一个128bit的整数，然后用RadixTree(翻看前面博客)来存储这个大整数，或者使用哈希树来存储，对于这样的大整数，我们不能简单地使用计算机的整数来做除法，而是使用程序模拟人工的除法方式来做除法并获得余数。内存映射文件(Memory-mapped file)，或称“文件映射”、“映射文件”，是一段虚内存逐字节对应于一个文件或类文件的资源，使得应用程序处理映射部分如同访问主内存。

主要用处是增加I/O性能，特别是用于大文件。对于小文件，内存映射文件会导致碎片空间浪费，因为内存映射总是要对其页边界，这起码是4 KiB。因而一个5 KiB文件将会映射占用8 KiB内存，浪费了3 KiB内存。访问内存映射文件比直接文件读写要快几个数量级。内存映射文件可以只加载一部分内容到用户的逻辑内存空间。这对非常大的文件特别有用。使用内存映射文件可以避免颠簸：把相当大的文件直接加载到内存时，由于可用内存不足，使得一边读取文件内存，同时把部分已经加载的文件从内存写入硬盘虚存文件中。内存映射文件由操作系统的内存管理程序负责，因此绕过了硬盘虚存的分页文件（page file）。

在各种数据结构（线性表、树等）中，记录在结构中的相对位置是随机的。因此在机构中查找记录的时需要进行一系列和关键字的比较。这一类的查找方法建立在“比较”的基础上。查找的效率依赖于查找过程中所进行的比较次数。之前我们介绍的各种基于比较的树查找算法，这些查找算法的效率都将随着数据记

录数的增长而下降。仅仅是有的比较慢（时间复杂度为O(n)），有的比较快（时间复杂度是O(logn)）而已。这些查找算法的平均查找长度是在一种比较理想的情况下获得的。在实际应用当中，对数据结构中数据的频繁增加和删除将不断地改变着数据的结构。这些操作将可能导致某些数据结构退化为链表结构，那么其性能必然将下降。为了避免出现这种情况而采取的调整措施，又不可避免的增加了程序的复杂程度以及操作的额外时间。

理想的情况是希望不经过任何比较，一次存取便能得到所查的记录，那就必须在记的存储位置和它的关键字之间建立一个确定的对应关系f，使每个关键字和一个唯一的存储位置相对应。因而在查找时，只要根据这个对应关系f找到给定值K的像f(K)。由此，不需要进行比较便可直接取得所查记录。在此，我们称这个对应关系为哈希（Hash）函数，按这个思想建立的表为哈希表。

在哈希表中对于不同的关键字可能得到同一哈希地址，这种现象称做冲突。在一般情况下，冲突只能尽可能地减少，而不能完全避免。因为哈希函数是从关键字集合到地址集合的映像。通常关键字的集合比较大，它的元素包括所有可能的关键字，而地址集合的元素仅为哈希表中的地址值。在一般情况下，哈希函数是一个压缩映像函数，这就不可避免的要产生冲突。

这样，使用MD5和选用更大的质数相结合的办法。这样就可以使得通过层次比较少的哈希树来获得对关键字区间的完整覆盖。这样就减少了比较操作的次数，并提高整体的工作效率。

3.6 分布式实现分析

在计算中，Hazelcast是基于Java的开源内存数据网格。它也是开发产品的公司的名称。Hazelcast公司由风险投资资助。在一个Hazelcast网格中，数据被一个均匀的节点之间分配计算机集群，从而允许水平缩放的处理和可用存储。备份也分布在节点之间，以防止任何单个节点的故障。Hazelcast通过内存中访问经常使用的数据和可弹性扩展的数据网格，提供中央，可预测的应用程序扩展。这些技术减少了对数据库的查询负载，并提高了速度 Hazelcast平台可以管理许多不同类型应用程序的内存。它提供了一个开放二进制客户端协议，以支持任何二进制编程语言的API。 Hazelcast和开源社区成员已经为包含Java，Scala，.NET Framework，C ++，Python，Node.js和Clojure的编程语言创建了客户端API 。Java和Scala可以用于客户端和嵌入式成员。

利用Hazelcast可以轻松开发出各种的分布式应用程序，而且部署非常简单，只需要嵌入它的二进制包就可以直接使用，避免再部署另外的系统来实现分布式，所以本系统选择了它来实现分布式的功能。

3.7 监控模块分析

本系统实现最简单的监控模块，一个监控模块首先要存储所有的日志数据，而且这个数据不能随意的更改，所以我改了Elasticsearch的源代码，让它来存储所有的sql更新记录，然后用可视化的框架来显示出结果。

Elasticsearch是基于Lucene的搜索引擎。它提供了一个分布式，多租户的全文搜索引擎，具有HTTP Web界面和无模式的JSON文档。Elasticsearch在开发的Java，并发布为开放源代码下的条款Apache许可证。官方客户可以使用Java，.NET（C＃），PHP，Python，Groovy等许多语言。 Elasticsearch是Apache Solr最受欢迎的企业搜索引擎，也是基于Lucene的。 Elasticsearch与名为Logstash的数据收集和日志解析引擎以及名为Kibana的分析和可视化平台一起开发。这三款产品设计用作集成解决方案，被称为“弹性栈”（以前称为“ELK堆栈”）。

3.8 本章小结

JSQL作为一个分布式数据库，本章讨论了作为一个分布式系统，它的设计目标。系统本身由很多模块组成。每个模块的实现都要用到不同的技术，本章还分析了jsql实现中每个模块所需要的详细技术，对每种技术进行了分析。

第四章 系统设计

4.1 总体架构设计

jsql是一个可以替代mysql的分布式java数据库。它的部署场景如题4-1所示。每个数据库都可以接受客服端的命令请求，底层数据会负责数据一致性的实现。

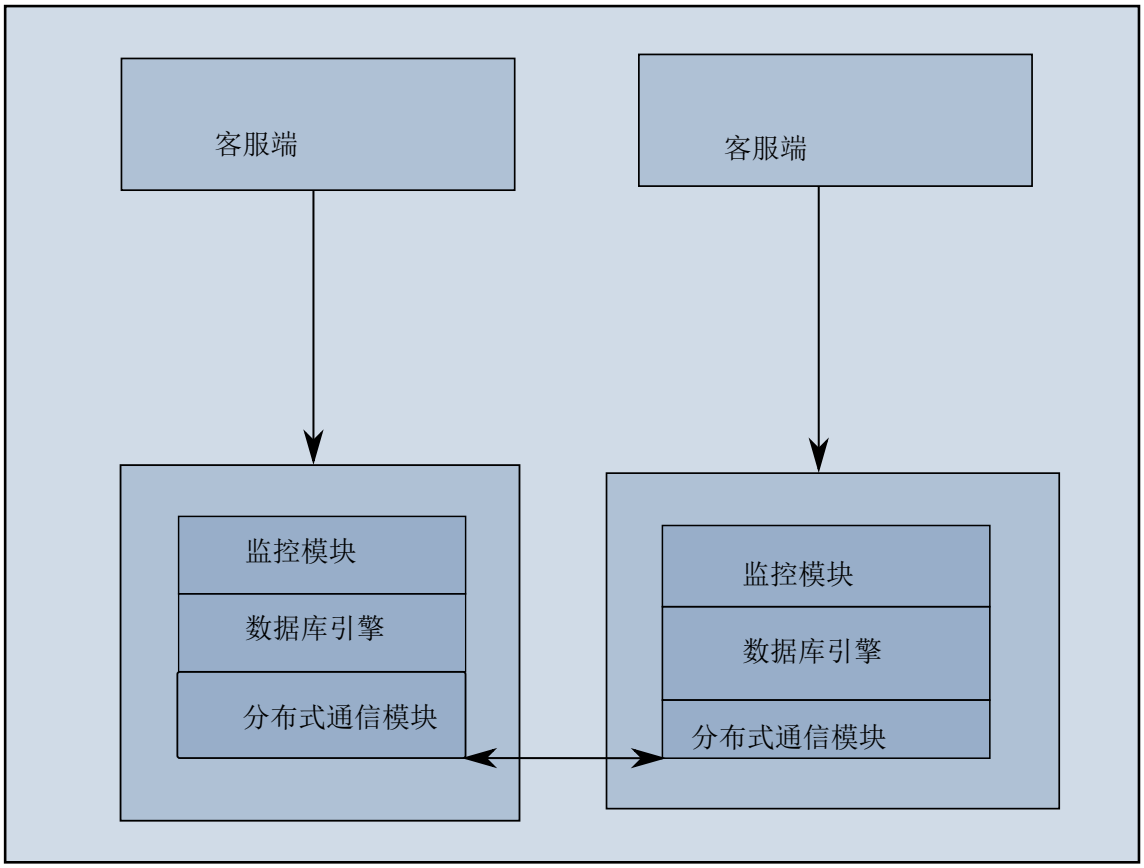


图 4-1 系统部署图

图4-2显示了系统的模块图。一共分为下面几个模块：

1.数据库模块，实现单机版数据库

(a) 网络层，实现前端网络层，接受客服端的连接请求

(b) 协议层，实现mysql的通信协议

(c) SQL层，解析和优化sql的执行

(d) 存储引擎层，基于哈希树的存储引擎

(e) 硬件层，基于内存映射文件的存储系统

2.分布式模块，实现分布式功能

3.监控模块，监控，报警功能

其中每一个模块的具体功能不一样，需要分别设计，下面是每个功能模块的详细设计。

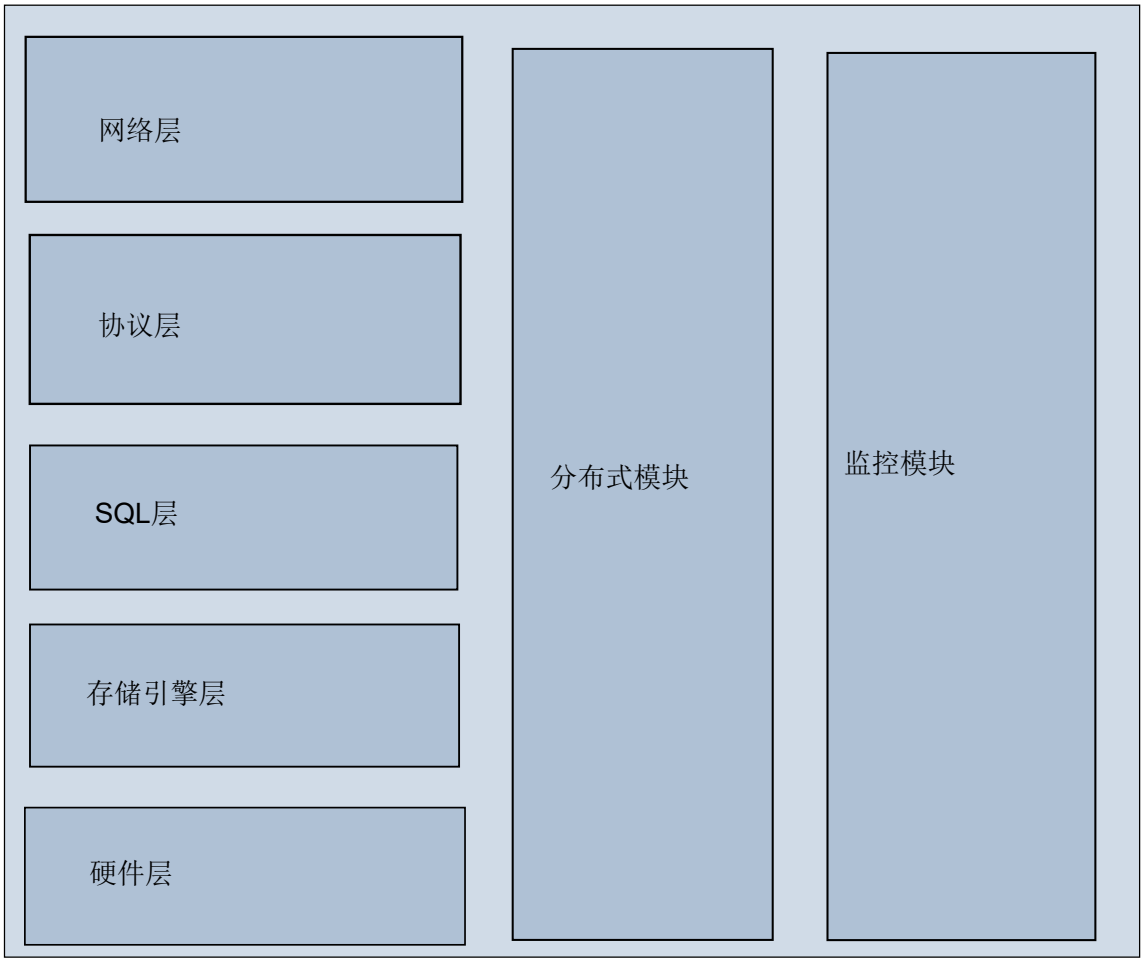


图 4-2 系统总体架构图

4.2 数据库模块详细设计

4.2.1 网络模块设计

所有的服务软件都需要实现网络模块，这样才能连接客服端的请求。JSQL用java语言开发，主要用到的是java的网络开发模块。但是直接用java语言开发网络功能非常的麻烦，所以本文基于Netty来开发直接的网络模块。具体的网络实现，包括线程池的规划都按照Netty的建议来开发。

4.2.2 通信协议设计

每个服务器和客服端通信都要实现自己的通信协议，考虑到mysql使用的广泛性。jsql采用mysql的通信协议。这样就不用自己开发协议了。图4-3显示了 mysql协

议中客服端和服务器之间的交互过程。本文基于这个交互过程来实现自己的通信协议。

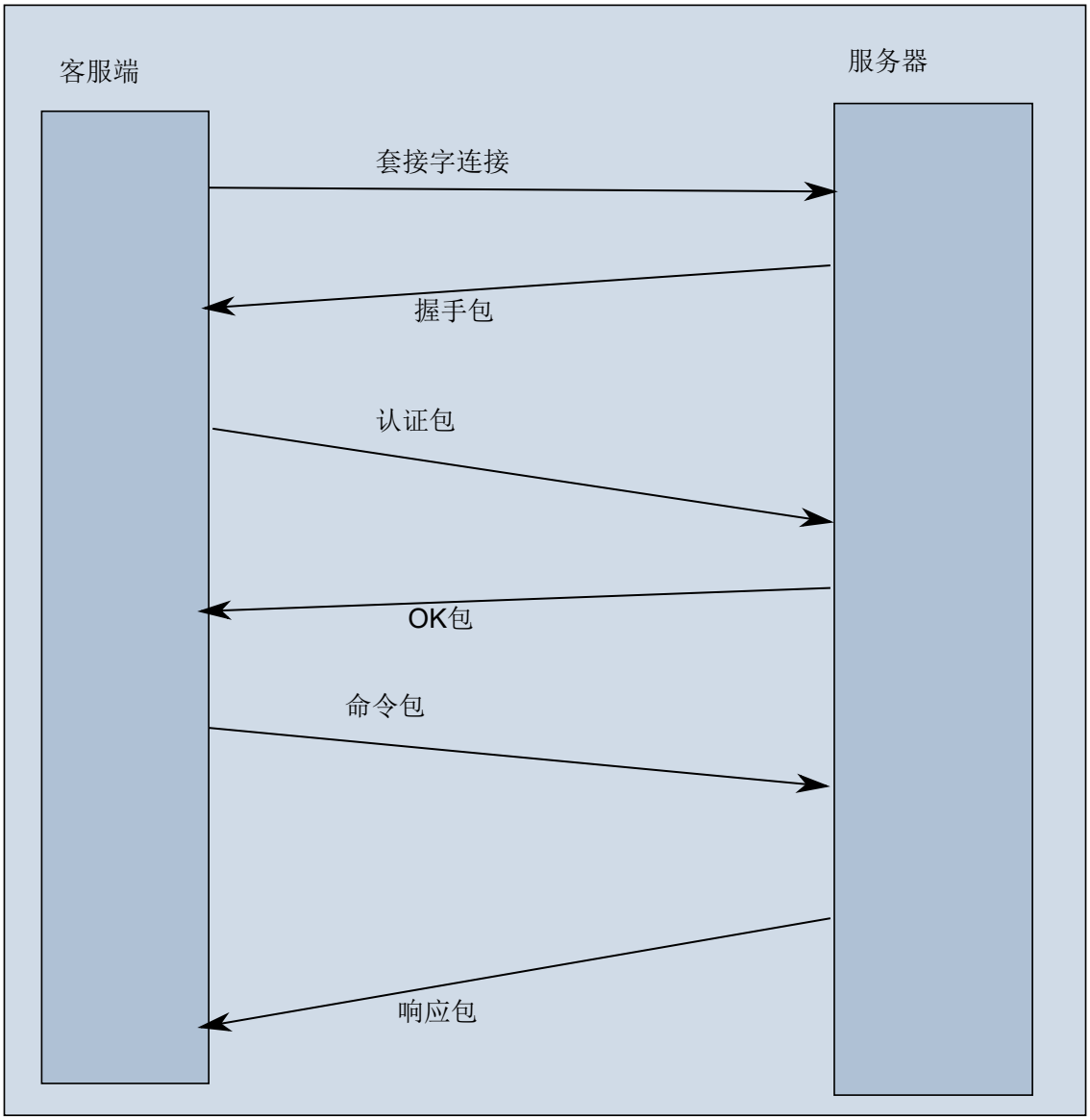


图 4-3 协议交互流程图

4.2.3 SQL引擎设计

mysql的SQL语句非常的复杂，自己解析很浪费时间，本文基于成熟的开源框架Druid 来实现自己的解析模块，解析流程如图4-4所示。

语法树优化

图 4-4 SQL解析流程图

4.2.4 存储引擎设计

存储引擎是存储系统的发动机，直接决定了存储系统能够提供的性能和功能。存储系统的基本功能包括：增、删、读、改，其中，读取操作又分为随机读取和顺序扫描。每种存储引擎底层都基于一种数据结构。比如常用的哈希表结构和B+树结构。本系统用的是一种叫HashTree的数据结构，它的结果如图3-3所示。

4.3 集群架构详细设计

要实现集群功能，肯定要用到网络功能，还要考虑各种不可靠的因素。为了系统的稳定性和可靠性，本文用到了一个开源的分布式的开发框架hazelcast，基于hazelcast的应用程序的结构图如图4-5所示。

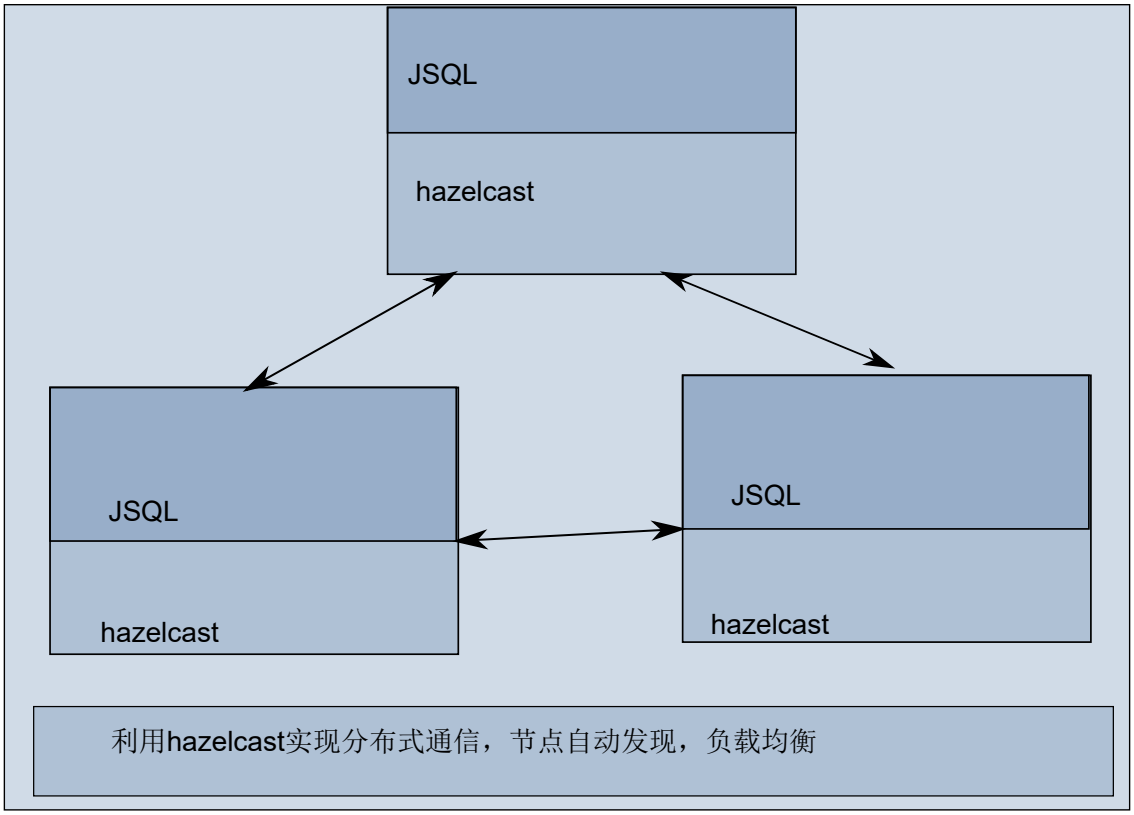


图 4-5 集群架构图

4.4 审计模块详细设计

图4-6显示了审计模块的实现结构图。从上大下一共可以分为下面4个模块：

1.grafana可视化和报警模块，主要用到了开源的图形框架，来显示底层的审计数据，同时也可以让用户设定预警数据，这样以后就可以在一定的情况下向用户报警。

2.elasticsearch模块用来存储监控数据，监控数据为了登陆日志和sql执行日志我们需要更改它的源代码才能用来作为审计数据库，因为审计数据库不能随意的更改和删除，只能查询和增加。

3.JSQL接口层，主要是为上面和下面的各层提供连接功能，为上面层提供数据接口，为下面层提供显示接口，使得更加容易使用。

4.本地日志层，主要是在文件系统中存储所有需要的日志记录，这样可以随时和审计数据库的数据来对比。审计数据的安全和一致性。

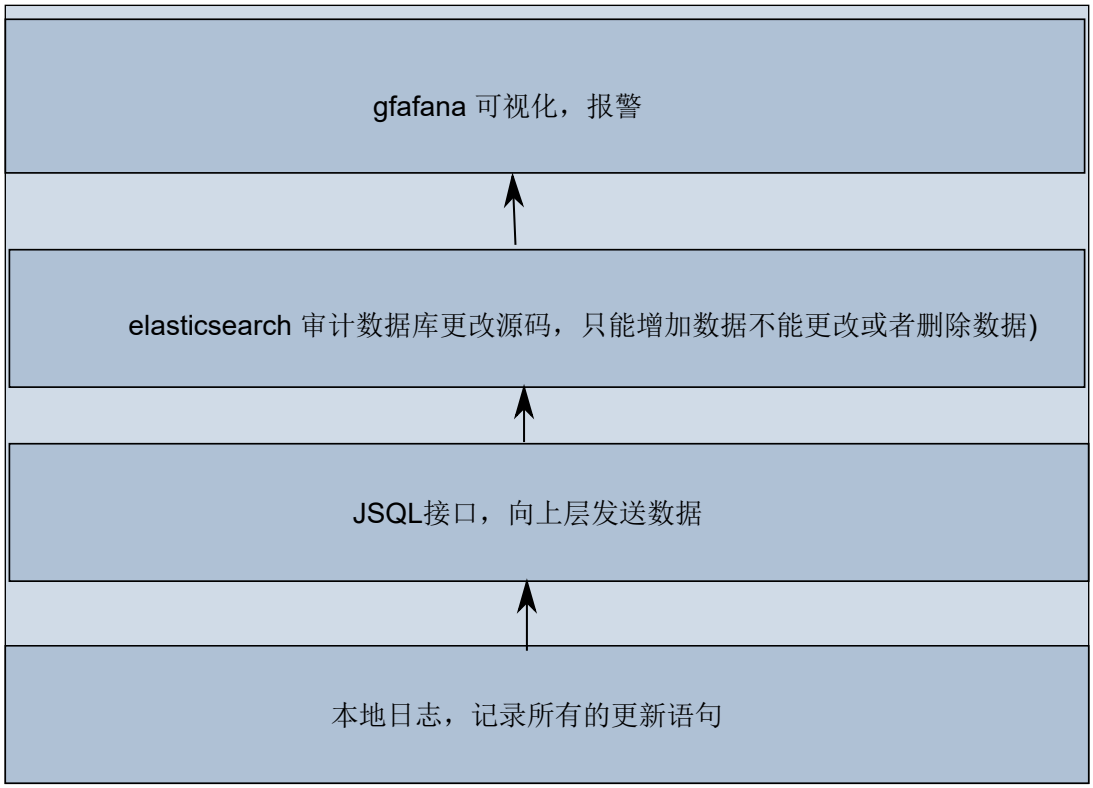


图 4-6 监控模块

4.5 本章小结

本章从总体上设计了系统的部署模式和模块图，对每个模块进行了详细了的模块设计，便于后面的具体实现。

第五章 系统实现

本文前面一章设计了本系统的架构图和各个模块的详细功能，本章给出每个功能模块的具体实现。

5.1 代码规范和总体结构

本数据库系统采用和JAVA语言类似的 Kotlin语言开发。JAVA中的代码以包为组织单位，这样组织代码的好处是逻辑结构分明。表5-1描述了本系统所有的包和每个包的详细功能。 图5-2给出了config包下面每个类的具体作用，这个包主要是

表 5-1 本系统所有包和各个包的作用

包	作用
io.jsql	启动或者关闭服务器
io.jsql.audit	审计监控功能
io.jsql.cache	数据库缓存
io.jsql.config	配置功能，读取外部配置文件
io.jsql.hazelcast	集群功能实现
io.jsql.mysql	实现mysql通信协议
io.jsql.netty	Netty实现高性能网络前端
io.jsql.orientstorage	嵌入式存储引擎模块
io.jsql.shutdown	关闭数据库功能
io.jsql.springaop	面向切面，实现日志等功能
io.jsql.sql	实现SQL解析模块
io.jsql.storage	存储引擎接口
io.jsql.test	测试包
io.jsql.util	所有常用的工具类

表 5-2 config包下面每个类的作用

类	作用
Capabilities	处理能力标识定义
ErrorCode	所有的错误代码
Fields	字段类型及标识定义
Isolations	事务隔离级别定义
Versions	本系统的版本号，通讯包

让用户可以配置数据库的各个方面，比如配置数据库的端口号，配置最大的连接数等等。后面讲详细解释其他包功能具体的实现。

5.2 数据库系统实现

SQL包下面的类主要是作为数据库管理类，让用户启动数据库服务器或者关闭数据库服务器。表5-3描述了SQL包下每个类的作用。

表 5-3 SQL包的每个类的作用

类	功能
ShutdownMain	关闭本机服务器
SpringMain	启动类，通过spring boot启动。 ioc，aop功能

数据库系统主要包括网络模块，SQL解析模块和存储引擎模块，下面分别描述每个模块具体的实现。

5.2.1 网络模块

网络模块主要是用了Netty来实现的，所有和网络有关的代码都在netty包下面。表5-4描述了每个类的功能。 下面的代码是netty包下面主要的代码，用来启

表 5-4 网络模块中各个类的作用

类	作用
ByteToMysqlDecoder	接受客服端的字节消息，转化为mysql的消息格式
ByteToMysqlPacket	包字节格式转化为包对象
MysqlPacketHander	解码器，处理接受到的包对象
NettyServer	前端线程池，接受客服端的请求

动网络服务器，接受用户的连接请求。

```
01 /**
02  *
03  * 服务器
04  */
05 @Service
06 class NettyServer {
07     fun start() {
08         val group = DefaultEventExecutorGroup(8)
09         val bossGroup = NioEventLoopGroup(1)
10         val workerGroup = NioEventLoopGroup()
11         try {
12             val b = ServerBootstrap()
13             b.group(bossGroup, workerGroup)
14                     .channel(NioServerSocketChannel::class.java)
```

```
15                      .option(ChannelOption.SO_BACKLOG, 100)
16                      .childHandler(object : ChannelInitializer<SocketChannel>(){
17                          @Throws(Exception::class)
18                          public override fun initChannel(ch: SocketChannel)
{
19                              val p = ch.pipeline()
20                               //p.addLast(new LoggingHandler(LogLevel.INFO));
21                               p.addLast("idle", IdleStateHandler(10,
5, 0))
22                               p.addLast("decoder", byteToMysqlDecoder)
23                               p.addLast("packet", bytetomysql)
24                               p.addLast(group, "hander", mysqlPacketHander)
25                           }
26                      })
27              val f = b.bind(PORT).sync()
28              // Wait until the server socket is closed.
29              logger.info("server start  complete...................
")
30              Minformation_schama.init_if_notexits()
31              if (config.distributed) {
32                  myHazelcast.inits()
33              }
34              f.channel().closeFuture().sync()
35          } finally {
36              // Shut down all event loops to terminate all threads.
37              bossGroup.shutdownGracefully()
38              workerGroup.shutdownGracefully()
39          }
40      }
41 }
```

除了实现网络模块以外，我们还要实现通信协议，系统采用了和mysql一样的通信协议，在mysql包下面实现了mysql的通信协议。表5-5给出了实现通信协议所

用到的所有的类。

表 5-5 通信协议实现所用到的类

文件	类
BindValue	class BindValue
BindValueUtil	object BindValueUtil
BufferUtil	object BufferUtil
ByteUtil	object ByteUtil
CharsetUtil	object CharsetUtil
MBufferUtil	object MBufferUtil
MySQLMessage	class MySQLMessage
PacketUtil	object PacketUtil
PreparedStatement	class PreparedStatement
SecurityUtil	object SecurityUtil
StreamUtil	object StreamUtil

网络服务器接受到客服端的连接以后，就要解析mysql的通信协议，包每一个包封装到具体的对象里面，表5-6描述了所有的mysql通信协议的封装对象。 mysql里面有很多的包，每个包都需要一个类来实现，下面是命令包的源代码，其他包的实现大体相同。

```
01 class CommandPacket : MySQLPacket() {
02     var command: Byte = 0
03     var arg: ByteArray? = null
04     override fun read(data: ByteArray) {
05         val mm = MySQLMessage(data)
06         packetLength = mm.readUB3()
07         packetId = mm.read()
08         command = mm.read()
09         arg = mm.readBytes()
10     }
11     @Throws(IOException::class)
12     fun write(out: OutputStream) {
13         StreamUtil.writeUB3(out, calcPacketSize())
14         StreamUtil.write(out, packetId)
15         StreamUtil.write(out, command)
16         out.write(arg)
17     }
18 }
```

表 5-6 mysql所有协议包的封装对象

文件	解释
AuthPacket	From client to server during initial handshake.
BinaryPacket	class BinaryPacket : MySQLPacket
CommandPacket	From client to server , the client wants the server to do something
EOFPacket	the EOF packet contains a warning
EmptyPacket	class EmptyPacket : MySQLPacket
ErrorPacket	From server to client in response to command, if error.
ExecutePacket	class ExecutePacket : MySQLPacket
FieldPacket	part of Result Set Packets. One for each column in the result set.
HandshakePacket	From server to client during initial handshake
HandshakeV10Packet	Connection Phase The Connection Phase performs these tasks
HeartbeatPacket	From client to server when the client do heartbeat between jsql cluster
LongDataPacket	class LongDataPacket : MySQLPacket
MySQLPacket	abstract class MySQLPacket
OkPacket	From server to client ,if no error and no result set
PingPacket	class PingPacket : MySQLPacket
PreparedOkPacket	class PreparedOkPacket : MySQLPacket
QuitPacket	class QuitPacket : MySQLPacket
Reply323Packet	class Reply323Packet : MySQLPacket
RequestFilePacket	load data local infile
ResetPacket	class ResetPacket : MySQLPacket
ResultSetHeaderPacke	The Result Set Header Packet is the first of several packets
RowDataPacket	From server to client. One packet for each row in the result set

协议主要有2个阶段，一个是认证阶段，一个是命令阶段，认证阶段就是接受客服端的请求，然后检查用户的认证信息，比如用户名和密码，下面代码主要用来实现用户认证的功能。

```
/**
 * 前端认证处理器
 * 处理auth包
 */
@Component
open class MysqlAuthHander : MysqlPacketHander {
    private fun handle0(auth: AuthPacket, source: OConnection) {
        source.schema = auth.database
        // check password
        if (!checkPassword(auth.password!!, auth.user!!)) {
            if (config.audit) {
            LoginLog(auth.user ?: "null", source.host, false).sendesServer()
            }
            failure(ErrorCode.ER_ACCESS_DENIED_ERROR,
      "Access denied for user '" + auth.user + "',
      because password is error ", source)
        } else {
            success(auth, source)
        }
    }
}
```

认证成功以后就是命令阶段，服务器接受客服端的命令，然后处理，下面是命令处理的具体代码。

```
/**
 * 前端命令处理器
 * 处理命令包
 */
@Component
```

```
06 class MysqlCommandHandler : MysqlPacketHander {
07     private fun handle0(data: CommandPacket, source: OConnection) {
08         logger.debug(data.toString())
09         logger.info("command info")
10         when (data.command) {
11             MySQLPacket.COM_INIT_DB -> initDB(data, source)
12             MySQLPacket.COM_QUERY -> query(data, source)
13             MySQLPacket.COM_PING -> ping(source)
14             MySQLPacket.COM_QUIT -> close("quit cmd", source)
15             MySQLPacket.COM_PROCESS_KILL -> kill(data, source)
16             MySQLPacket.COM_STMT_PREPARE -> stmtPrepare(data, source)
17             MySQLPacket.COM_STMT_SEND_LONG_DATA->stmtSendLongData(data,source)
18             MySQLPacket.COM_STMT_RESET -> stmtReset(data, source)
19             MySQLPacket.COM_STMT_EXECUTE -> stmtExecute(data, source)
20             MySQLPacket.COM_STMT_CLOSE -> stmtClose(data, source)
21             MySQLPacket.COM_HEARTBEAT -> heartbeat(data, source)
22             else -> source.writeErrMessage(ErrorCode.ER_UNKNOWN_COM_ERROR,
23                     "Unknown command")
24         }
25     }
26     private fun query(data: CommandPacket, source: OConnection) {
27         val mm = MySQLMessage(data.arg!!)
28         mm.position(0)
29         try {
30             val sql = mm.readString(source.charset)
31             source.sqlHander.handle(sql!!, source)
32         } catch (e: UnsupportedEncodingException) {
33             source.writeErrMessage(ErrorCode.ER_UNKNOWN_CHARACTER_SET,
"Unknown charset '" +
34       source.charset + "'")
35             e.printStackTrace()
36         }
37     }
```

```
38 }
```

5.2.2 SQL解析模块

表 5-7 SQL前端连接模块相关的类

类	作用
MysqlSQLhander	接受前端的语句，处理用户的请求
OConnection	前端连接对象，一个客服端一个连接
ONullConnection	继承连接对象，但是不处理具体任务，用在分布式中
OconnectionPool	管理前端的连接，作为一个连接池对象

sql解析主要用到了druid开源的框架，解析sql以后就要做具体的数据处理，下面的代码主要用处理用户的sql语句。

```
01 /**
02  * 处理sql语句的入口
03  */
04 @Component
05 class MysqlSQLhander : SQLHander {
06     @Autowired
07     lateinit private var allHanders: AllHanders
08   //所有的sql处理器容器
09     @PostConstruct
10     override fun handle(sql: String, c: OConnection){
11   //处理正常的sql语句，前端连接
12         if (config.audit) {
13             SqlLog(sql, c.user ?: "null", c.host).sentoELServer()
14         }
15         logger.info(sql)
16         if (logger.isDebugEnabled) {
17             logger.debug(sql)
18         }
19         val sqlStatement: SQLStatement
20         try {
21             sqlStatement = sql.tosql()
22             val hander = allHanders.handerMap[sqlStatement.javaClass]
```

```
23              if (hander != null) {
24                  hander.handle(sqlStatement, c)
25              } else {
26                  sqlStatement.accept(MSQLvisitor(c))
27              }
28              if (isupdatesql(sql)) {
29                  if (config.distributed) {
30                      myHazelcast.exeSql(sql, if (c.schema == null) ""
31            else c.schema!!)
32                  } else {
33                      myHazelcast.exesqlLocal(sql, if (c.schema == null)
34            "" else c.schema!!)
35                  }
36              }
37              return
38          } catch (e: Exception) {
39      //如果不是合法的mysql语句，就报错
40          //druid支持的语句就用上面的方法语句处理，如果不支持，就会有
异常，
41        //就自己写代码解析sql语句，处理。
42           //下面是drop event语句的例子，这个例子druid不支持，所以自
己写
43              handleotherStatement(sql, c, e)
44          }
45      }
46      fun handle(sql: SqlUpdateLog, c: OConnection) {
47    //处理来自其他服务器的sql语句，同步，不需要前端连接
48          logger.info(sql.toString())
49          if (logger.isDebugEnabled) {
50              logger.debug(sql.toString())
51          }
52          val sqlStatement: SQLStatement
53          try {
```

```
54              val parser = MySqlStatementParser(sql.sql)
55              sqlStatement = parser.parseStatement()
56              val hander = allHanders.handerMap[sqlStatement.javaClass]
57              if (hander != null) {
58                  hander.handle(sqlStatement, c)
59              } else {
60                  sqlStatement.accept(MSQLvisitor(c))
61              }
62              return
63          }
64      }
65  }
```

```
01  /**
02   * 所有的sql语句处理器必须是这个类的子类.
03   * 比如Mupdate。Mselect
04   */
05  abstract class SqlStatementHander {
06
07        //返回值只有4种可能，不然报错!!!
08        //一种是long类型， 一种是MyResultSet 一种是null ,
09      // 一种是string表示错误的消息
10        //返回其他对面都是错误的
11      fun handle(sqlStatement: SQLStatement, connection: OConnection)
{
12          try {
13              val result = handle0(sqlStatement, connection)
14              when (result) {
15                  null -> connection.writeok()
16                  is MyResultSet -> onsuccess(result.data,
17          result.columns, connection)
18                  is Long -> onsuccess(result, connection)
19                  is String -> connection.writeErrMessage(result)
```

```
20                  else -> connection.writeok()
21              }
22          } catch (e: Exception) {
23              e.printStackTrace()
24              onerror(e, connection)
25          }
26
27      }
28  }
```

mysql当中的sql语句有很多种，每一种语句都要做不同的处理，5-8描述了sql模块下实现的各种类型的语句。

表 5-8 SQL模块下实现的各种类型的语句

类	作用
io.jsql.sql.handler.adminstatement	处理管理语句
io.jsql.sql.handler.componed_statement	处理组合语句
io.jsql.sql.handler.data_define	处理数据定义语句
io.jsql.sql.handler.data_mannipulation	处理数据操作数据
io.jsql.sql.handler.preparestatement	处理准备语句
io.jsql.sql.handler.replication_statement	处理复制语句
io.jsql.sql.handler.tx_and_lock	处理事务和锁控制语句
io.jsql.sql.handler.utilstatement	处理其他工具语句

每一种类型语句下面又回有很多种具体语句的实现，表5-9给出了数据操纵类型语句实现的所有相关的类和响应的功能。

表 5-9 数据操纵数据实现所相关的类

文件	作用
MSelectHandler	处理查找语句
Mcall	处理函数调用语句
Mdelete	处理删除语句
Mdo	处理mysql中的do语句
Mhandler	所有类的基类
Minsert	处理插入语句
MloaddataINfile	处理导入文件的语句
Mloadxml	处理导入xml的语句
Mrepelace	处理解释语句
MselectVariables	处理查找变量的语句
Msubquery	处理子查询
Mupdate	处理更新语句

5.2.3 存储引擎模块

存储引擎有关的功能主要在storage包下面实现，表5-10给出了 storage包下面各种文件的功能。 其中DB类作为底层存储引擎的接口，他的接口如表5-11所示。

表 5-10 storage包下面各种文件的作用

文件	功能
DB	接口，规定所有的存储引擎应该实现的和数据库有关的功能
Table	接口，规定所有的存储引擎应该实现的和表有关的功能

和表有关的接口如表5-12所示。

表 5-11 数据库存储引擎的函数接口

接口	函数签名
close	abstract fun close(): Unit
createdbAsyn	abstract fun createdbAsyn(dbname: String): Unit
createdbSyn	abstract fun createdbSyn(dbname: String): Unit
deletedbAyn	abstract fun deletedbAyn(dbname: String): Unit
deletedbSyn	abstract fun deletedbSyn(dbname: String): Unit
exe	abstract fun exe(sql: String, db: String): Unit
exesqlNoResultAsyn	abstract fun exesqlNoResultAsyn(sql: String, dbname: String): Unit
exesqlforResult	abstract fun exesqlforResult(sql: String, dbname: String):Oresult
getallDBs	abstract fun getallDBs(): List<String>
getdb	abstract fun getdb(dbname: String): ODatabaseDocument
query	abstract fun query(sqlquery: String, dbname: String): Stream<OElement>

表 5-12 数据库引擎和表有关功能的接口

函数	函数签名
createtableSyn	abstract fun createtableSyn(dbname: String, createTableStatement
droptableSyn	abstract fun droptableSyn(dbname: String, table: String): Unit
getalltable	abstract fun getalltable(dbname: String): List<String>
gettableclass	abstract fun gettableclass(tablename: String, db: String): OClass
selectSyn	abstract fun selectSyn(oClass: OClass, dbname: String): Stream<OElement>

前面说过，本系统实现的存储引擎用的是哈希树这种数据结构。哈希树的实现主要有2个类，一个类实现哈希树的节点，一个类实现哈希树给外部模块提供调用接口。下面是其中实现的关键代码。

```
01 public class HtreeNode {
02     public int high;//root is 0
03     public int hashtable_size;
04     public HtreeNode[] childs;
```

```
05     public boolean hasV;
06     public String key;
07     public Object values;
08     public HtreeNode(int high, String key, Object values) {
09         this.high = high;
10         hashtable_size = HashCodes.codes[high];
11         this.key = key;
12         this.values = values;
13         childs = new HtreeNode[hashtable_size];
14         hasV = key != null;
15     }
16 }
```

```
01 public class HTreeMap implements Map<String, Object> ,Serializable{
02     @Override
03     public Object get(Object key) {
04         if (key == null) {
05             throw new NullPointerException("key not null");
06         }
07         int hashcode = Math.abs(key.hashCode());
08         HtreeNode htreeNode = root;
09         while (htreeNode != null) {
10             LOGGER.debug("this node is:" + htreeNode.key);
11             if (htreeNode.hasV && key.equals(htreeNode.key)) {
12                 LOGGER.debug("find " + key);
13                 return htreeNode.values;
14             }
15             int hashindex = hashcode % htreeNode.hashtable_size;
16             htreeNode = htreeNode.childs[hashindex];
17         }
18         return null;
19     }
20     @Override
```

```
21      public Object put(String key, Object value) {
22          if (key == null || value == null) {
23              throw new NullPointerException("key not null");
24          }
25          int hashcode = Math.abs(key.hashCode());
26          HtreeNode htreeNode = root;
27          HtreeNode pre = null;
28          while (htreeNode != null) {
29              if (htreeNode.hasV && key.equals(htreeNode.key)) {
30                  Object stemp = htreeNode.values;
31                  htreeNode.values = value;
32                  return stemp;
33              } else if (!htreeNode.hasV) {
34                  htreeNode.hasV = true;
35                  htreeNode.values = value;
36                  htreeNode.key = key;
37                  return null;
38              }
39              pre = htreeNode;
40              int hashindex = hashcode % htreeNode.hashtable_size;
41              htreeNode = htreeNode.childs[hashindex];
42          }
43          int hashindex = hashcode % pre.hashtable_size;
44          pre.childs[hashindex] = new HtreeNode(pre.high + 1, key, value);
45          allnodes.add(pre.childs[hashindex]);
46              return null;
47      }
48      @Override
49      public Object remove(Object key) {
50          if (key == null) {
51              throw new NullPointerException("key not null");
52          }
53          int hashcode = Math.abs(key.hashCode());
```

```
54          HtreeNode htreeNode = root;
55          while (htreeNode != null) {
56              if (htreeNode.hasV && key.equals(htreeNode.key)) {
57                  htreeNode.hasV = false;
58                  return htreeNode.values;
59              }
60              int hashindex = hashcode % htreeNode.hashtable_size;
61              htreeNode = htreeNode.childs[hashindex];
62          }
63          return null;
64      }
65  }
```

除了实现数据结构以外还要实现存储引擎，也就是把数据存储在文件系统当中，其中主要用到内存映射文件，下面是其中的关键代码。

```
01 /**
02  * The type M storage.
03  * 文件前面2m保留做头信息1024*1024*2/4096=512页面
04  */
05 class MStorage {
06     public void write(long pageNumber, ByteBuffer data) throws IOException
{
07         FileChannel f = getChannel(pageNumber);
08         int offsetInFile = (int) ((Math.abs(pageNumber) %
09     Pagesize.max_page_number) * Pagesize.page_size);
10         MappedByteBuffer b = buffers.get(f);
11         if (b.limit() <= offsetInFile) {
12             b = addfilesize(f, offsetInFile, b);
13         }
14         //write into buffer
15         b.position(offsetInFile);
16         data.rewind();
17         b.put(data);
```

```
18      }
19      public ByteBuffer read(long pageNumber) throws IOException {
20          FileChannel f = getChannel(pageNumber);
21          int offsetInFile = (int) ((Math.abs(pageNumber) %
22      Pagesize.max_page_number) * Pagesize.page_size);
23          MappedByteBuffer b = buffers.get(f);
24          if (b == null) { //not mapped yet
25              b = f.map(FileChannel.MapMode.READ_WRITE, 0, f.size());
26          }
27          //增加文件大小，64m为单位
28          if (b.limit() <= offsetInFile) {
29              b = addfilesize(f, offsetInFile, b);
30          }
31          b.position(offsetInFile);
32          ByteBuffer ret = b.slice();
33          ret.limit(Pagesize.page_size);
34          if (!transactionsDisabled || readonly) {
35              // changes written into buffer will be directly written
into file
36              // so we need to protect buffer from modifications
37              ret = ret.asReadOnlyBuffer();
38          }
39          return ret;
40      }
41  }
```

```
01  /**
02   * 和内存指针差不多，new 后得到地址,这里地址是page index
03   * 每个页面开始分别是type，记录大小，数据，页面后2个字节用来连接每
个页面
04   * 一个页面pagesize-2-4-4
05   */
06  public class DiscIO implements MdiscIO {
```

```
07      @Override
08      public int write(Object o) {
09          byte[] bytes = ObjectSeriaer.getbytes(o);
10          int[] pages = pagemanager.getfreepanages(bytes.length);
11          if (pages.length == 1) {
12              try {
13                  ByteBuffer buffer = storage.read(pages[0]);
14                  if (o instanceof DHtree)
15                      buffer.putShort(Pagesize.pagehead_tree);
16                  else if (o instanceof DHtreeNode) {
17                      buffer.putShort(Pagesize.pagehead_node);
18                  } else {
19                      buffer.putShort(Pagesize.pagehead_other);
20                  }
21                  buffer.putInt(bytes.length);
22                  buffer.put(bytes);
23                  storage.write(pages[0], buffer);
24                  ObjectMap.putorupdate(o, pages[0]);
25                  return pages[0];
26              } catch (IOException e) {
27                  e.printStackTrace();
28              }
29          } else {
30              return writemorepage(bytes, pages, o);
31          }
32          return 0;
33      }
34      @Override
35      public int update(Object o, int recid) {
36          if (!ObjectMap.map.containsKey(recid)) {
37              return -1;
38          }
39          byte[] bytes = ObjectSeriaer.getbytes(o);
```

```
40          if (bytes.length <= Pagesize.page_size_for_content) {
41              try {
42                  ByteBuffer buffer = storage.read(recid);
43                  if (o instanceof DHtree)
44                      buffer.putShort(Pagesize.pagehead_tree);
45                  else if (o instanceof DHtreeNode) {
46                      buffer.putShort(Pagesize.pagehead_node);
47                  } else {
48                      buffer.putShort(Pagesize.pagehead_other);
49                  }
50                  buffer.putInt(bytes.length);
51                  buffer.put(bytes);
52                  storage.write(recid, buffer);
53                  ObjectMap.putorupdate(o, recid);
54                  return recid;
55              } catch (IOException e) {
56                  e.printStackTrace();
57                  return -1;
58              }
59          } else {
60              updatemoredata(bytes, recid, o);
61              return recid;
62          }
63      }
64  }
```

5.3 集群架构的实现

集群功能主要是利用了开源的hazelcast框架来实现，其中的功能全在hazelcast包下面实现，表5-13给出了该包下面每个类的具体的作用。 其中最关键的代码如下。

```
01 //sql队列 复制队列命令队列。2个锁。发布
02 @Component
```

表 5-13 集群模块下面各个类的作用

类	作用
LogFile	本地文件系统中存储日志文件
MyHazelcast	利用Hazelcast的分布式数据结构实现集群，比如分布式队列和分布式锁
ReplicationCMD	分布式对象，在不同集群之间传输
SqlUpdateLog	当前系统LSN最大值，新的事务日志LSN将在此基础上生成

```
03 class MyHazelcast : ItemListener<SqlUpdateLog> {
04     private fun exeSqlforReplication() {
05         Collections.sort(localqueneReplication)
06         var log: SqlUpdateLog? = localqueneReplication.poll()
07         var lastlsn: Long = 0
08         while (log != null) {
09             locals_maxlsn = log.LSN
10             logger.info("exeSqlforReplication exe sql " + log)
11             oNullConnection!!.schema = log.db
12             sqLhander.handle(log, oNullConnection)
13             logFile!!.write(log)
14             lastlsn = log.LSN
15             log = localqueneReplication.poll()
16         }
17         isreplicating = false
18         log = localquene.poll()
19         val remotel = iAtomic_remote_lsn!!.get()
20         while (log != null) {
21             locals_maxlsn = log.LSN
22             logger.info("exe  Sql : " + log)
23             oNullConnection!!.schema = log.db
24             sqLhander.handle(log, oNullConnection)
25             logFile!!.write(log)
26             lastlsn = log.LSN
27             if (lastlsn > remotel) {
28                 remotequene!!.offer(log)
```

```
29              }
30              log = localqueneReplication.poll()
31          }
32          if (lastlsn > remotel) {
33              iAtomic_remote_lsn!!.set(lastlsn)
34          }
35      }
36      /**
37       *  本机发出的sql语句.记录到本地logfile。
38      同时发布到其他服务器*/
39      fun exeSql(sql: String, db: String) {
40          if (isreplicating) {
41              val l = iAtomic_remote_lsn!!.get()
42              val log = SqlUpdateLog(l + 1, sql, db)
43              localquene.addLast(log)
44          } else {
45              locals_maxlsn++
46              val l = iAtomic_remote_lsn!!.addAndGet(1)
47              val log = SqlUpdateLog(l, sql, db)
48              logger.info("exe sql " + log)
49              logFile!!.write(log)
50              remotequene!!.offer(log)
51          }
52      }
53      private fun exesqlforremoteData() {
54          Collections.sort(localquene)
55          var log: SqlUpdateLog? = localquene.poll()
56          var last: Long = 0
57          while (log != null) {
58              locals_maxlsn = log.LSN
59              logger.info("exesqlforremoteData()  " + log)
60              oNullConnection!!.schema = log.db
61              sqLhander.handle(log, oNullConnection)
```

```
62              logFile!!.write(log)
63              last = log.LSN
64              log = localquene.poll()
65          }
66          val l = iAtomic_remote_lsn!!.get()
67          if (last > l) {
68              iAtomic_remote_lsn!!.set(last)
69          }
70
71      }
72
73  }
```

5.4 数据审计模块的实现

审计模块有关的功能全部在audit包下面实现，表5-14给出了audit包下面每个类的具体的作用。

表 5-14 audit包下面每个类的作用

类	作用
LoginLog	简单对象，记录登陆日志
SqlLog	简单对象，记录SQL执行日志
elasticUtil	工具类，包日志记录发送到ELK

5.4.1 审计数据库

审计数据库用Elasticsearch来实现，作为审计数据库，不能让用户随意的更改，所以本系统更改了它的源代码，使得它只能增加数据和查找数据不能更改和删除数据。其中主要存储的对象如图5-14所示。

5.4.2 审计管理器

审计管理功能全部在audit下面实现，主要是存储本地的日志文件，然后发布到审计数据库。提供给前端可视化的接口。

5.4.3 审计可视化模块的实现

审计可视化模块的实现用到了grafana，主要用来监控ELK中的数据。

5.5 本章小结

本文前面一章设计了本系统的架构图和各个模块的详细功能，本章给出每个功能模块的具体实现。。

第六章 系统测试

6.1 测试环境

本文的测试环境为个人的计算机。它的硬件和软件参与如下

1.硬件参数：

(a) CPU：intel(R) Xeon(R) E5620

(b) 内存：8G

(c) 磁盘：500G

(d) 网络：100Mbit/s

2.软件参数：

(a) 操作系统：windows10

(b) java版本：8.0

6.2 功能测试

6.2.1 数据库功能测试

功能测试设计的细节非常多，项目繁杂。这里将给出几个关键的功能测试的条件、目的、步骤和结果。其他更为细节的，比如对失败的查询的测试，或其他类似的，比如与插入、更新类似的删除数据，删除表，由于篇幅原因，就不再细述。

1.系统启动测试

(a) 测试条件：代码编码完成，系统功能正常

(b) 测试步骤：

i. 启动系统

ii. 用mysql客服端连接系统

(c) 测试结果：系统能正常启动，而且mysql客服端也可以连接上系统

2.工具语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句show databases，发送给服务器

(c) 测试结果：系统返回当前所有的数据库。

3.建表语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句create table test(id int,name varchar(100))

iii. 回头，发送命令给服务器

(c) 测试结果：系统返回正确，而且文件系统里面多了一个文件。

4.插入语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句insert into test(id,name) values(1,'changhong');

iii. 回头，发送命令给服务器

(c) 测试结果：系统返回正确

5.查询语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句select * from test;

iii. 回头，发送命令给服务器

(c) 测试结果：系统返回一行数据。

6.更新语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端

ii. 输入语句update test set name='changhong1'

iii. 回头，发送命令给服务器

iv. 输入语句select * from test

v. 回头，发送命令给服务器

(c) 测试结果：系统执行正确，返回一条数据，名字字段weichangohng1

7.更新语句测试

(a) 测试条件：系统启动成功，客服端连接上系统

(b) 测试步骤：

i. 打开命令行客服端
ii. 输入语句delete from test
iii. 回头，发送命令给服务器
iv. 输入语句select * from test
v. 回头，发送命令给服务器

(c) 测试结果：系统执行正确，没有返回数据

6.2.2 集群功能测试

集群的功能测试和单机版一样，考虑到不可能每条语句的测试。本次测试只测试最常用的insert语句：

1.测试条件：

(a) 数据库1启动成功，没有任何数据
(b) 数据库2启动成功，没有任何数据
(c) 数据库1和数据库2都有一个test表

2.测试步骤：

(a) 打开命令行客服端，连接数据库1
(b) 输入语句insert into test(id,name) values(1,'changhong')并且执行;
(c) 断开连接，连接数据库2
(d) 输入语句select * from test;
(e) 回头，发送命令给服务器

3.测试结果

(a) 数据库1执行语句正确
(b) 语句2执行正确，并且返回一条数据

6.2.3 审计功能测试

审计功能的测试主要是测试可视化的显示功能。

1.测试条件：

(a) 数据库1启动成功，没有任何数据
(b) 数据库1有一个test表

2.测试步骤：

(a) 打开命令行客服端，连接数据库1
(b) 输入语句insert into test(id,name) values(1,'changhong')并且执行;
(c) 打开grafana web界面

3.测试结果

(a) 数据库1执行语句正确

(b) 通过web界面可以观察到插入的数据和执行者的信息

(c) 图6-1,图 6-2和图 6-3显示了监控的可视化结果

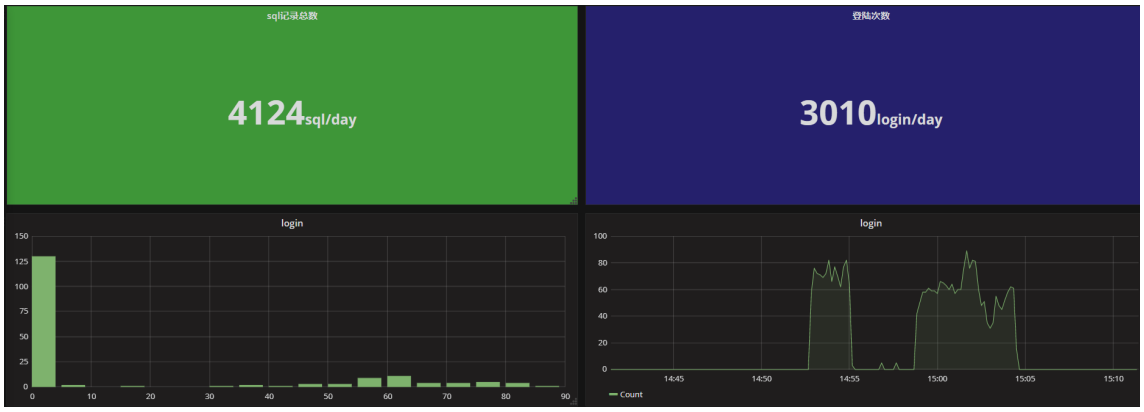


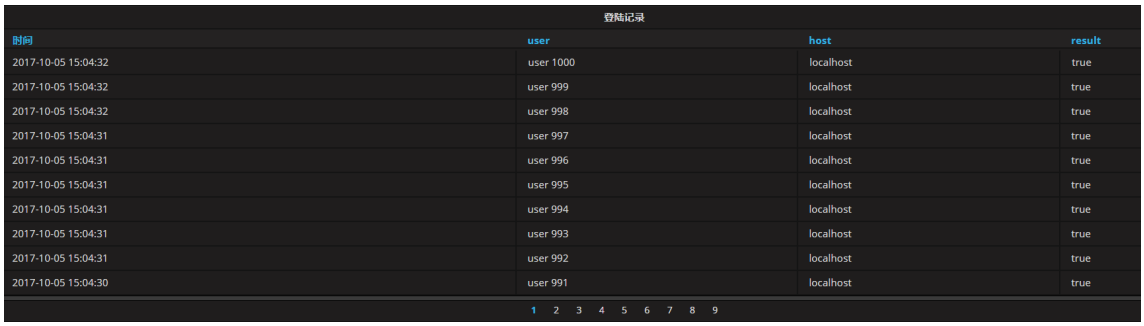
图 6-1 监控可视化结果

图 6-2 监控可视化结果

6.3 性能测试

数据库的更删改查是最常用的操作，其中又以查找使用的场景更多，所以本节主要测试数据库的查询性能。jsql和mysql都可以通过JDBC来连接，下面的源代码是测试代码，用来测试查询次数和执行时间的关系。执行结果如图6-4所示。

```
01    fun test() {
02        //sql执行次数
03        val sqlnumber = intArrayOf(10, 100, 1000, 3000, 5000, 10000)
```



登陆记录

时间	user	host	result
2017-10-05 15:04:32	user 1000	localhost	true
2017-10-05 15:04:32	user 999	localhost	true
2017-10-05 15:04:32	user 998	localhost	true
2017-10-05 15:04:31	user 997	localhost	true
2017-10-05 15:04:31	user 996	localhost	true
2017-10-05 15:04:31	user 995	localhost	true
2017-10-05 15:04:31	user 994	localhost	true
2017-10-05 15:04:31	user 993	localhost	true
2017-10-05 15:04:31	user 992	localhost	true
2017-10-05 15:04:30	user 991	localhost	true

1 2 3 4 5 6 7 8 9

图 6-3 监控可视化结果

```
04          val mysql = "jdbc:mysql://localhost:3306/changhong?" + "user=root&"
+
05                  "password=0000&useUnicode=" + "true&characterEncoding=UTF8"
06          var connection = DriverManager.getConnection(mysql)
07          var statement = connection.createStatement()
08          println("mysql sqlnumber to times:")
09          for (number in sqlnumber) {
10              val start = System.currentTimeMillis()
11              for (i in 1..number) {
12                  statement.executeQuery("select * from test")
13              }
14              val end = System.currentTimeMillis()
15              println("$number      ${end - start}")
16          }
17          connection.close()
18          val jsql = "jdbc:mysql://localhost:9999/changhong?" + "user=root&"
+
19                  "password=0000&useUnicode=" + "true&characterEncoding=UTF8"
20          connection = DriverManager.getConnection(jsql)
21          statement = connection.createStatement()
22          println("jsql sqlnumber to times:")
23          for (number in sqlnumber) {
24              val start = System.currentTimeMillis()
25              for (i in 1..number) {
26                  statement.executeQuery("select * from test")
```

```
27              }
28              val end = System.currentTimeMillis()
29              println("$number      ${end - start}")
30          }
31          connection.close()
32      }
33  }
```

SQL执行次数	MYSQL完成时间（毫秒）	JSQL完成时间（毫秒）
10	5	4
100	31	40
1000	252	315
3000	1207	941
5000	648	1504
10000	1297	4495

图 6-4 mysql和jsql的性能比较

从图6-4可以看出，随着SQL执行次数的增加，mysql和jsql的执行时一般都响应的增加，在3000执行次数为3000以下时，jsql和mysql的性能相当，当执行次数大于5000时，mysql的性能小幅度的超过jsql。再从变化幅度来看，当执行语句是5000次数的时候，mysql的执行时候审计比执行3000次数的时候还要少，这点可能是因为mysql本身的缓存或者优化有关。而jsql就相对来说随着执行次数的增加，时间就随着增加，符合预期。

除了jdbc的测试以外，本次测试还用了一个流行的测试框架JMeter。 Apache JMeter是Apache组织开发的基于Java的压力测试工具。用于对软件做压力测试，它最初被设计用于Web应用测试，但后来扩展到其他测试领域。它可以用于测试静态和动态资源，例如静态文件、Java 小服务程序、CGI 脚本、Java 对象、数据库、FTP 服务器，等等。JMeter 可以用于对服务器、网络或对象模拟巨大的负载，来自不同压力类别下测试它们的强度和分析整体性能另外，JMeter能够对应用程序做功能/回归测试，通过创建带有断言的脚本来验证你的程序返回了你期望的结果。为了最大限度的灵活性，JMeter允许使用正则表达式创建断言。用JMeter测试jsql的结果如图6-5所示。用JMeter测试jsql的结果如图6-6所示。

从测试结果来看，jsql相对mysql的性能有一定的差距。其中一个原因是jsql是用纯java语言开发的，而mysql用c语法开发，其中自然有一定的性能损耗。

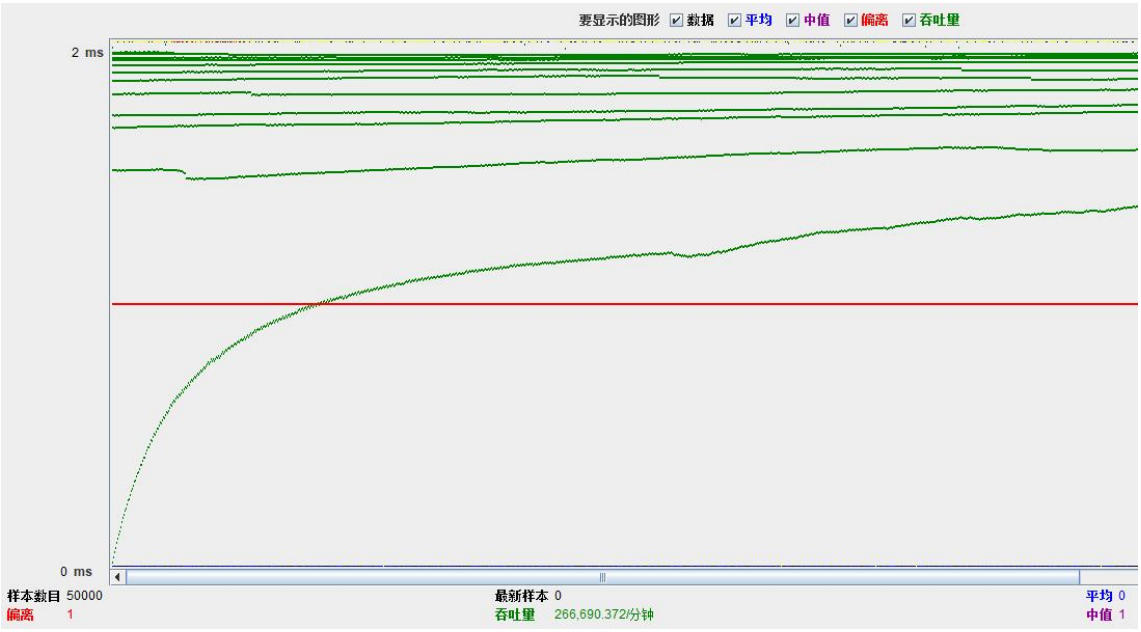


图 6-5 jsql的JMeter性能测试

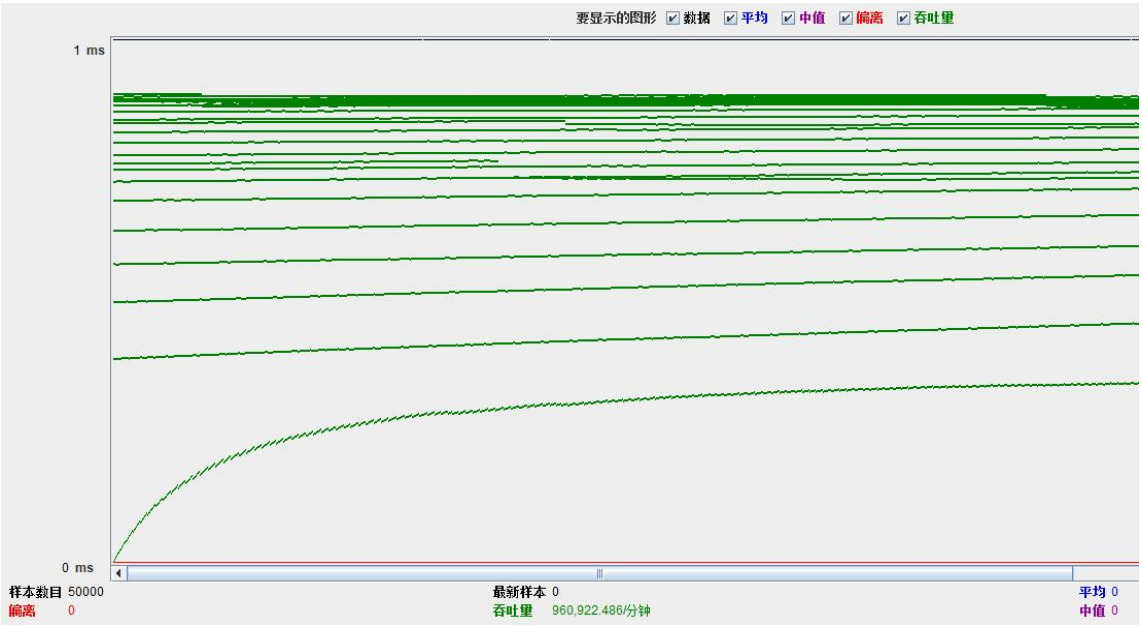


图 6-6 mysql的JMeter性能测试

6.4 本章小结

本章对JSQL数据库系统进行测试，主要分为功能测试和性能测试，功能测试包括数据库功能测试，集群功能测试以及审计功能的测试。性能测试分为JDBC和JMeter测试两部分。

第七章 总结与展望

现在社会的发展，离不开各种各样的数据库系统。关系型数据库在很多关键场合有不可替代的优势，对关系型数据库有关的理论研究和实践非常有意义。因此，在导师的指导下，在研究生阶段作者主要做数据库有关的开发工作。最终，将理论知识和实践技能相结合，开发出了这套分布式数据库系统。

分布式系统的研发是一个不小的挑战。作为这个系统的开发者，我首先在理论方面做了很多学习和研究，包括硬件硬件有关的知识，和数据库事务相关的理论，以及各种分布式相关的协议。理论只能作为指导，而不能成为一个系统。在实现方面，我主要深入学习了JAVA语言，对多线程和高性能网络开发技能也进行了深入的学习。最终实现的这套数据库系统作者认为有如下优先：

1.从数据库引擎到SQL模块完全采用JAVA语言编写，具有跨平台和安全的特点。

2.利用哈希树实现了高性能的存储引擎，特别适合现在的应用程序。

3.在系统的架构设计上，采用了面向对象的思想，对各个模块进行了很好的。划分，有利于对系统进行进一步的完善。

4.从数据库系统本身加入了审计功能。

当然，由于作者水平有限，本系统难免还有很多需要完善的地方，总结起来有下面几个方面：

1.系统的数据恢复机制不够完善，后续需要对存储引擎进行进一步的开发，加入日志等功能。

2.分布式数据库中数据迁移目前还没有实现，这是值得进一步研究的课题。

3.关系型数据库大都有存储过程和触发器这些功能，本系统作为一个实验室产品，目前还没有完全的实现这些功能。

作者认为，虽然现在出现了很多的Nosql数据库系统，但是关系型数据库的作用是无法替代的，电子商务和各种银行业务都需要关系型数据库的支持，所以对关系型数据库的学习和实践特别有意义。另外，数据库的安全问题越发严重，时常发生管理员篡改数据的情况发生，所以作者觉得，每个数据库系统，都要从底层加入安全审计功能，这样我们的数据才能保证最基本的可靠性。关于本系统，需要做的事情还有很多，但是我相信随着进一步的开发，本系统的功能会进一步完善。

致 谢

时间过的真快，转眼间，2年多的研究生生活就快要结束了。回顾自己的研究生学习生涯，感慨万千。论文的完成，除了自己的努力以外，更离不开老师和学弟们的帮助，在论文成稿之际，衷心感谢给予自己悉心指导和热情帮助的各位老师和学弟们。

论文的完成，首先要感谢我的校内导师曹晟教授，在整个论文的的写作过程中，曹老师都给予了我很大的关心和帮助。从作者毕业论文的选题、写作一直到最终完成的过程中，曹老师都是在百忙的工作中以一贯认真负责的态度认真仔细阅读作者的论文，给予作者耐心的指导，使得论文能够顺利的完成。他严肃的科学态度，严谨的治学精神，以及精益求精的工作作风，深深地感染和激励着我。

在这一年多的时间里，我还要感谢我的学弟们。感谢你们陪我一起开发这个分布式数据库系统，我们一起学习，一个努力，才能让本系统顺利完成。任何一个系统都要靠团队合作，更何况分布式系统这个既具有挑战性的工程项目，没有你们的帮助，就不可能按时完成这个系统。对学弟们的帮助，在此表示非常的感谢。

最后，作者非常感谢负责评审论文的教师、专家和教授，感谢你们认真负责的阅读论文，感谢你们为论文提出的宝贵意见和建议。