分类号 TP311.13 密级 公开

UDC 004.9 编号 10299Z1408018

****

硕 士 学 位 论 文

**基于HBase的交通流数据实时存储与查询优化方法的**

**研究与实现**

**Design and Implementation of HBase-based Traffic Stream**

**Data Real-time Storage And Query System**

指导教师 李 星 毅

作者姓名 瞿 龙 俊

申请学位级别 硕士 专业名称 计算机技术

论文提交日期 2017年04月 论文答辩日期 2017年06月

学位授予单位和日期 江苏大学 2017 年 06 月

答辩委员会主席 \_\_\_\_\_\_\_\_\_\_\_\_\_\_

评阅人 \_\_\_\_\_\_\_\_\_\_\_\_\_\_

学位论文版权使用授权书

江苏大学、中国科学技术信息研究所、国家图书馆、中国学术期刊（光盘版）电子杂志社有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致，允许论文被查阅和借阅，同时授权中国科学技术信息研究所将本论文编入《中国学位论文全文数据库》并向社会提供查询，授权中国学术期刊（光盘版）电子杂志社将本论文编入《中国优秀博硕士学位论文全文数据库》并向社会提供查询。论文的公布（包括刊登）授权江苏大学研究生处办理。

本学位论文属于不保密 □ 。

学位论文作者签名： 指导教师签名：

年 月 日 年 月 日

独 创 性 声 明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容以外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果，也不包含为获得江苏大学或其他教育机构的学位或证书而使用过的材料。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

摘 要

随着经济的飞速发展，智能交通技术得到了快速发展，以数据为基础的交通控制等技术得到业界的重视，交通数据采集技术迅速发展，采集手段多样化，数据量激增，我国沿海地区一个中型城市每天产生数千万条交易数据，每年汇集的数据量达到百TB级。面对如此庞大的数据量，必须对其存储和查询的系统进行优化，以适应实际应用需求。

Hadoop是成熟的大数据处理技术体系，其HDFS和MapReduce技术提供了高效的数据存储能力和数据分析能力。HBase分布式数据库底层采用Hadoop的分布式文件系统实现存储，支持Hadoop并行计算框架，使用HBase作为海量数据的存储媒介，具有更高的可靠性和数据处理能力。为解决传统关系型数据库处理海量数据存储和查询的性能效率低的问题，本文主要研究基于HBase的交通流数据实时存储与查询优化方法，并研发了DEMO系统，实现交通工程中对数据的实时处理要求，主要研究工作和创新点如下：

1. 借鉴前人的研究成果，对HBase的数据模型和集群架构进行优化，设计出了一种符合交通流数据应用特征的存储模型。
2. 为了防止大数据量并发写入导致的写阻塞等性能问题，引入了数据预处理层、数据缓冲层和数据写入层，构建了一个基于Redis集群服务器的缓存层。
3. 设计缓存客户端的数据查询语义解析方法，实现了基于本地缓存模块、Redis缓存服务器、HBase集群服务器的协同的查询解析和调度模块，提高了查询效率。
4. 构建了基于HBASE的交通数据实时存储和查询DEMO系统，通过实验和效率比对验证了研究成果的可用性和高效性。

**关键词： HBase，交通数据流，Redis集群，SQL解析，TwemProxy**

ABSTRACT

Intelligent Transportation System is a typical interdisciplinary application system, which combines signal processing technology, information processing technology and monitoring technology to provide more efficient traffic control for the management department. It can predict and alleviate the traffic congestion in time, providing a favorable basis for city planning. Intelligent transportation system data sources are very extensive, intelligent traffic monitoring equipment, roadside GPS equipment, and even the vehicle information of parking lot belong to the category of intelligent traffic data.

With the rapid development of economy, the automobile has become an indispensable means of transportation. Along with the continuous improvement of transportation, the channels for acquiring intelligent traffic data are becoming more and more diversified. Taking one large city as an example, according to government data released on the website，the city's motor vehicle population in 2015 exceeded 4.3 million, an average of one hundred families have 52 private cars. As of 2016, the city has a total of 15 473 roads, vehicle identification sensor deployed in each crossing being over more than 5000, producing tens of millions of data per day, hundreds of terabytes of data annually. With the development of urban construction, digital scale will continue to expand. Faced with such a huge amount of data, developers must optimize their storage and query system, otherwise, in such a large amount of data in a single query time-consuming will be inestimable. At present, most of the intelligent transportation system are using the traditional relational database as a storage medium, such as MySQL, Oracle, etc. These traditional relational databases have problems such as high write latency and difficulty in horizontal extension.

As a more mature large data processing technology, Hadoop has been gradually favored by the major companies and used to deal with large data-related business scenarios. Hadoop's two most prominent features, HDFS and MapReduce, provide efficient data storage and data analysis capabilities. The HBase is located in the Hadoop ecosystem and supports for MapReduce parallel computing distributed database, the bottom of which using HDFS implementation. Therefore, the use of HBase as a traffic flow data real-time storage and query system storage media, than the traditional relational database with higher reliability and processing power.

In order to solve the performance flaw of traffic flow data on traditional relational database, this paper designs and realizes a real-time storage and query system of traffic flow data based on HBase, studying data writing and data reading on the basis .The main research work is as follows:

1. Firstly, the paper deeply understands and analyzes the related technologies and characteristics of HBase, and focus on understanding the data model and cluster architecture of HBase. On this basis, we design a traffic data storage model.
2. Secondly, on the basis of traffic data storage model, data preprocessing layer, data buffer layer and data writing layer are designed and introduced in order to prevent performance problems such as write blocking caused by large data volume.
3. In the aspect of data query, the paper introduces the SQL parsing function and implements a high performance cache server based on Redis cluster. By cooperating with several modules and optimizing the Redis cache data model, the efficiency of data query is improved.
4. Finally, on the basis of theoretical design, the experimental design is carried out. Through the correlation analysis and data comparison, it provides strong support for the theory, and proves the necessity of the real-time storage and query system of traffic flow data.

**Keywods：HBase, Traffic Data Flow, Redis cluster, SQL Parsing, TwemProxy**

目 录

[摘 要 III](#_Toc474308093)

[ABSTRACT V](#_Toc474308094)

[第一章 绪论 1](#_Toc474308095)

[1.1 课题研究背景与意义 1](#_Toc474308096)

[1.2 主要研究内容 1](#_Toc474308097)

[1.3国内外研究现状 2](#_Toc474308098)

[1.4本文组织结构 2](#_Toc474308099)

[第二章 技术相关 4](#_Toc474308100)

[2.1 Hadoop技术 4](#_Toc474308101)

[2.1.1 HDFS 4](#_Toc474308102)

[2.1.2 MapReduce 6](#_Toc474308103)

[2.2 HBase技术 6](#_Toc474308104)

[2.2.1 HBase集群架构 7](#_Toc474308105)

[2.2.2 HBase数据模型 8](#_Toc474308106)

[2.3 Redis技术 10](#_Toc474308107)

[2.3.1 Redis概述 10](#_Toc474308108)

[2.3.2 Redis数据类型 10](#_Toc474308109)

[2.3.3 持久化 12](#_Toc474308110)

[2.3.4 主-从复制 13](#_Toc474308111)

[2.4 本章小结 14](#_Toc474308112)

[第三章 交通流数据实时存储与查询系统的设计 15](#_Toc474308113)

[3.1道路车辆信息数据模型 15](#_Toc474308114)

[3.2 系统整体架构 15](#_Toc474308115)

[3.3 HBase集群模型结构设计 18](#_Toc474308116)

[3.3.1 Column Family 18](#_Toc474308117)

[3.3.2 Row Key 18](#_Toc474308118)

[3.4 Redis缓存服务器设计 19](#_Toc474308119)

[3.4.1 需求分析 19](#_Toc474308120)

[3.4.2系统架构 21](#_Toc474308121)

[3.4.3 Redis 键值设计 22](#_Toc474308122)

[3.4.4 缓存淘汰算法 23](#_Toc474308123)

[3.5 本章小结 25](#_Toc474308124)

[第四章 交通流数据实时存储与查询系统的实现 26](#_Toc474308125)

[4.1 数据写入功能 26](#_Toc474308126)

[4.1.1 数据写入流程 26](#_Toc474308127)

[4.1.2 数据缓冲层的实现 27](#_Toc474308128)

[4.1.3 数据写入层的实现 30](#_Toc474308129)

[4.1.4 HBase集群扩展 32](#_Toc474308130)

[4.2 SQL解析 32](#_Toc474308131)

[4.2.1 Phoenix解析器介绍 32](#_Toc474308132)

[4.2.2 Phoenix解析器配置 35](#_Toc474308133)

[4.3 缓存集群服务器的实现 35](#_Toc474308134)

[4.3.1 KeepAlived配置 35](#_Toc474308135)

[4.3.2 Redis Cluster搭建 38](#_Toc474308136)

[4.3.3 TwemProxy配置 38](#_Toc474308137)

[4.4 HBase与缓存的交互 39](#_Toc474308138)

[4.4.1 本地缓存 39](#_Toc474308139)

[4.4.2 Cache Client 44](#_Toc474308140)

[4.5 集群优化配置 46](#_Toc474308141)

[4.6 本章小结 47](#_Toc474308142)

[第五章 实验设计与结果分析 48](#_Toc474308143)

[5.1 系统环境搭建 48](#_Toc474308144)

[5.1.1 硬件环境 48](#_Toc474308145)

[5.1.2 软件环境 49](#_Toc474308146)

[5.1.3 实验数据准备 49](#_Toc474308147)

[致 谢 51](#_Toc474308148)

[在校期间发表论文 52](#_Toc474308149)

第一章 绪论

1.1 课题研究背景与意义

随着互联网和信息采集技术的飞速发展，人们迎来了数据的大爆炸时代，越来越多的数据被产生并被挖掘出价值[[[1]](#endnote-1)]。其中，非结构化的数据，例如全文文本、音频信息、视频媒体、网络日志等，超过了数据总量的80%以上。国内外的一些著名公司，如Facebook、Twitter、Tabao等，用户规模量大，并发请求多，产生PB级别的非结构化数据[[[2]](#endnote-2)]，由此催生了一系列高性能、高可扩展性、廉价的分布式存储系统。

HBase[[[3]](#endnote-3)]作为Apache社区软件基金会的子项目，是Google于2006年提出BigTable[[[4]](#endnote-4)]数据模型的一种开源实现。HBase是一种以列为单位的、可自由扩展伸缩的键值对分布式存储系统，其底层数据存储实现方案采用的是Hadoop分布式文件系统HDFS。目前，许多互联网企业开始将HBase作为主流数据库使用。

分布式数据库的出现为现代城市智能交通系统[[[5]](#endnote-5)]提供了新的机遇。智能交通系统将道路监控拍摄的图片、视频等信息进行收集并传输到省市级的数据中心，并从图片、视频中读取相应的信息存储至数据库[[[6]](#endnote-6)]。在进行数据分析时，通过实时的查询请求获取对应数据信息。随着城市的发展，道路监控设备的完善，截止至2016年，道路监控几乎覆盖到城市所有路口和高速公路、国道、省道等主要出口。截止2016年6月底，全国机动车保有量达2.85亿量，随着汽车保有量逐年增加，中等地级市每年能产生数以十亿条的记录。随着采集设备的多样化，如全球定位系统（GPS）、路口路段监控设施、停车场车辆信息采集器等设备的覆盖，采集的数据类型也呈现出多元化特性，如交通流监测数据、车辆信息识别数据等。这些不断产生的数据已经演变成了一种极具价值的交通数据集，若配备以高效的存储与检索方案，对提升交通管理水平具有十分重大的意义。

1.2 主要研究内容

本文关注的重点对象为海量数据的实时存储与高并发的查询请求时的性能问题，传统关系型数据库，如MySQL[[[7]](#endnote-7)]、Oracle[[[8]](#endnote-8)]、DB2[[[9]](#endnote-9)]等，由于二维数据无法扩展到多个存储节点，因此，在存储大量非结构化、半结构化数据时扩展性较差。另外，高并发的应用请求也会导致关系型数据库的性能问题。以HBase为代表的非关系型数据库，具有高度可扩展性、支持非结构化和半结构化数据类型、支持随机存取访问、 存储效率高等特点[3]，适合作为交通流数据的存储媒介。但HBase在实际应用的过程中，也存在着一些缺陷。HBase使用键值对作为数据存储模型，只提供了存取功能，未提供SQL解析器、不支持二级索引查询，不支持聚集查询等。

HBase采用基于类B+树索引，可以高效的进行基于主键的单点和范围数据查询，而在处理未指定主键的查询时，HBase采用扫描（Scan）整张数据表策略，导致延迟大，响应时间较长。在传统的关系型数据库领域，为了减少响应时间，一般会借助索引缓存策略进行解决，如Dbcache[[[10]](#endnote-10)]、OracleAS[[[11]](#endnote-11)]等。本文针对HBase在处理未指定主键时查询效率较低的现象，在研究国内外现有缓存策略的基础上，提出一种基于本地缓存和Redis集群服务器的多级索引查询模型，同时，根据交通流数据多源、实时性高、数据量大的特点，设计了一种多层缓冲写入模型。本文主要工作内容包括：

1. 分析了在设计交通流数据的实时存储与查询系统中运用到的关键技术。首先分析了Hadoop的两个特性：HDFS与MapReduce，在此基础上，了解HBase的系统架构，数据模型，以及HBase的读写的缓存机制MemStore与Block Cache，随后研究了Redis集群相关技术。
2. 在深入了解相关技术后，对交通流数据实时存储与查询系统进行了理论设计。首先，根据数据的特性，设计了一种基于HBase集群架构方案。同时，为了提升查询效率，构造了一种复合主键生成方式。其次，分析了Redis内存数据库在缓存方面的优势以及现有的应用技术，设计了一种基于Redis的缓存的键-值存储模型。在此研究结果上，提出了一种基于热度值积累的缓存淘汰策略。
3. 在上述理论支撑下，基于本文设计的基于HBase集群服务器的交通流数据实时存储与查询系统，在数据写入模块实现了数据预处理、缓冲队列和数据并发写入。在数据查询模块实现了SQL解析、本地缓存和基于Redis的集群服务器。
4. 为了验证本系统的可行性，搭建一个Hadoop+ZooKeeper+HBase分布式存储系统和KeepAlived+TwemProxy+Redis分布式缓存服务器进行相关测试，并对实验结果进行分析。实验证明，在数据写入方面，使用本系统搭建的集群能承受比原生HBase集群或传统关系型数据库更高的并发写入请求，在数据查询方面，比传统HBase集群效率提升约25%。

1.3国内外研究现状

大数据[[[12]](#endnote-12)][[[13]](#endnote-13)]通常是指数量级达到PB级（1024TB）、EB级（百万TB）或更高的数量级的数据，包括一些常用的传统型数据库无法处理的较为复杂的半结构化和非结构化数据集。大数据的到来伴随着对传统关系型数据库的挑战，同时，也带来了机遇，为了解决关系型数据库峰值性能差、容错性和可扩展性差等问题，非关系型的数据库[[[14]](#endnote-14)]（简称NoSQL）应运而生。

对于海量数据集、高负载请求或者特殊的数据类型，如半结构化或非结构化数据等，非关系型数据库相比传统数据库有较大的优势。目前NoSQL数据库是学术界和产业界热门话题，出现了较多比较成熟的研究成果和应用。NoSQL数据库设计思路主要来源于Google的BigTable论文理论和Amaze的Dynamo[[[15]](#endnote-15)]分布式存储系统实现。BigTable是一个分布式的、以列为基准的、支持版本控制的多维表格系统，将一个表的数据按照主键进行分片，是一个强一致性的系统。Dynamo是一个分布式存储系统，以一致性哈希算法均匀划分数据，具有更好的稳定性、可用性和故障恢复能力。

目前较为成熟的NoSQL中，按照数据模型分类，共可以分为四类：

1. 键-值存储系统：键-值存储系统中值使用键值映射的关系保存数据，将键作为索引，将键对应的值作为具体的数据存储，它可以存储非结构化的数据，开发人员需要自行处理值的解析。常见的键-值存储系统，如Redis[[[16]](#endnote-16)]、Dynamo等。
2. 面向列存储系统：传统的关系型数据库是面向行进行存储的，如MySQL、Oracle等。在部分非关系型数据库中，数据是以列为单位进行存储，将相同列的数据存储在一起，支持列的动态扩展。常见的面向列的存储系统，如HBase、Cassandra[[[17]](#endnote-17)]和BigTable的其他实现。
3. 文档存储系统：与键-值存储系统不同，文档存储系统将键映射到包含一定信息的文档之中，文档中存储着具体的数据信息，文档的格式由开发人员自由指定。常见的文档存储系统，如MongoDB[[[18]](#endnote-18)]、CouchDB[[[19]](#endnote-19)]等。
4. 图像存储系统：图像存储系统使用图的特性对数据进行存储，使用节点、边和属性的关系描述数据。相比关系型数据库，图像数据库存储具有图关系的数据效率较高，且可以直接映射图关系。常见的图像存储系统，如Neo4J[[[20]](#endnote-20)]等。

作为Google Hadoop生态圈的一员，HBase作为典型的非关系型数据库收到各大互联网公司的青睐。为了提高HBase的查询效率，一些商业公司将缓存索引部署在HBase数据库上，常见的有华为公司的Hindex[[[21]](#endnote-21)]、NGDATA公司的HBase-Indexer[[[22]](#endnote-22)]和Patras大学的Interval Index[[[23]](#endnote-23)]等。

Hindex：华为公司的Hindex针对HBase查询设置二级索引方案，提供了一种声明式的索引策略。Hindex使用协处理器对索引进行创建和维护，客户端和开发人员无需对索引进行管理。同时，Hindex优化了索引行键，将索引数据和实际数据分布于同一个集群区域中，提升查询性能。Hindex查询需要访问所有的数据存储区域，由于大部分查询请求目标数据较少，在分布式系统中执行查询指令会导致许多未存储目标数据的区域也执行了查询过程，耗费较大的计算资源，降低了检索效率。

HBase-Indexer：HBase-Indexer，全称为Lily HBase Indexer，是NGDATA公司为了将Lily系统中相关HBase数据存储到Solr[[[24]](#endnote-24)]中而研发的中间件索引。其原理是将HBase中数据更新以异步的形式发送至索引服务器上，索引服务器进行数据分析并生成索引数据后，将索引数据推送至Solr Cloud中。在处理查询请求时，会首先进入Slor服务器进行数据定位，根据定位结果访问HBase集群。由于数据通过异步进行更新操作，因此无法满足某些实时性高的业务需求。

Interval Index：由希腊Patras大学研究的Interval Index主要解决HBase在非主键范围查询时效率低下的问题。它通过Hadoop MapReduce构建一个线段树[[[25]](#endnote-25)]索引结构存储在内存中，并将线段树中的索引上界保存在HBase中，以支持高效的范围查询。Interval Index对于范围查询效率较高，但对于少量记录的单点查询，则线段树的开销和维护成本较大，使用范围有限。

1.4本文组织结构

本文共包含六个正文章节，各个章节具体组织结构如下：

第一章为绪论。主要介绍了本文所立足的研究背景、研究的核心内容以及现阶段国内外相关领域研究现状。介绍了大数据在现阶段处理交通流数据方面的应用与发展，阐述了将大数据应用于智能交通领域的必要性。

第二章为相关技术介绍。介绍了实现交通流数据实时存储与查询系统所需要掌握的部分技术要点。主要包括Hadoop分布式计算框架、HBase分布式数据库的数据结构与集群架构、Redis相关特性与数据类型等。

第三章为交通流数据实时存储与查询系统的设计方案，首先介绍了道路车辆信息数据模型，然后依次介绍了HBase集群模型和Redis缓存服务器的设计方案。

第四章为交通流数据实时存储与查询系统的实现，分为数据写入模块和数据读取模块的实现。数据写入模块主要介绍预处理层、缓存层和数据写入层，数据读取层主要实现SQL解析、本地缓存和基于Redis的缓存服务器的实现和Cache Client的实现。

第五章为实验设计，根据第三章和第四章的理论指导，进行实验分析和结果对比，用实验数据验证系统的合理性。

第六章为总结与展望，对本文所描述的交通流数据实时存储与查询系统做出全面总结，并展望下一步的研究工作。

第二章 技术相关

交通流数据实时存储与查询系统是在HBase现有集群的基础上进行开发与扩展的系统，因此，本章首先对Hadoop和HBase的相关概念进行基本的介绍，介绍HBase系统的架构、数据存储模型、HBase缓存模型等。接着，由于交通流数据实时存储与查询系统采用Redis作为缓存系统，本章同时介绍了Redis相关技术。本章的最后，将简要介绍其他主流的NoSQL技术。

2.1 Hadoop技术

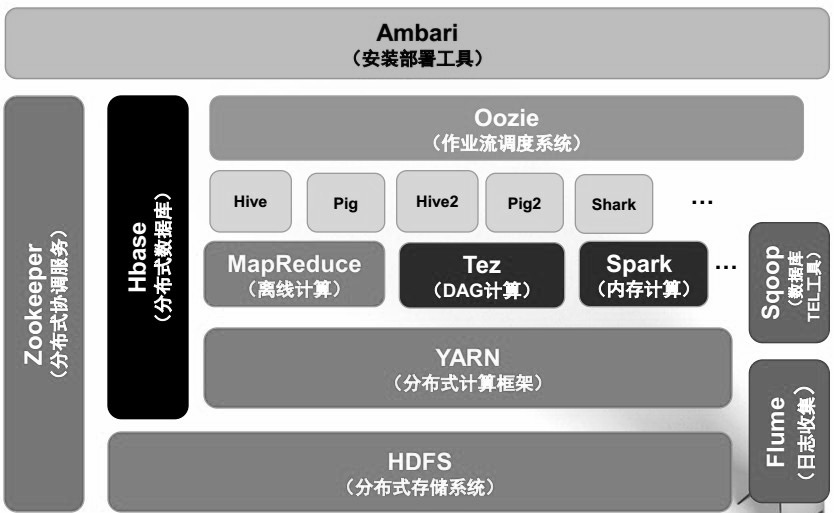
Hadoop是由Apache基金会开源的进行分布式并行计算的大数据处理基本架构，两个核心特性为HDFS[[[26]](#endnote-26)]和MapReduce[[[27]](#endnote-27)]，它可以在数千台并行的服务器集群上实现高度自由扩展，每台服务器都可以进行数据计算和资源存储。用户无需关注Hadoop计算框架底层实现情况，即可充分的利用该框架进行高效的计算和存储。Hadoop生态系统如图2.1所示：

图2.1 Hadoop生态系统

2.1.1 HDFS

Hadoop分布式文件系统（Hadoop Distributed File System，简称HDFS）的理论来源为Google发表的论文。HDSF通过数据流式访问模式进行大容量文件的存储和优化，运用于廉价的商用集群上[25] [[[28]](#endnote-28)]。主要体现在如下方面[[[29]](#endnote-29)]：

超大文件：相比其他类型分布式文件系统，如KASS、DFS等，HDFS在处理大体积的文件上有较大的优势。HDFS支持数百MB，数百GB，甚至TB级的文件。

流式数据访问：一次写入，多次使用是高效的访问方式。数据集通常是由数据源生成或复制而来。读取完整的数据集的时间效率比读取第一条记录的时间效率更重要。

商用硬件：HDFS在商业用途部署上，主要面向的对象是廉价的商业服务器。它可以稳定的运行在大量廉价商用服务器上。当单个服务器遇到故障时，HDFS被设计成能够自动转移至运行正常的其他集群服务器而不被用户察觉。

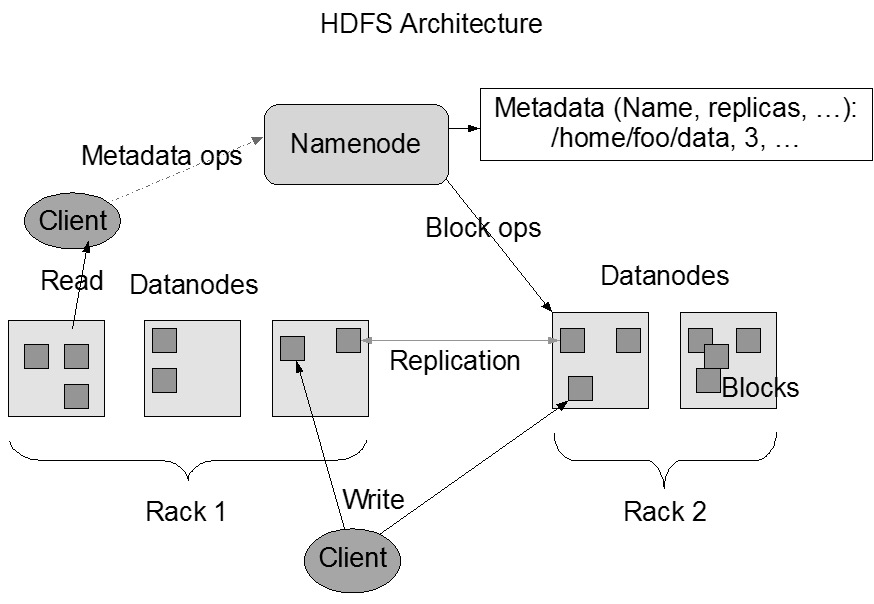
HDFS体系结构如图2-2所示：

图2-2 HDFS体系结构

如图所示，HDFS是用户（Client）采用由NameNode和DataNode构成的主-从（Master-Slave）模式运行。

NameNode：即主-从模式中的主（Master）角色，负责管理并维护整个文件系统的文件和目录，协调管理主-从模式中的从（Slave）角色，即DataNode，指导DataNode执行底层IO操作。NameNode作为Master节点自身不进行任何业务数据的存储操作，只负责统一管理数据表元数据信息，定期与各个DataNode通信以获取集群最新状态。

DataNode：即主-从模式的从（Slave）角色，是文件系统的工作节点，它接受NameNode统一管理检索和存储数据内容，并周期与NameNode通信更新节点数据。

Client：用户角色，用户通过Hadoop客户端或编程语言对文件系统进行访问，通过NameNode和DataNode协调通信实现文件的操作。

2.1.2 MapReduce

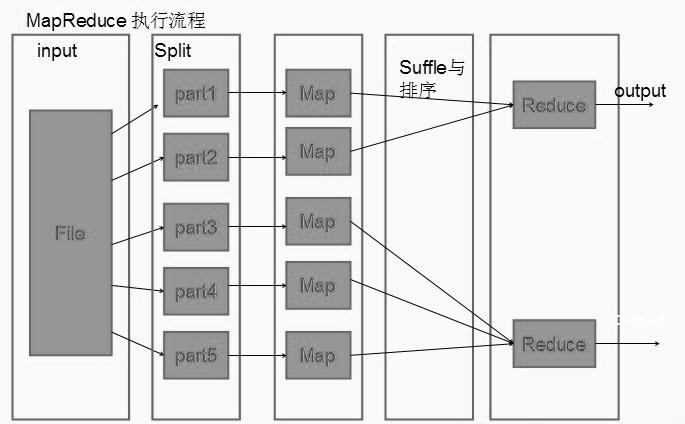
MapReduce是Hadoop生态系统中用来大数据处理的一个分布式编程模型和计算架构，是来自于Google提出的MapReduce论文理论[[[30]](#endnote-30)]的开源实现。MapReduce处理大数据集的过程如图2-3所示：

图2-3 MapReduce执行流程

MapReduce采用“分而治之”的思想，MapReduce将运算任务的处理流程简化为两个步骤：Map与Reduce，Map是将一个大规模数据集的任务，分发给各个TaskTracker协同运算，TaskTracker由JobTracker统一管理调度。Reduce则是将分解后多任务处理的结果整合，得到最终结果[[[31]](#endnote-31)]。

2.2 HBase技术

HBase起源于Google的Chang等人发表的论文“BigTable：A Distributed Storage System for Structured Data4”。2008年初，Hadoop成为Apache基金会顶级项目，HBase成为Hadoop子项目。2010年6月，HBase成为Apache基金会顶级项目。随着版本迭代，其项目的稳定性、高扩展性、安全性及功能方面均有较大的完善，随着互联网的发展和Hadoop生态圈的日益完善，Hadoop已经成为了海量数据处理的最佳解决方案，HBase作为Hadoop生态圈中的分布式数据库也崭露头角。目前HBase稳定版本为1.3.0。

2.2.1 HBase集群架构

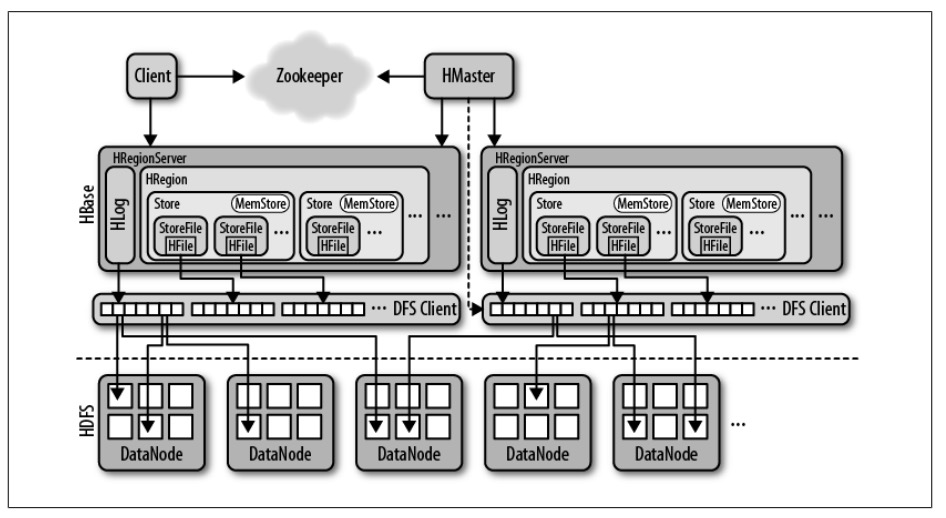
和Hadoop架构类似，HBase也是采用主-从（Master-Slave）管理模式。图2-4为HBase 架构图：

图2.4 HBase架构图

如上图所示，HBase主要包含Zookeeper集群、HMaster节点和若干个Region Server节点、HLog日志系统、数据存储底层采用HDFS文件存储模型，Client访问客户端等3。

1. Zookeeper

Zookeeper是成熟的开源分布式应用程序协同服务组件[[[32]](#endnote-32)]，它为Hadoop和HBase提供强一致性保证，是Hadoop分布式管理不可缺少的一个模块，主要用来进行统一集群管理、统一命名服务、集群配置管理、分布式协调、消息队列等操作。

1. HMaster

HMaster3是HBase主-从模式的主（Master）角色，即主控节点，类似于HDFS的NameNode。控制并管理着HBase集群，监控所有RegionServer集群实例，维护数据库的元数据信息，负责集群负载均衡，分配Region等操作。

1. Region Server

Region Server3是HBase主-从模式的从（Slave）角色，是HBase核心模块，主要响应用户的I/O请求，并在HDFS中读取或写入数据，维护Region信息。Region Server内部管理了若干个Region对象。如果Region过大，Region Server会对其进行拆分。

1. HLog

HBase在处理写入数据操作时，采用LSM架构[[[33]](#endnote-33)]，首先写入内存中，为了保证数据一致性，每个Region Server在处理更新数据请求（Put操作、Deletes操作） 时，会先记录到Write Ahead Log[[[34]](#endnote-34)]（WAL）中，然后再进行更新，当Region Server遇到突发情况时，可以通过Log进行恢复操作。HLog是HBase的一种WAL的实现，每个集群服务器都包含一个HLog日志实例。

2.2.2 HBase数据模型

HBase是Google论文中BigTable的开源实现，采用和BigTable类似数据存储模型。HBase底层利用HDFS采用Key-Value键值对存储。相比其他NoSQL数据库，HBase的Key不只是纯自定义字符串，而是由多个属性构成：行健（Row Key）、列族（Column Family）、列限定词（Column Qualifier）和时间戳（Timestamp），HBase键值对结构如式（2.1）3所示：

|  |  |  |
| --- | --- | --- |
|  |  | （2.1） |

1. 逻辑存储视图

HBase以数据表的方式存储，表由数据行（Row）和数据列（Column）构成，数据表划分数个列族（Column Family），表2-1所示的是一张HBase表的逻辑存储视图，通过它可以更方便的描述HBase数据模型：

表2.1 HBase逻辑存储视图

|  |  |  |  |
| --- | --- | --- | --- |
| **Row Key** | **Timestamp** | **Column Family（HTML）** | **Column Family（HREF）** |
| “site” | T5 |  | anchor:cnnsi.com=”CNN” |
| T4 |  | anchor:my.look.ca=”CNN.com” |
| T3 | contents:html=”<html>…” |  |
| T2 | contents:html=”<div>…” |  |
| T1 | contents:html=”<head>…” |  |

如上表所示，表中的“site”字段表示Row Key，T1，T2..等表示Timestamp，表中共有两个Column Family，分别为HTML列和HREF列，每个Column Family都有各自的Qualifier，表中“contents”和“anchor”分别为各自Column Family的Qualifier。通过Row Key、Column Family和Qualifier可以确定一个CELL，再根据Timestamp可以确定一个版本，例如{site,HTML:contents,T1}→“<head>…”。HBase 表格具有稀疏性，上表中部分列为空值，不占用存储单元。

下面逐个介绍HBase数据模型特性：

1. Row Key

HBase中主健（Row Key）相当于传统关系型数据库表中的主键，在数据表中具有唯一性且非空。每个Row Key对应着一条记录。HBase数据表通过Row Key的不同进行排序，依次划分至不同Region，从而实现分布式的数据存储和查询。Row Key长度应设计在合适的范围之内，不宜过长，Row Key的长度影响检索效率[[[35]](#endnote-35)]。

在数据查询方面，HBase提供了三种不同的查询方式：通过定义单个Row Key进行单点查询，通过设置主键范围（Row Key Range）进行范围查询，若用户未具体指定Row Key的值，HBase则采用全表“扫描（Scan）”的方式检索满足条件的数据。由于HBase采用类B+树模型存储，对于第一种和第二种检索方式效率较高，而对于第三种检索方式，则需要耗费较长时间，效率较低。因此在设计Row Key时，需要充分考虑检索效率，进行快速读取，减少读写次数。

1. Column Family & Column Qualifier

HBase 每张数据表中都有若干个Row Key，每一个Row Key都表示一条记录，每条记录有若干Column Family构成，每个列族里有若干个列，每个列只属于唯一的列族，每个列都有列修饰符Column Qualifier用来标识不同的列，通过Column Family：Column Qualifier可以定位到待检索的列。

1. Timestamp

时间戳用来区分不同版本的数据信息，Timestamp默认值为数据通过新增或修改写入数据表中的时间，可以作为版本号。Timestamp是64位整型数据，精确到毫秒，由系统自动根据写入时间生成或用户指定具体值。

1. Cell

元素是由HBase Key-Value确定的唯一的单元。元素本身并不存在任何数据类型，由字节流构成。

1. 物理存储视图

HBase物理模型采用Column Family为单位 进行数据存储，如表2-2所示：

表2.2 HBase物理存储视图

|  |  |  |
| --- | --- | --- |
| **Row Key** | **Timestamp** | **Column Family：HTML** |
| “site” | T3 | contents:html=”<html>…” |
| T2 | contents:html=”<div>…” |
| T1 | contents:html=”<head>… |
| **Row Key** | **Timestamp** | **Column Family：HREF** |
| “site” | T5 | anchor:cnnsi.com=”CNN” |
| T4 | anchor:my.look.ca=”CNN.com” |

表2-1逻辑存储视图中，部分空值是不被存储到物理视图中的，如表2-1中Column Family为“HTML”，Timestamp为T5的记录值不存在，在表2-2中不作为存储对象。当HBase进行数据检索时，若指定了Timestamp，则按照指定的Timestamp返回记录，若未指定Timestamp，则按照最新记录，即最后写入HBase 数据表中的数据返回。

2.3 Redis技术

Redis（Remote Dictionary Server）是一种开源、支持网络、基于内存、键值对Key-Value结构的高性能数据库35，采用ANSI C语言编写，是目前主流内存数据库之一[[[36]](#endnote-36)]，也是本文中缓存技术所采用的数据库。本章将对Redis技术做一个全面的研究。

2.3.1 Redis概述

Redis是一款开源的、高性能键值对数据库[[[37]](#endnote-37)]，提供了强大的客户语言支持，包括C/C++、Ruby、JavaScript、PHP、Python等，对用户来说，无需考虑语言的限制，易上手。Redis的键支持多种数据格式且这些数据类型都完整的支持Redis数据库的原子性操作：新增、删除、查找、交集、并集等。Redis为了保证系统的性能，将内存作为介质，Redis执行的操作会先被记录在内存中，根据事先设定好的持久化周期写入硬盘。

Redis是一个高性能的数据库，在Redis官方给出的测试报告15中得知：在Linux 2.6，Xeon X3320 2.5GHZ环境配置下，SET操作频率为110000次/秒，GET操作频率为81000次/秒。

除丰富的数据格式、性能高等之外，Redis还具有持久化、主从复制等特性，是一种优秀的缓存服务器选择方案。

2.3.2 Redis数据类型

Redis使用一个redisObject对象管理所有的key和value值，如图2-5所示。redisObject由数据类型（Type）、编码方式（Encoding）、数据指针（Ptr）、虚拟内存（Vm）和其他信息（Other）组成。Type表示Redis中键值对象的数据类型，Encoding表示不同的数据格式在Redis中的存储方式。Redis支持五种数据类型，分别是：哈希对象（Hash）、列表（Lists）、无序集合（Sets）、有序集合（ZSet）及字符串（String）[[[38]](#endnote-38)]。

图2.5 Redis核心对象——redisObject

1. 字符串类型

字符串（String）类型是各个数据库中最常见的一种数据存储类型，Redis同样也支持字符串类型。它能存储任何形式的字符数据，甚至是二进制数据。通常情况下，键值也默认是字符串类型，但Redis还支持数值（Int）型。另外，Redis也为字符串类型提供了丰富的操作命令。

1. 哈希对象类型

哈希对象（Hash，也称散列）类型是一种特殊的、使用类似字典对象结构[[[39]](#endnote-39)]的、以键值对的形式存储的数据类型。哈希对象存储的是键和键值的映射关系，一个哈希对象类型的键能包含不超过232-1个数据字段。在Redis数据库中，哈希对象类型的字段值只支持String类型，不支持其他数据类型。

1. 列表类型

列表（Lists）类型是Redis中常见的一种数据存储类型，可以存储一系列有序字符串列表，从底层设计和实现角度来说，Redis通过双向链表[[[40]](#endnote-40)]（Double Linked List）的方式实现数组。常见的操作是往数组头部或尾部追加记录，或者检索数组中某个片段值。通过向列表类型两端追加新记录时，时间消耗复杂度为常量级，即O(1)。当通过索引检索片段值时，访问速度通常较慢。由于Redis列表的特性，可以将Redis作为消息队列[[[41]](#endnote-41)]使用。与哈希对象类型相同，列表类型原则上能包含不超过232-1个数据字段信息。

1. 无序集合类型

集合（Sets）类型是一种无序的列表数据类型，在集合中，每个元素都是互异且无序的，与散列和列表类型相同，集合类型能包含不超过232-1个字段。表2.3列出了部分集合与列表的异同点。

表2.3 集合类型和列表类型异同对比

|  |  |  |
| --- | --- | --- |
| **特性** | **集合（Sets）** | **列表（Lists）** |
| **最大存储量** | ≤232-1 | ≤232-1 |
| **有序性** | 否 | 是 |
| **唯一性** | 是 | 否 |

从底层设计和实现角度来说，集合类型在Redis中是通过值为空的散列表（Hash Table）实现的，因此，当执行数据添加、数据删除、非空数据判断等操作时，时间消耗复杂度为O（1），可以进行的高效的交、并、差集操作。

1. 有序集合类型

Redis支持两种类型的集合：无序集合（Sets）和有序集合（sorted Set），Zset是在集合的基础上，为集合中的每一个元素增加了权重，使元素实现了排序的功能。ZSet不仅拥有Sets的功能操作，还可以排序相关操作：TOP-N元素、指定范围内的元素等。

ZSet在某些方面和Lists类型相似：

* 1. Lists类型和ZSet类型中记录都是有序的。
  2. Lists类型和ZSet类型都可以查询某一片段的数据值。

但对于两者的应用场景，却有着很大的区别：

1. 由于Lists类型在Redis底层的设计和实现是链表，因此在操作链表头部和尾部数据时，速度较快，而随着数据量的增加，维护链表中间的数据却较为困难，因此，Lists类型适合作为记录“日志”类型数据库使用。
2. ZSet类型底层的设计和实现是通过散列表和跳跃表（Skip List）实现的。因此在维护有序集合任意位置数据效率都较高。时间复杂度为O(log(N))。
3. 有序集合在空间复杂度上高于列表，会耗费更多内存资源。

2.3.3 持久化

Redis将内存作为存储介质进行数据存储，若没有持久化操作，一旦发生意外宕机，则会导致数据丢失。Redis支持两种方式的持久化操作：Snapshot（RDB）和Append Only File（AOF）337。

1. RDB

定时快照，又称为RDB，即在指定的时间间隔内，将Redis数据以快照的形式持久化成文件存储到磁盘硬件介质上的方式[[[42]](#endnote-42)]。RDB是一个非常紧凑的文件，它保存的数据是某个时间节点的数据集。RDB经常被用来进行数据备份，在指定的时间间隔，例如每12小时，备份一个RDB文件。这样即使遇到问题，也可以还原到指定版本的数据集。RDB持久化方式相对高效，但数据完整性无法保证，虽然RDB允许设置不同的保存节点，但是由于RDB保存的是整个数据集，处理时间较长，因此，即使每隔5分钟执行一次RDB操作，但仍然有一定概率会意外宕机而导致丢失当前5分钟数据。

1. AOF

附加指令文件，又称AOF，即将Redis操作请求的指令通过追加的形式存储到日志记录中，一旦Redis发生意外宕机，只需要读取日志文件，并顺序执行一遍指令即可。开启AOF功能之后，默认会每隔一秒执行一次写入操作，即FSync，若发生意外宕机，Redis提供了一个工具“redis-check-aof”可以对不完整的日志进行修复操作.随着Redis使用时间的增加，AOF日志文件体积也会变得很大，为了避免文件随着数据量变大情况发生，Redis实现了重写（Rewrite）机制，即当日志文件达到了预设的阈值时，会启动内容压缩，只保留能保障数据恢复的最小可用指令集。

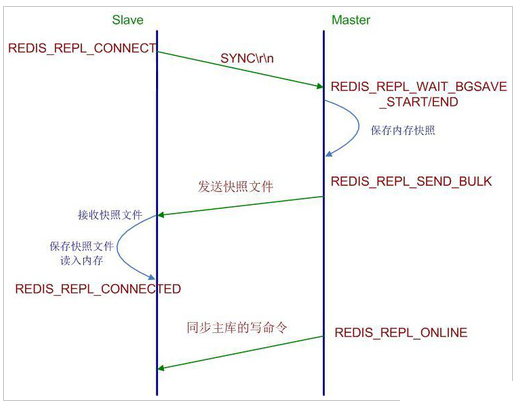
Redis允许同时启用RDB和AOF，既保证了数据的安全性，又使得数据备份简单，保证了Redis的操作性能[[[43]](#endnote-43)]。

2.3.4 主-从复制

通过RDB和AOF持久化操作，Redis保证了即使在意外宕机情况下，数据也能保证不被损失（少量损失），然而数据和日志是备份在本机服务器上，若服务器硬盘出现问题，则会导致日志和数据丢失，为了解决单机硬盘故障导致的数据丢失无法提供服务的问题，和MySQL主从同步类似，Redis也提供了主-从（Master-Slave）复制功能，可以自动同步数据至其他服务器实例。Redis主-从复制时序图如图2.6所示。

Redis的主从复制依赖持久化的RDB策略，在Master服务器接收到Slave的同步请求之后，会执行一次RDB的持久化操作，并将生成的持久化RDB文件发送给Slave，Slave在收到备份文件之后，利用备份文件完成一次主-从数据全量同步。若Slave在同步的过程中，Master接收到了数据操作，则Master会将更新操作暂时记录，并在Slave完成RDB同步之后依次发送给Slave进行同步。

Redis的持久化功能和主-从复制功能有效的保证了数据的安全性。正是如此，Redis为交通流数据实时存储与查询系统提供了安全、可靠的数据缓存服务。

图2.6 Redis主-从复制时序图

2.4 本章小结

本章为论文的技术介绍部分，主要介绍了交通流数据实时存储与查询系统所关注的相关技术，先是对Hadoop进行简要概述，主要介绍了Hadoop两个核心模块：HDFS和MapReduce。在此基础上，对Hadoop分布式数据库HBase进行介绍，重点阐述了HBase集群架构和数据模型。然后，介绍了交通流数据实时存储与查询系统使用的缓存数据库——Redis，重点剖析了Redis的基础特性、数据类型、持久化和主-从复制等特性。

第三章 交通流数据实时存储与查询系统的设计

针对第一章绪论中提出了智能交通系统数据存储的需求，本文在第二章相关技术条件下，设计了一套基于HBase的交通流数据实时存储与查询方案。交通流数据涵盖的内容非常广阔，包括道路上交通流运行信息、车辆信息、交通事件信息以及基础设施信息四大类[[[44]](#endnote-44)]。本章首先对智能交通系统进行简要概述，然后分析本文研究重点道路车辆车牌数据的特性设计HBase存储模型，设计出一种高效存储车辆车牌信息方案，最后阐述了交通流数据实时存储系统的整体设计

3.1道路车辆信息数据模型

由于智能交通数据的种类多样性，本文主要研究对象为各个道路交通路口路段的经过车辆车牌数据，表3.1为原始的道路车辆车牌数据的数据结构。

表3.1 道路车辆车牌数据源结构

| **字段属性** | **数据类型** | **字段描述** | **长度值** |
| --- | --- | --- | --- |
| ID | Number | 逻辑主键 | 20 |
| CAR\_NO | Varchar | 车辆车牌记录 | 10 |
| COLOUR\_NO | Number | 车辆车牌颜色 | 10 |
| RECORD\_TIME | Varchar | 车辆识别时间 | 20 |
| POINT\_NO | Number | 监测点编码 | 10 |
| ROAD\_NO | Number | 车道号 | 10 |
| BUST\_PIC | Varchar | 近景图片地址 | 255 |
| DISTANCE\_PIC | Varchar | 远景图片地址 | 255 |

注：“COLOUR\_NO”字段采集自“车牌颜色编码”数据表。“POINT\_NO”字段采集自“监测点编码”数据表。“ROAD\_NO”字段采集自“车道编码”数据表。

3.2 系统整体架构

针对交通流数据海量、异构、高速、多源等特性，本文设计并实现了交通流数据实时存储与查询系统，系统主要分为两个部分：交通流数据写入模块与交通流数据查询模块。图3.1为交通流数据实时存储与查询系统的系统架构图。如图所示左半部分为数据实时写入模块，右半部分为数据查询模块。下面将从各个功能模块分别讲述本系统。

图3.1 系统架构

1. 数据存储模块

数据存储模块主要负责将多种输入源，如全球定位系统（GPS）、地磁棒、线圈、车辆监控等设备，或停车场等场地获取到的数据通过本系统实时同步至HBase集群数据库中。共分为五个模块：前端输入、数据预处理、数据缓冲、数据写入、数据库集群。

前端输入：前端输入是由用户发起的，由专业设备不断的将数据传入系统的数据预处理模块。

数据预处理：系统接收到来自前端的数据写入请求，需要对数据进行进一步的清洗[[[45]](#endnote-45)]处理。将杂乱无规则的数据处理为符合规范的、完整的能被保存的数据。首先将数据按预设规则进行分类，同一类型的数据归并到一起。其次，脏值处理[[[46]](#endnote-46)]，由于输入设备技术上良莠不齐，部分数据会有缺失或重复，若缺失非关键属性，则使用默认值填充，若缺失关键属性，则废弃当前记录值并告知前端用户。最后，将预处理完成的数据提交至数据缓冲区。

数据缓冲区：经过预处理过的数据进入数据缓冲区后，会根据数据类型的不同而进入不同的缓冲区队列，缓冲区可以对海量数据进行实时监控。如图3.2所示。数据缓冲区由若干个数据队列构成，为数据进行分流和缓存归类。每个缓冲队列均有一个阈值，当到达队列阈值或手动触发刷入（Flush）操作时，数据会进入数据写入模块。

图3.2 数据缓冲区架构图

数据写入模块： 当缓冲队列中的数据达到阈值或触发刷入操作时，数据会进入数据写入模块，记录会在本模块中，通过多线程写入的方式同时对数据进行写入操作，本模块会生成HBase所需的数据写入指令，生成策略见下文。

数据库集群：为系统提供数据持久化服务的模块，负责数据的存储及维护，数据经过预处理、缓存和写入操作之后，都会被存储到集群中，同时，数据库集群也负责处理数据查询请求。

1. 数据读取模块

数据读取是指用户使用SQL命令进行数据查询，并将数据从HBase集群返回给用户的过程。除数据库集群模块外，还有前端输入、语法与语义检查校验模块、查询管理模块和Redis缓存数据库模块。

前端输入：与写入模块不同，读取模块前端输入的内容为结构化查询语言，，查询管理模块会将标准SQL语言解析成HBase和Redis可以识别的命令。

语法与语义检查校验：检查前端输入的SQL语句是否标准规范，若不符合语法规则，则会取消当前查询请求并告知用户。

查询管理模块：查询管理模块可以与Redis或HBase交互，可以将标准SQL语言解析为Redis或HBase可执行的命令，具体见下文。同时，查询管理模块还负责维护Redis缓存数据库。

Redis缓存服务器：本系统的核心模块之一，负责缓存访问频率较高的数据，当用户发起查询请求时，会先经过Redis缓存数据库检索，如命中操作，则读取缓存值，若未命中操作，则进入HBase集群检索。

本地缓存：由于Redis缓存服务器采用分布式架构，对于少量访问频率较高的缓存记录，访问非当前服务器也会导致性能影响。因此，在Redis缓存服务器之前设置一个本地缓存模块，将小部分数据保存在当前服务器，提高查询效率。

本文接下来将重点讨论HBase模型结构设计、Redis缓存数据库设计及相关内容、SQL解析及相关内容和缓存淘汰策略等内容。相关数据接入设计可以参考相关文献[[[47]](#endnote-47)]，元数据设计可参考相关文献[[[48]](#endnote-48)]

3.3 HBase集群模型结构设计

3.3.1 Column Family

HBase存在列族的概念，在处理数据库查询时，首先定位到相应的列族。在新建HBase数据库时，需要提前定义好列族属性，与普通关系型数据库不同，HBase不需要提前严格定义列属性，HBase的Column Family支持动态添加列属性。

如表3.1所示，所有字段均为同一类型数据，即都是交通路口路段的监测信息，根据HBase权威建议，目前同一数据表下跨列族访问效率较低。出于对查询效率的考虑，较好的方案为将这些数据放到同一个列族内。为此，我们建立一个HBase数据表，在建立数据表时，指定唯一列族，该列族下有若干个列属性，分别对应表3.1中各个字段属性。

3.3.2 Row Key

目前，HBase支持三种数据检索方式：

1. 指定Row Key的单条记录查询
2. 定义Row Key范围查询记录片段
3. 不指定Row Key，执行扫描操作

对于情况1和情况2，HBase查询效率较高，而在数据量较大时，情况3查询效率较低。为了充分发挥Row Key在数据检索时的优势，满足交通数据流检索的实际需求，合理的Row Key设计是实现交通流数据实时存储与查询系统的重要因素，较好的Row Key设计不但能提升检索效率，降低冗余带来的性能问题，同时一定程度上也能解决“写热点[[[49]](#endnote-49)]”问题，提升写入数据的性能[[[50]](#endnote-50)]。

从表3.1得知，道路车辆车牌数据主要由车辆车牌记录、车辆识别时间、监测点编码以及其他相关监测数据构成，主要查询字段为车辆车牌记录、车辆识别时间、监测点编码。因此，在设计Row Key组成时，应该充分考虑车辆车牌记录、车辆识别时间、监测点编码这三者关系进行组合，将它们作为复合Row Key。

根据HBase检索策略，将数据拆分（Split）3到不同的Region可以提升并行查询的能力，在主键上设置一个前缀（Prefix），设置Region拆分策略，HBase会将记录保存至指定Region。

根据上述描述，表3.2展示的是一种交通流数据实时存储与查询系统的Row Key方案示例。

表3.2 Row Key设计方案示例

|  |  |  |  |
| --- | --- | --- | --- |
| **复合Row Key** | | | |
| **PREFIX** | **CAR\_NO** | **POINT\_NO** | **RECORD\_TIME** |
| 2017 | 沪B37D83 | 10300501 | 2017-03-05 00:01:33 |
| 2017 | 沪A065C8 | 20501808 | 2017-03-04 11:26:33 |
| 2017 | 沪B5692Q | 20502003 | 2017-03-04 11:20:47 |
| 2016 | 沪B87QCF | 10300501 | 2016-11-30 12:51:59 |
| 2016 | 沪B67C96 | 10100403 | 2016-11-30 08:13:01 |

该方案中，Row Key由前缀（PREFIX）、车辆车牌信息（CAR\_NO）、监测点（POINT\_NO）和记录时间（RECORD\_TIME）四个属性复合而成。在本例中，前缀根据当前年份自动生成，HBase会根据不同的年份将记录保存至不同的Region中。Row Key的格式如公式3.1所示：

|  |  |  |
| --- | --- | --- |
|  |  | （3.1） |

例如，<2017><沪B37D83>-<10300501>-<2017-03-05 00:01:33>表示查询一条车辆车牌记录为沪B37D83，于2017-03-05 00:01:33在编号为10300501的监测点记录，该条记录被保存在名为2017的Region中。该方案查询维度为车辆车牌信息，若需要将查询维度设置为监测点或者记录时间，则只需要将“POINT\_NO”或者“RECORD\_TIME”放置于“PREFIX”之后的第一个位置即可。该方案增加了前缀，避免了读写热点，数据分布更均匀，更适合存储交通流数据，能充分利用HBase集群的计算和存储性能。

3.4 Redis缓存服务器设计

3.4.1 需求分析

Redis是一款高性能的Key-Value数据库，但Redis在处理数请求时，采用单进程、单线程的方式执行16。因此，只运行一台Redis数据库实例，会受限于系统I/O、存储容量等限制，在处理大数据请求时，会导致进程阻塞，单机Redis无法承载海量的交通流数据查询请求，因此需要依靠分布式集群Redis。多机Redis为数据配置多个副本，当其中任意一个Redis实例宕机时，集群可以有效的保证不丢失数据和影响用户访问。集群Redis具有更高的可靠性、稳定性，性能也更强大[[[51]](#endnote-51)]。

目前各大公司为了解决单机Redis实例并发请求和可靠性的缺陷，自主研发了分布式的Redis集群解决方案，主要思路是“分片”（Sharding），基本思想是把一个数据库分成多个部分，放到不同的服务器上，目前有三种实现方案[[[52]](#endnote-52)]：

1. 客户端分片

客户端分片（Pre-Sharding）是指将数据分布到Redis不同实例的任务移交给开发人员实现。由开发人员指定具体的路由规则。该方案主要集中在客户端实现，主要方法为对Redis记录的key值进行散列计算，并根据计算的值选择对应的Redis实例。该方案完全由开发人员手动控制，每一个Redis实例的增减或调整，均需要 由开发人员手动操作完成，系统可维护性较差，难以定位故障发生点。

1. 代理分片

代理分片（Proxy），即使用第三方分布式中间件，为Redis提供一个代理层，将分片工作交给代理层处理。中间件接收到来自客户的数据请求时，会根据路由规则，将这些请求转发给对应的Redis实例处理并返回结果。Twitter公司基于代理分片的思想，设计并开源了一个基于Redis的分布式代理中间件TwemProxy[[[53]](#endnote-53)]，TwemProxy速度快，能够维护持久的服务器连接，支持代理到多个服务器，是目前范围内稳定性最佳、使用人数最多的分布式中间件。当然，TwemProxy由于自身也是单点，因此需要使用KeepLived实现高可用方案。

1. Redis Cluster

Redis Cluster16是Redis 3.0中推出的新特性。它与代理分片不同之处，Redis Cluster不存在中心节点，集群中每个Redis实例只负责部分数据的存储任务。Redis Cluster不采取一致性哈希方式存储，而是采用Hash 槽（Slot）方式，每个Redis Cluster共有16384个Hash Slot，具体每个数据保存至哪个Slot，则是由Key的CRC16编码通过对16384取模计算决定。Redis推出了Redis Cluster客户端程序，用户可以在客户端中向任意Redis实例发出请求，每个Redis实例负责读写数据，同时负责与集群中其他Redis实例交互，若数据不存储在当前请求的Redis实例中，则会定位至数据存储的Redis实例并引导客户端访问该Redis实例。

由于客户端分片技术维护成本较高不利于扩展，而Redis Cluset是Redis 3.0中新特性，技术尚未成熟。由文献[[[54]](#endnote-54)]可知，TwemProxy批量读写吞吐率远高于Redis Cluster单条的吞吐率，Redis Cluster与TwemProxy的单条读写吞吐量不相上下。Redis Cluster虽说相比TwemProxy少了分布式中间件，理论上会更高效，但是否适合实际生产环境，还需要慢慢验证。因此，本文选择TwemProxy作为交通流数据实时存储与查询系统的高性能缓存集群解决方案。

3.4.2系统架构

TwemProxy Redis缓存集群系统架构如图3.3所示。主要由客户端（Client）模块， KeepAlived[[[55]](#endnote-55)]模块、分布式代理中间件TwemProxy、监控模块和Redis数据库模块。

图3.3 Redis缓存集群系统架构

客户端模块：客户端是指用户通过命令向Redis集群发出读写命令，在本系统中，客户端为查询管理模块，由该模块作为客户向Redis集群发起写入或查询请求。

KeepAlived模块：客户端通过虚拟IP地址访问Redis集群，无需修改Redis Client代码，KeepAlived提供虚拟IP地址和端口[[[56]](#endnote-56)]。由于TwemProxy是单点访问，而KeepAlived可以将多个无状态的单点通过Virtual Internet Protocol（虚拟IP ，简称VIP）漂移的方式构建高可用服务。KeepAlived实现基础是虚拟路由冗余协议[[[57]](#endnote-57)](Virtual Router Redundancy Protocol，简称VRRP)。

分布式代理中间件：TwemProxy是一个轻量级的Redis单线程代理中间件，支持Memcached ASCII协议和更新的Redis协议，由Twitter开源，是缓存服务器集群管理工具，TwemProxy可以减少对Redis集群的连接数。通过TwemProxy可以将多台通用服务器水平扩张Redis，有效的避免单点故障问题，支持失败节点自动删除，TwemProxy支持一致性哈希，可以将客户端请求的数据通过计算分发到相应的Redis实例中。

监控模块：监控模块由Sentinel和Redis-Agent组成，Sentinel是Redis官方16推荐的Redis高可用（HA）解决方案。Sentinel主要功能为

1. 监控：监控Redis主-从服务器是否运行正常
2. 通知：通知系统管理员Redis主-从服务器出现异常，并给出异常报告。
3. 自动故障转移：若Redis主-从服务器的Master发生意外故障导致宕机，Sentinel会选择一台Slave实例升级为Master，并重新配置其他Slave与其同步。
4. 配置提供者：Sentinel监控并管理者Redis主-从服务器的配置信息，若发生故障，则会通过Redis-Agent提供新的Master的地址和配置信息。

Redis-Agent负责监控Sentinel是否正常运行，若Sentinel进行了自动故障转移，则将新的Redis主-从服务器的Master地址发送给TwemProxy，更新TwemProxy的配置文件并重启TwemProxy进程，保障缓存服务器的正常、安全、稳定运行。

3.4.3 Redis 键值设计

高效、精准的在缓存中命中数据是缓存设计的基础，将频繁访问的记录缓存至Redis服务器，可以大幅提升记录查询效率。为了避免HBase非主键查询时的扫描操作，需要为部分访问频率较高的数据建立索引。文献[[[58]](#endnote-58)]提出了一种高效Redis Key设计方案，即将非主键属性和属性值作为数据存储Key，本文在此基础上进行了改进，设计了一种针对交通流数据实时存储与查询系统的方案，如公式3.1所示：

|  |  |  |
| --- | --- | --- |
| **Key**： |  | （3.1） |
| **Value**： | <Index: [HBASE\_IDS] , Frequency: FRE , Count: VISIT> |

下面分别介绍Key-Value各个字段的含义：

Flag：表示索引标识符，用来标识当前所属的HBase非主键列的属性别名，若直接采用非主键列的属性作为索引标识符，会出现许多key值字符冗余情况，设计合理长度的Key值有利于提升查询效率，降低系统开销。本文建议采用HBase非主键列的属性别名作为索引标识符，例如，道路号信息ROAD\_NO别名为“rn”，记录时间POINT\_NO别名为“pn”等。

Record：表示当前HBase列字段的记录值，即在非主键查询过程中，访问次数较多的列的具体值，表3.3为一个简单的示例数据，表3.4列出所有可能的Redis Key。

表3.3 示例数据

|  |  |
| --- | --- |
| **ROAD\_NO** | **POINT\_NO** |
| 2013024 | 10300501 |
| 2014047 | 20501808 |

表3.4 所有可能Key

|  |  |
| --- | --- |
| **REDIS\_KEY** | **REMARK** |
| rn\_2013024 | ROAD\_NO=2013024 |
| rn\_2014047 | ROAD \_NO=2014047 |
| pn\_10300501 | POINT\_NO=10300501 |
| pn\_20501808 | POINT\_NO=20501808 |

从表中可以看出，对于不同列的不同属性值，均可以用唯一的Key来表示，从而保证了Redis Key的唯一性。也是缓存数据库检索的标准。

Index：该字段存储的是列表在序列化字符串后满足当前Key的HBase Row Key记录值，例如表3.4所示的记录1，Index表示“满足HBase数据表中列名为“CAR\_NO”，值为沪B37D83的所有记录的Row Key组成的列表”。本文定义“-”作为分隔符，若满足条件的Row Key为“沪B37D83”、“沪A065C8”、“沪B67C96”，则Index值为“沪B37D83-沪A065C8-沪B67C96”。当系统查找模块根据指令查找当前Key时，TwemProxy会将当前值返回，客户端会将返回的Index按分隔符进行分割，将基于非主键的查询拼接成基于Row Key的范围查询，提升查询效率。

Frequency：时间周期内积累的热度[[[59]](#endnote-59)]，热度值越高，则表示在固定时间内，当前记录值被访问的次数越多，反之则越少。热度值是考察缓存命中率的重要指标之一。

Count：当前时间周期内被访问的次数，当前缓存记录值每被访问一次，则Count步进1。当前访问次数是考察缓存命中率的重要指标之一

3.4.4 缓存淘汰算法

通常对于NoSQL数据库，当遇到数据检索和处理瓶颈时，设置缓存是一种行之有效的方法。缓存技术一直是众多互联网公司研究热点，通过将部分访问频率较大的数据存储在缓存数据库中，可以提高检索效率，降低系统性能损耗。然而，随着访问量的增加，越来越多数据被添加到缓存数据库中，导致缓存数据库容量愈发增大，命中率降低。因此，需要设置一个阈值，当缓存数据库容量达到该阈值时，将部分热度较低的数据淘汰出数据库中，这就是缓存淘汰算法。

目前主流的缓存淘汰算法共有三种，分别为先进先出置换算法、最近最少使用置换算法、最近最不常用置换算法。

1. 先进先出置换算法

先进先出置换算法[[[60]](#endnote-60)]（First In Frist Out，简称FIFO），其核心思想为最近进入数据库的数据最容易被访问，如果一条记录最早进入数据库，那么这条记录被访问的概率就越低。FIFO的算法淘汰依据为停留时间，停留时间越久的则越会被淘汰。由于该方法使用时间作为淘汰判定依据，命中率较低。

1. 最近最少使用置换算法

最近最少使用置换算法[[[61]](#endnote-61)]（Least Recently Used，简称LRU），其核心思想为最近被访问过的记录在将来被访问到的概率越大，反之概率越低。LRU算法淘汰的依据为最近访问时间，最近未被访问的数据，越容易被淘汰出数据库。LRU算法只考虑了最近访问时间，未考虑记录的访问频率。

1. 最近最不常用置换算法

最近最不常用置换算法[[[62]](#endnote-62)]（Least Frequently Used，简称LFU），其核心思想为在固定的时间窗口内，记录被访问的次数越多，则该条记录在将来被访问到的概率越大。LFU算法考虑到了访问频率对记录的影响，是命中率最高的算法，然而由于算法较为复杂，实现难度较大

HBase自身存在缓存设计，Region Server上共有两个缓存：MemStore和BlockCache。MemStore是写缓存，Region Server会为每个Region提供MemStore，当HBase触发写操作时候，会将数据写入MemStore中，当MemStore达到某个阈值（默认64KB）时，会启动写入程序，将数据持久化到磁盘中。BlockCache是读缓存，当客户端发起查询请求时，先进入MemStore查询是否存储待查数据，若未命中，则进入BlockCache查询，如若命中数据，则返回，若未命中，则进入磁盘读取数据，并将数据缓存到BlockCache中，以便后续使用。

当BlockCache的容量达到上限时，会启动自身缓存淘汰策略，HBase的淘汰策略采用LRU算法，LRU只考虑到了BlockCache中最近被访问的记录，未考虑到不同记录的热度不同。本文参考文献59中提出的基于热度累积的缓存替换策略，提出了一种供参考的缓存替换策略，基本思想是使用指数平滑的方式计算热度值，根据热度值进行缓存淘汰，热度计算公式见公式3.2。

公式3.2中，SCORE表示计算出的热度值，SCOREN-1表示前N-1个时间周期中，根据访问频次计算出的热度值。SCOREN表示根据公式计算出的当前时间周期中的热度值。visitCount为访问次数，即在当前时间周期内，记录被访问过的频度。countPeriod为时间周期。visitCount/countPeriod表示时间周期内访问次数/时间周期，表示当前时间周期记录被访问的热度值。α为权重系数，用来平衡当前时间周期记录被访问的热度值和前N-1时间周期的热度值之间的影响力关系，α越大，则表示当前时间周期记录被访问的热度值对SCORE影响越大，反之历史热度影响越大。

在客户端发起缓存记录查询时，先进入Redis缓存服务器进行记录查询，对命中的缓存记录进行访问次数计数，当查询周期达到countPeriod或缓存服务器容量达到阈值时，会触发缓存淘汰算法，首先根据公式3.2计算记录的当前时间周期内的热度值，然后根据热度值进行排序，将热度值较低的记录淘汰出缓存服务器中。

当前缓存淘汰策略考虑到了记录访问频次与将来被访问概率的关系，且考虑到历史时间周期内被访问的热度值的影响，因此可以避免频繁访问的记录值在短期未被访问就被淘汰出缓存服务器的问题。通过访问频次计算热度，能够较好的反映记录实际被访问情况，因此留在缓存服务器中的记录也更加合理。

3.5 本章小结

本章从系统架构和关键内容设计两个部分对交通流数据实时存储与查询系统做了详细介绍，首先，简单介绍了交通流数据和道路车辆信息数据模型，其次，对HBase分布式集群的系统，包括主键、列族给出了详细的设计方案，最后，在Redis缓存分布式服务器的设计方面，给出了一种常见的键值模型设计和缓存淘汰算法。

第四章 交通流数据实时存储与查询系统的实现

在上一章中详细介绍了交通流数据实时存储与查询系统的设计方案，主要包括系统整体架构设计、HBase数据模型设计和Redis缓存服务器设计等。本章将在此基础上，给出交通流数据实时存储与查询系统具体实现，主要分为数据写入模块和数据查询模块。

4.1 数据写入功能

4.1.1 数据写入流程

一个完整的数据写入流程的时序图如图4.1所示。

图4.1 数据写入时序图

如图4.1所示，一个完整的数据写入流程为：

第一步，客户端数据来源具有多样性，系统每秒处理的数据可能来自多个渠道，在进入下一流程之前，需要在预处理时对数据首先进行分类。以本文模型为例，数据来源共有三处：车辆全球定位系统、停车场数据和路口路段监控设备数据。可以在每一条记录上增加一个记录来源Type前缀，标识不同的记录来源，表4.1所示的是一种前缀示例。缓冲层可以根据前缀类型的不同，将记录转发至不同的缓冲队列中。

表4.1 记录来源前缀示例

|  |  |  |
| --- | --- | --- |
| **记录来源** | **前缀** | **示例** |
| 路口路段监控设备 | MONITOR | MONITOR\_粤B234PP#0#2015-08-… |
| 停车场设备 | DEPOT | DEPOT\_粤B08980-110203802-… |
| 车辆全球定位系统 | GPS | GPS\_粤BVM939@2015080... |

第二步，数据进入预处理层，需要进行“脏记录”处理45。“脏记录”处理共分三步：

1. 判断核心数据是否缺失，如车辆车牌信息、监测时间信息、监测点信息等，若这些记录信息缺失，则该条记录保存则没有意义，在预处理层中，会直接删除当前记录值不进行处理。
2. 非核心数据信息缺失，如车辆颜色信息、车型信息等非核心信息缺失，在预处理层会对该属性添加预设的默认信息或从其他来源中获取46。
3. 规范数据格式，由于不同数据来源的格式各不相同，统一编码规范有利于数据写入层进行写入操作，因此，预处理层需要将不同来源的数据按照预定预设的规则进行重新编码。

第三步，在完成预处理之后，数据会进入缓冲层，缓冲层由等待队列、各类型缓冲管理器、缓冲接收服务器共同组成，具体实现见下文。

第四步，当数据在缓冲队列存储，达到一定阈值或手动触发写入时，会执行数据写入操作，数据通过优化写入的方式持久化到HBase集群中。

4.1.2 数据缓冲层的实现

数据缓冲层共由四部分组成：缓存接收服务器、缓存管理器、缓存队列控制和等待队列。如图4.2所示。

图4.2 数据缓冲层示意图

下面讲述各个部分的具体实现：

1. 缓冲接收服务器

缓冲接收服务器（简称接收器）作用为接收预处理之后的规则数据，接收器需要先根据预定规则划分每条数据，并与规则字典中属性名对应，形成序列对象，表4.3为某规则字典示例，表4.4为经过该规则字典序列的示例。

表4.3 某规则字典示例

|  |  |
| --- | --- |
| **规则公式** | **备注** |
| {PREFIX}-{ CAR\_NO }-{RECORD\_TIME}-{POINT\_NO} | 前缀-车辆车牌信息-记录时间-记录地点 |

表4.4 示例记录与对应序列对象

|  |  |
| --- | --- |
| **记录值** | **序列对象** |
| MONITOR\_粤B234PP\_20501413\_2015-08-06 12:31:20 | PREFIX: MONITOR,  CAR\_NO: 粤B234PP,  POINT\_NO: 20501413,  RECORD\_TIME: 2015-08-06 12:31:20 |
| GPS\_粤S73C81\_20501411\_2015-08-01 10:36:20 | PREFIX: GPS,  CAR\_NO: 粤S73C81,  POINT\_NO: 20501411,  RECORD\_TIME: 2015-08-01 10:36:20 |
| DEPOT\_粤BFR326\_ 30503002 \_ 2015-07-08 00:00:51 | PREFIX: GPS,  CAR\_NO: 粤BFR326,  POINT\_NO: 30503002,  RECORD\_TIME: 2015-07-08 00:00:51 |

1. 缓冲管理器

数据变成序列对象之后，根据PREFIX的不同，被发送到不同的缓冲管理器中，缓冲管理器的作用为将不同的序列对象通过计算告知队列控制器该条数据的插入的队列序号。具体实现方式为一致性哈希[[[63]](#endnote-63)][[[64]](#endnote-64)]（DHT）。

哈希算法为将任意长度的二进制数据通过某种计算映射为固定长度的二进制值，该值被称为哈希值。判断哈希的好坏由四个标准：平衡性（Balance）、单调性（Monotonicity）、分散性（Spread）、负载（Load）。常用的线性哈希算法为：

|  |  |  |
| --- | --- | --- |
|  |  | （4.1） |

其中，N表示全部等待队列的个数。在缓冲层中，对等待队列的添加或者删除、或者队列故障之后自动脱离操作，都是缓冲层经常遇见的情况。公式4.1不足之处在于队列删除或者添加之后，部分数据会被丢失，该算法违背了单调性原则。

一致性哈希是目前较为主流的分布式哈希算法之一，它对简单的线性算法进行的改进，解决了线性哈希带来的热点问题。图4.3为一致性哈希算法示意图。

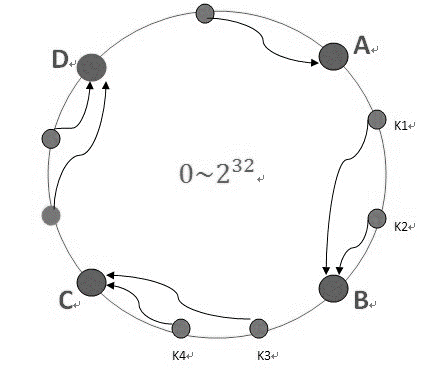


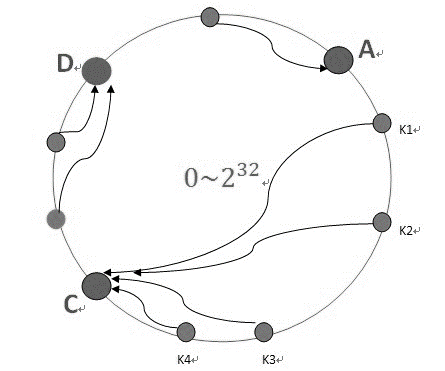
图4.3 一致性哈希算法示意图

如图4.3所示，假设一个缓冲管理器共有A、B、C、D四个等待队列，这四个队列被分配到0~232的一个环形上，我们将计算值为0~28范围存储到A队列，28+1~216范围存储到B队列，依次类推C、D。计算值方法如公式4.2所示。

|  |  |  |
| --- | --- | --- |
|  |  | （4.2） |

获取每条数据的记录时间RECORD\_TIME属性值，并对记录时间进行Hash计算获取散列值，对该值进行232取余数计算KN，根据KN的值按顺时针方向将其映射到离其最近的目标队列上。如图中K1、K2目标队列为B队列，K3、K4目标队列为C队列，以此类推。

当其中某个等待队列出现故障时，根据一致性哈希工作原理，会将其映射到顺时针方向上的第一个等待队列，受影响的数据仅仅为环上的故障队列和其上一个队列之间的数据值。如图4.4所示，当B队列故障时候，原先位于B队列的K1和K2将进入C队列中，此时C队列的范围为28+1~224。同样，若新增一个队列，则只需要将属于原队列的一部分数据添加至新队列即可。

图4.4 故障队列示例

一致性哈希算法通过减少队列意外故障影响范围的方式解决了队列增减导致的数据Hash问题，从而解决了并发环境下缓冲负载均衡的问题。

1. 缓冲队列控制

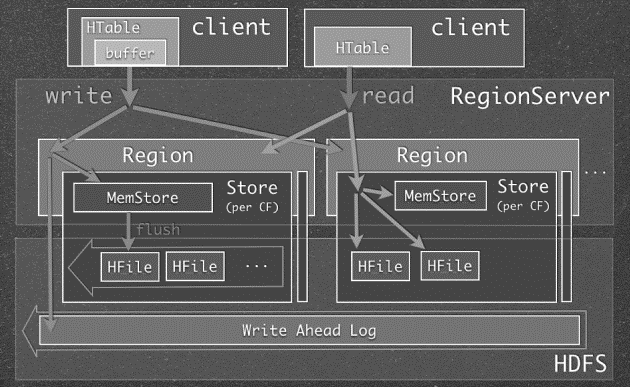
缓冲队列控制主要负责数据并发等待队列的维护、数据入队操作和监控队列Flush操作。目前有较多成熟消息队列中间件，如ActiveMQ[[[65]](#endnote-65)]、RabbitMQ[[[66]](#endnote-66)]等，均可以作为缓冲队列实现方案，本文将不再赘述。

1. 等待队列

等待队列是缓冲层的核心模块，主要负责待写入HBase数据的暂存操作，队列有两个操作，入队和出队，由缓冲队列控制模块统一管理。生产者将数据写入队头（入队操作），消费者从队尾读取数据（出队操作）。

4.1.3 数据写入层的实现

数据写入层实现从HBase MemStore多线程向HBase集群写入数据，写入示意图如图4.5所示。当Region Server收到客户端发来的写请求时，Region Server会将请求转发至对应的Region。Region中包含若干个列族和列，不同列族的数据存储在各自的HStore中，每个HStore由一个MemStore和数个HFile文件构成。MemStore位于Region Server主存中，而HFile作为持久化文件，存在于HDFS中，Region Server会先将写请求的数据保存至MemStore，当达到一定阈值时，再刷入至HFile中。

图4.5 HBase写入数据流程

在配置数据写入层时，需要关注HBase MemStore相关问题。首先，HBase不会根据开发者使用模式进行相关自动化调整，MemStore相关配置需要开发者自行手动配置以取得更好的性能。其次，Flush操作会严重影响HBase集群的Read性能，因此，MemStore需要配置适当的Flush条件。最后，Flush方式也需要开发者慎重选择，Flush方式可能会影响到HBase数据表的设计。

本文给出了一种较科学的MemStore配置方案，供开发人员参考。对于写入流程来说，主要配置写入操作触发的时机和阻断（Block）时机，具体配置如下：

对于普通Flush操作，这类Flush发生时，并不影响并行写请求，主要配置Flush Size和Lower Limit，属性名称分别为： hbase.hregion.memstore.flush.size和hbase.regionserver.global.memstore.lowerLimit，在配置MemStore大小时，需要综合考虑Region Server 能承载的Region总量，若设置Flush Size过小，随着Region数量的不断增加，Flush操作会越来越频繁。

对于高写入请求的系统，上述配置并不能完全符合要求，比如车辆信息数据，在某些路口路段交通复杂，车流量大，导致集群写负荷较高。此时我们希望MemStore不要超过安全写入设置，因此需要一种阻断机制，即当写负荷过大时，需要阻断写操作，直至MemStore恢复到“可管理”的大小。主要配置UpperLimit和Multiplier，属性名称分别为：hbase.regionserver.global.memstore.upperLimit和hbase.hregion.memstore.block.multiplier。应该尽量避免“写阻塞”的情况发生，“写阻塞”节点的存在会导致整个集群速度变慢。

压缩HDFS持久化的数据，如HFile等有利于节省硬盘空间和网络I/O操作。在MemStore进行写入操作写入HFile时，数据会被压缩，压缩操作对写入操作处理过程影响较小，但能有效的解决“写阻塞”问题。

4.1.4 HBase集群扩展

随着数据量的不断扩大，HBase集群节点容量不再满足日益增长的业务需求，此时需要使用新的服务器进行集群扩展。集群扩展流程如下：

1. 准备新服务器，并为服务器命名，如Slave5，配置免密登录SSH。
2. 修改Master实例的hosts文件，将新服务器IP地址和实例名追加至文件尾部。
3. 修改配置文件hbase-site.xml，在属性ase.zookeeper.quorum的value内容中增加新实例名。
4. 修改配置文件RegionServers，在末尾追加新实例名。
5. 安装Hadoop、HBase和Zookeeper运行环境，从Master中拷贝配置文件至各个Slave实例。
6. 关闭Slave5防火墙设置，关闭DHCP。
7. 重启Hadoop与HBase。

4.2 SQL解析

HBase作为非关系型数据库，本身不支持结构化的查询语言。为了提升本系统的易用性，我们为数据读取功能模块增加了SQL解析功能，实现用户只需写一条SQL语句，系统解析模块负责解析该SQL并生成对应的HBase查询指令，并在HBase中执行的效果。

4.2.1 Phoenix解析器介绍

交通流数据实时存储与查询系统主要的业务场景为查询，来自路口路段监控设备、全球定位系统、停车场等源头的数据被不断写入系统，当系统稳定运行后，数据写入模块无需太多人为干涉。因此，本文实现的SQL解析的重点为数据查询模块。

目前比较成熟的SQL解析方案，根据操作方式的不同，可以分为三类：

1. 以Map-Reduce为核心，单任务使用HBase客户端访问。
2. 以Google交互式数据分析系统[[[67]](#endnote-67)]（简称Google Dremel）为核心, 单任务使用HBase客户端访问。
3. 以HBase协处理器[[[68]](#endnote-68)]（简称HBase-Coprocessor）为核心，客户端合并多个节点的处理结果。

本文使用由Apache公司推出的开源HBase SQL解析工具——Phoenix[[[69]](#endnote-69)]，Phoenix是一个Java中间件，由saleforce.com开源，可以帮助开发人员使用熟悉的结构化查询语言操作HBase。为了最小化性能开销，Phoenix使用原生HBase APIs，并没有使用MapReduce框架，同时，协处理器被放置在服务器端执行，从而保证最小化客户端到服务器的数据传输。

本系统的SQL解析设计思路是通过将Phoenix放置在服务器上。经过解析后，得到HBase查询片段，然后拼装成HBase命令发送给服务器集群，拿到HBase返回的数据之后转发给客户端。

Phoenix支持从一个或多个数据表中检索数据，SQL语法与传统的结构化查询语言类似。

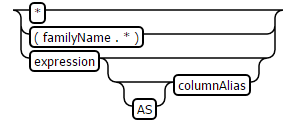
每个SELECT语句均有一个SELECT EXPRESS，用于指定返回数据格式，支持三种类型：“\*”表示选择Table中所有列信息。使用“Column Family.\*”用来选择数据表列族中的所有列。或使用一个表达式用来计算返回的数据。语法图如图4.6所示。

图4.6 SELECT EXPRESS语法图

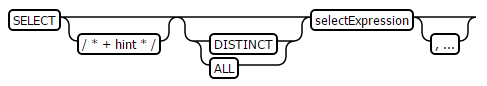
DISTINCT字段根据检索条件过滤掉所有重复的查询结果，ALL为默认显示所有查询结果，语法图如图4.7所示。

图4.7 DISTINCT与ALL语法图

FROM字段用来标识需要查询的数据表，语法图如图4.8所示。FROM字段后面可以引用一个或若干个任意定义别名的数据表，且该数据表可以定义数个动态列，语法图如图4.9所示，或定义一个新的SELECT子句并设置别名。Join Type为指定连接方式：内连接（Inner）、左外连接（Left Outer）或右外连接（Right Outer）。为了获得最佳性能，Phoenix建议依据预期每个表中使用行数从大至小进行排序。

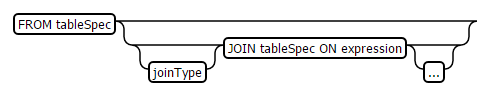
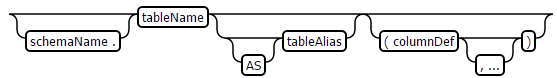
图4.8 FROM语法图

图4.9 TableSpec语法图

WHERE字段后面跟属性表达式，多个属性表达式之间可以用AND或OR关键词进行连接，语法图如图4.10所示。

图4.10 WHERE语法图

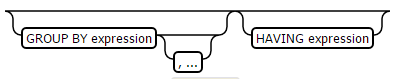
GROUP BY字段可以对结果按照指定表达式进行分组，HAVING字段可以在分组完成之后，进行过滤行操作，GROUP BY和HAVING字段后面分别跟一个表达式。语法图如图4.11所示。

图4.11 GROUP BY和HAVING语法图

另外，Phoenix支持多种聚合查询关键字，如MAX、MIN、SUM、COUNT等，具体使用教程可参考官方示例69。

4.2.2 Phoenix解析器配置

配置Phoenix流程如下所示：

1. 从Apache Phoenix官方网站69下载最新版Phoenix安装程序。
2. 将与本系统HBase集群相兼容的Phoenix服务器安装到每个Region Server的lib目录中。
3. 重启HBase集群。
4. 将Phoenix客户端 Jar包添加至HBase客户端的Class Path目录下。
5. 下载并安装Phoenix SQL客户端，以便于针对HBase集群制定执行计划。

4.3 缓存集群服务器的实现

本文仅示例一种基于TwemProxy+Redis+Sentinel+KeepAlived的Reids服务器集群搭建方案，服务器系统采用Cent OS 6.5 X64，具体搭建方案可参考网络文献[[[70]](#endnote-70)]或访问Reids官方网站16。

4.3.1 KeepAlived配置

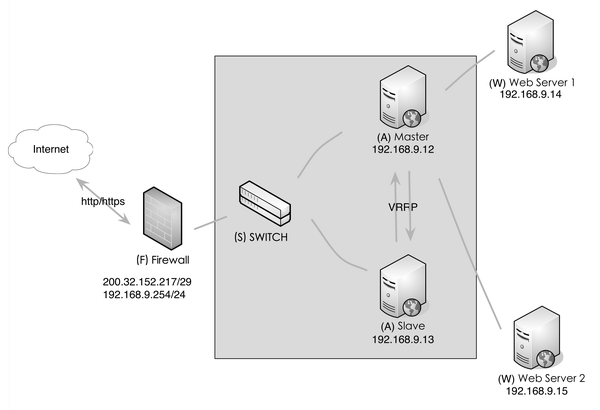
KeepAlived是常用作多机热备份需求的一款组件，作用于服务器的负载均衡层，利用其提供的浮动IP服务56，可以提高系统的可用性。图4.12为一个KeepAlived示例。

图4.12 KeepAlived示例

KeepAlived是基于VRRP协议57来实现的服务高可用方案55，KeepAlived主要有三个模块，分别为Core、Check和VRRP，其中，Core模块为KeepAlived的核心模块，主要负责进程管理、服务维护以及配置文件的加载和解析。Check模块负责KeepAlived的健康监测。VRRP模块实现虚拟路由冗余协议。

在VRRP协议实现里，虚拟MAC地址为00-00-5E-00-01-XX，XX为虚拟路由器标识符（Virtual Router Identifier，简称VRID），在同一时间，只有唯一一台物理设备占用该路由器标识符。物理路由器组通过IP地址（默认为224.0.0.18）定时推送消息。每个Router均有优先级，优先级别范围为1至255，高级别路由器将升级为Master。

KeepAlived服务安装流程如下，更具体的配置可参考网络资源：

1. 下载KeepAlived组件

[root@master-pc ~]# yum -y install keepalived

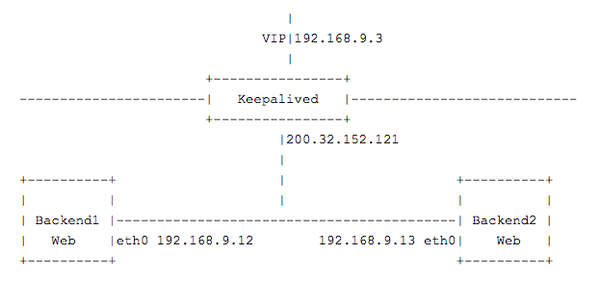
在本系统网络环境中，KeepAlived存在于内网IP，如图4.13所示。

图4.13 内网KeepAlived示意图

1. 配置内网IP，配置文件路径为/etc/sysconfig/，文件名为network-scripts。配置完成后重启network服务。配置信息如下所示：

[root@ master-pc network-scripts]# vi ./ifcfg-eth0

DEVICE=eth0

HWADDR=00:2B:00:2E:00:D0

TYPE=Ethernet

UUID=ffffffff-ffff-ffff-ffff-ffffffffffff

ONBOOT=yes

NM\_CONTROLLED=yes

BOOTPROTO=static

IPADDR=192.168.9.12

NETMASK=255.255.255.0

GATEWAY=192.168.9.254

[root@ master-pc network-scripts]# /etc/init.d/network restar

1. 配置iptables规则，配置信息如下所示。

[root@master-pc ~]# iptables -L

[root@ master-pc ~]# iptables -I INPUT -i eth0 -d 224.0.0.0/8 -p vrrp -j ACCEPT

[root@ master-pc ~]# iptables -I OUTPUT -o eth0 -d 224.0.0.0/8 -p vrrp -j ACCEPT

[root@ master-pc ~]# service iptables save

1. 配置KeepAlived配置文件，路径位于/etc/keepalived目录下，文件名为keepalived.conf。配置信息如下所示。

vrrp\_script chk\_nginx {

script "/etc/keepalived/check\_nginx.sh"

interval 2

weight -5

fall 3

rise 2

}

vrrp\_instance VI\_1 {

state MASTER

interface eth0

mcast\_src\_ip 192.168.9.12

virtual\_router\_id 51

priority 101

advert\_int 2

authentication {

auth\_type PASS

auth\_pass 1111

}

virtual\_ipaddress {

192.168.9.3

}

track\_script {

chk\_nginx

}

}

1. 配置完成后，启动KeepAlived：

[root@master ~]# /etc/rc.d/init.d/keepalived start

Starting keepalived: [ OK ]

此时将会看到KeepAlived相关启动信息，若想关闭KeepAlived或重启KeepAlived，只需将start参数替换成stop或restart即可。通过命令“ip addr show eth0”可以看到相关VIP信息，查看VIP是否绑定。至此，KeepAlived配置完成。

4.3.2 Redis Cluster搭建

在Master实例服务器和Slave实例服务器上分别安装Redis程序，主要流程如下所示：

1. 从Redis官方网站16上下载最新版Redis安装程序。目前稳定版本3.2。
2. 将Redis安装程序通过SCP命令分别部署至Master和Slave实例服务器上（需先开启SSH免密登录）。
3. 在Master实例服务器上，修改/etc/redis.conf文件内容，具体如下：

port 6379

bind 0.0.0.0

timeout 300

1. 在Slave实例服务器上，修改/etc/redis.conf文件内容，具体如下：

port 6379

bind 0.0.0.0

timeout 300

slaveof 192.168.9.\*\* 6379 // \*\*表示当前Master实例服务器IP地址

1. 启动Redis服务，并验证是否配置完成。

4.3.3 TwemProxy配置

TwemProxy的安装需要先完成Redis Cluster的安装，TwemProxy安装流程如下所示：

1. 从官方网站下载最新版TwemProxy安装包，并将安装包通过SCP命令复制到Redis的Master和Slave实例中。
2. 配置TwemProxy Master与Slave，修改/usr/local/twemproxy文件夹下的nutcracker.yml文件，修改内容如下所示。
3. 使用TwemProxy启动脚本启动TwemProxy Master与TwemProxy Slave。
4. 测试Redis服务是否正常。

Redis1:

listen:0.0.0.0:22122 //监听IP地址和端口号

hash:fnvla\_64 //指定key值的哈希算法

hash\_tag:”{}” //哈希标签，指定计算部分

distribution:ketama //一致性哈希算法

auto\_eject\_hosts:true //是否根据server的状态重建集群

redis:true //指定当前的服务器通讯协议

servers: //配置实例的IP地址与端口号以及权重

* 192.168.9.12:6379: 1 server1
* 192.168.9.13:6379: 1 server2

4.4 HBase与缓存的交互

4.4.1 本地缓存

在交通流数据中，有一类数据，如国家标准车牌类型数据等，具有访问频率高、数据量较小等特点，对于这类数据，交通流数据实时存储与查询系统提供了一种可配置开放的本地缓存（Local Cache）策略。区别于Redis集群服务器，本地缓存仅保存小数据量的数据，且该数据热度值较大。当客户端发起数据查询请求时，若配置开放了本地缓存，则会优先进入本地缓存检索。由于本地缓存仅在当前服务器保存，不涉及分布式集群管理，因此，性能和查询效率较高。

定义一个Node节点类，共保存三个属性：键（Key）、值（Value）、过期时间（ExpeireTime）。如下所示：

public class Node {

private Object key; //键

private Object val; //值

private long expeireTime; //过期时间

//省略GET/SET方法

}

由于本地缓存位于本地服务器上，为了适应更多的业务场景需求，需要对缓存容量进行严格控制，本系统中定义了一个缓存淘汰算法的接口IStrategy，开发者可以实现该接口提供自定义的缓存淘汰算法。IStrategy接口如下所示：

public interface IStrategy<E> {

public void insert(E node); //插入Node数据

public void update(E node); // 更新Node数据

public E delete(); // 根据淘汰策略删除数据

public void delete(E node); //指定删除数据

public void hitNode(E node); //访问Node数据

}

本文给出了LRU算法和LFU算法的简单示例。

LRU算法：使用LinkedList作为链表的实现。新增操作时，将Node节点放置到LinkedList顶部。更新操作时，首先删除LinkedList中原Node节点并将新的Node节点放置到LinkedList顶部。当进行删除操作时，若指定了Node节点，则删除对应的Node，若未指定Node节点，则删除LinkedList中最后一个节点。由于新增和更新操作发生在链表顶部，因此，位于LinkedList顶部的数据必然是最近使用的数据，反之位于链表尾部的数据为最少使用的数据，优先删除。部分关键见附录1所示。

LFU算法：由于LFU算法是以访问频次作为淘汰依据，首先定义一个LFU\_NODE，将Node和访问频次Count作为属性封装至FLU\_NODE中，代码如下：

class LfuNode<E>{

public E node;//Node数据节点

public int count; //访问次数

}

LFU算法与LRU算法类似，也采用LinkList作为链表的实现。当进行数据新增操作时，先申请一个LFU\_NODE实例，并将频次Count值设置为1，追加至链表的尾部。当进行访问操作时，遍历整个LinkedList，找到当前数据Node对应的LFU\_NODE，并将其频次Count自增操作。当进行更新操作时候，则遍历整个LinkedList找到对应的LFU\_NODE，替换Node节点并将频率Count自增。当进行删除操作时，若指定了具体数据Node，则遍历整个LinkedList，找到对应的LFU\_NODE节点并删除，若未指定具体数据Node，则找到Count值最小的LFU\_NODE移出链表。部分关键代码见附录1所示：

在Local Cache实现层，使用HashMap作为存储介质，将Node作为存储的值类型。在初始化Cache时需要配置两个属性：缓存淘汰策略（默认使用LRU）、缓存记录数（默认存储2000条键值对记录）。Cache提供了三个操作函数：SET、GET和Expire。

通过SET可以更新或者新增一条记录到本地缓存，调用SET方法时，首先会进入HashMap中查询有没有当前数据Node，若存在，则直接更新Node并调用缓存淘汰策略的UPDATE方法。若不存在，则判断当前记录条数是否超过CacheSize，若超出范围，则调用DELETE方法进行缓存删除，直到符合新增条件，创建Node实例并放入HashMap。GET方法可以根据Key的值获取对应的数据Node，针对缓存失效的问题，通常有两种解决方法。一种是消极方案，当Node节点被访问时，根据ExpeireTime判断是否失效。若失效，再转到Redis缓存服务器查询。另一种是积极方案，周期性的遍历本地缓存链表，若存在失效数据，则直接删除。本地缓存中GET方法采用消极方案，即先根据Key获取对应的Node对象，再根据Node中ExpeireTime判断是否失效，若失效，则从链表中删除该记录并告知客户端。Expire方法可以销毁指定Key对应的记录值，根据Key找到对应的Node，并调用缓存淘汰策略的DELETE方法删除。核心代码见附录1所示。

4.4.2 Cache Client

在HBase集群中部署缓存服务器，主要有两大方向实现，一种为客户端实现方式，另一种为服务器端实现方式。

客户端实现方式共有两种方式。

一种是采用修改HBase Client源码方式，在GET操作部分加入关键代码，在GET方法内部先进行缓存服务器的访问。另一种为封装HBase Client，自定义一个GET操作类，先访问缓存服务器，最后访问HBase原生GET API。

服务器端实现方式与客户端实现方式类似，共有两种方式。

一种是修改HBase Server端GET操作的源码，在GET方法内部加入缓存服务器访问。另一种是在服务器端添加缓存代理层，解析来自HBase客户端的GET请求。

客户端和服务器端实现方式中的第一种方式，均是采用直接修改HBase 原生GET操作的方式实现，从性能上讲，会比封装GET操作类好，但修改HBase源码会导致与当前版本HBase绑定，当版本更新时，需要重修修改源码，维护成本较大。服务器端第二种方式设计缓存代理层，可以解决对HBase版本的过分依赖关系，但实现一个服务器端的缓存代理层难度较大，需要对HBase底层通信机制较为熟悉。因此，本系统采用的方法为客户端的第二种实现方式，即在客户端封装一个GET操作类。既解决了版本依赖关系，同时开发难度也比服务器端更低，且扩展性较好。

按照客户端第二种实现方式，主要实现细节为Cache Client模块，对原生HBase API进行封装，同时，该模块也将成为Redis集群服务器的客户端入口。

Client客户端UML类图如图4.14所示，主要分为三部分Cache Client、Cache和Redis Cluster。在Java中操作Redis需要Jedis，目前Jedis更新至2.1.0版本。如果需要Redis连接池，则需要导入common-pool-1.5.5.jar。Jedis是Redis的Java客户端，Jedis提供了完整的Redis的Java API，可以完整的支持Redis客户端，具体API可参考官方资料16。

Cache Client为对外接口，可以通过Cache Client对整个缓存服务器进行配置，共有四种模式供开发者配置：

只使用HBase：在用户进行数据查询时，直接进入HBase进行Scan操作，并将数据直接返回给用户，不进行缓存操作。

只使用Redis集群缓存：在用户进行数据查询时，进入Redis集群服务器进行查询，若未命中数据，则进入HBase查询，这是常用的缓存处理方式。

只使用本地缓存：对于部分数据量小，但会被频繁访问的数据，或本身缓存数据量小，无需使用Redis集群服务器进行缓存处理的情况，Cache Client提供了一种本地缓存策略。在用户进行数据查询时，进入本地缓存进行查询，若未命中数据 ，则进入HBase查询。该策略适用于小数据量缓存。

同时使用Redis集群缓存和本地缓存：该策略设置二级缓存，当用户进行数据查询时，首先进入本地缓存，若未命中数据，则进入Redis集群服务器进行查询，若仍然未命中，则进入HBase进行Scan操作。该策略在支持大数据量缓存的情况下，兼顾了缓存中被频繁访问的数据，命中率和缓存检索效率最高。查询流程如图4.15所示。优先级顺序为Local Cache>Redis Cluster>HBase Scan。

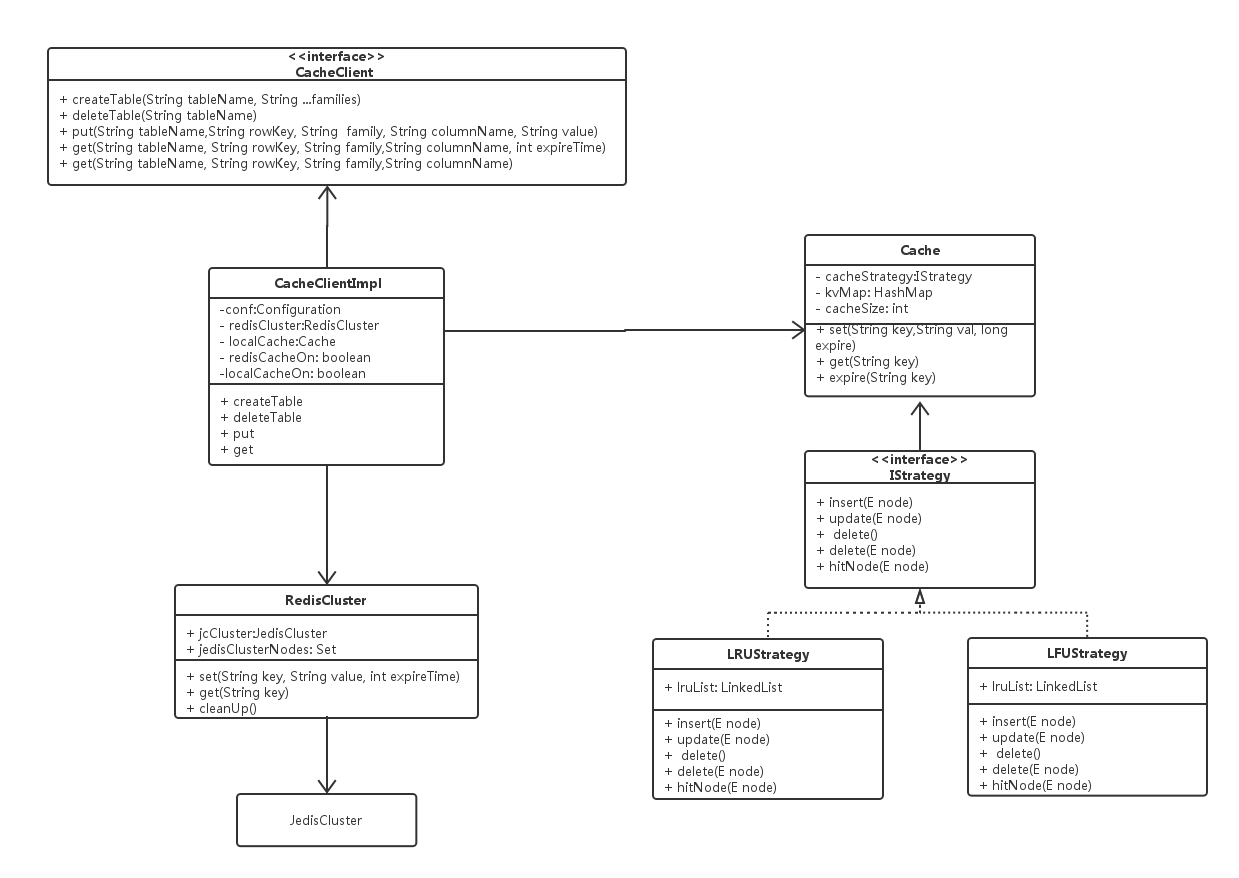


图4.14 Client客户端UML类图

对于不同的缓存策略的使用，要根据实际应用场景进行具体分析。当遇到需要大量缓存数据且数据访问概率较平均业务时，需要考虑Redis集群服务器，当遇到少量访问频繁的数据时，Local Cache则更为合适。



图4.15 缓存策略执行流程

4.5 集群优化配置

目前对于Hadoop+Zookeeper+HBase集群搭建方案较为成熟，本文将不再对其进行赘述，具体搭建方案可参考相关网络文献或Apache HBase相关书籍3。为了保障HBase大量写入时无障碍，提升读写性能，可以进行如下配置：

1. 禁用Major Compaction。

HBase默认设置为一天执行一次Major Compaction，在进行Major Compaction操作时，Region会因为合并StoreFile导致当前Region不可读，在数据读写高峰期时，Major Compaction会导致读写延迟，因此，需要将Major Compaction禁用，并设置在固定空闲时间进行Major Compaction即可。

1. 禁用Split

与Major Compaction相似，HBase通过Split可以将Region进行分裂操作，在HBase 0.94.0版本中引入了SplitPolicy，可以主动干预Split。由于在Split过程中无法进行读写操作，因此，建议禁用自动Split，使用SplitPolicy手动执行。

1. 增加BlockingStoreFiles值

在HBase进行写入Region操作时，会先检测当前HFile空间是否足够，若不够写入容量，则会先中止写入流程，等待其他线程Compact。因此，设置一个合理的BlockingStoreFiles可以保证数据的实时写入。

4.6 本章小结

本章在第三章理论基础上，给出了一种交通流数据实时存储与查询系统的具体实现。主要从数据写入模块和数据读取模块两个方面做了介绍。数据写入功能介绍了数据预处理层、缓冲层和写入层的实现方案。在SQL解析模块，本文介绍了Phoenix的配置方案。对于缓存服务器的实现，给出了一种常见的KeepAlived+TwemProxy+Redis Cluster配置方案。详细设计了一种基于本地缓存和Redis缓存服务器的Cache Client。最后，给出了HBase集群的优化配置相关建议。

第五章 实验设计与结果分析

在上一章给出了交通流数据实时存储与查询系统的具体实现，在本章中，将根据上一章提出了具体实现搭建测试环境并进行性能测试，主要测试包括数据写入测试、缓存命中率测试、缓存淘汰测试等。

5.1 实验环境搭建

5.1.1 硬件环境

本次实验测试选用5台配置相同的标准服务器，硬件配置情况如表5.1所示。

表5.1 配置清单表

|  |  |
| --- | --- |
| **属性** | **配置信息** |
| 中央处理器（CPU） | 4 Core Intel Xeon E3-1230 V2 3.3GHZ |
| 内部存储器（Memory） | 4GB DDR3 1600MHZ |
| 外部存储器（HDD） | 1TB 7200RPM 64MB SATA 6Gb/s |
| 网络适配器（Network） | Bandwidth 1Gbps |
| 操作系统（OS） | CentOS-6.5-i386 |

将所有实验服务器进行磁盘格式化，并重新安装操作系统，最大程度保障了各个服务器在实验阶段处于初始化一致状态。 各台独立服务器通过交换机使用网线连接，组成集群通路，并为各个服务器设置对应IP地址和网关，配置hosts文件。具体设置方案如表5.2所示。

表5.2 网络地址配置清单

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **主机名** | **IP地址** | **备注** |
| 1 | Master | 192.168.58.130 | HBase Master节点、TwemProxy、Redis Master节点 |
| 2 | Slave\_01 | 192.168.58.131 | HBase Slave节点、TwemProxy、Redis Slave节点 |
| 3 | Slave\_02 | 192.168.58.132 | HBase Slave节点、TwemProxy、Redis Slave节点 |
| 4 | Slave\_03 | 192.168.58.133 | HBase Slave节点、TwemProxy、Redis Slave节点 |
| 5 | Slave\_04 | 192.168.58.134 | HBase Slave节点、KeepAlived服务 |

注：网络掩码：255.255.255.0，网关地址：192.168.58.2

如表5.2所示，Hadoop集群服务器采用CentOS 6.5作为操作系统，其中一个为Master服务器，即NameNode，另外四个为Slave服务器，即DataNode，采用静态IP地址方式通信。Redis集群服务器采用一台Redis Master和三台Redis Slave。 Slave\_04服务器提供KeepAlived服务

5.1.2 软件环境

在硬件环境搭建完成之后，为不同服务器实例搭建软件环境，主要分为Java运行环境、Hadoop环境、HBase环境和Zookeeper环境，软件配置清单如表5.3所示。

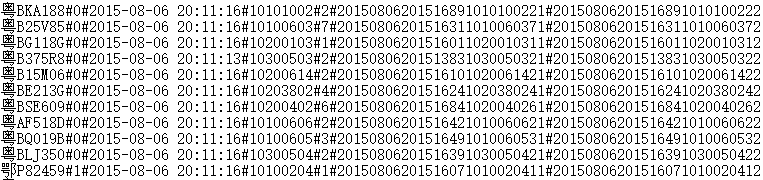
表5.3 软件配置清单

|  |  |
| --- | --- |
| **名称** | **备注** |
| JDK\_1.6\_Linux | 由于服务器实例采用32位Cent OS操作系，因此选用JDK1.6 |
| Hadoop\_2.2.0 | Hadoop安装程序 |
| hbase-0.98.24-hadoop2-bin | HBase安装程序，此处需要区分Hadoop1和Hadoop2版本 |
| Zookeeper\_3.4.6.tar | Zookeeper服务程序 |
| Redis\_2.4.10 | Redis安装程序 |
| KeepAlived\_1.2.8 | KeepAlived服务程序 |
| Autoconf\_2.69 | Shell脚本工具 |
| TwemProxy-Master | TwemProxy安装程序 |

5.1.3 实验数据准备

实验采用的数据集为国内真实交通流数据集，数据来源为国内某市2015年06月至2015年10月期间各路口路段监测设备采集数据、车辆全球定位系统数据等。数据规模为5亿条，数据集共100GB。为了使实验请求更真实的还原实际操作，实验中引入了一种预测系统行为和性能的自动化负载测试工具——LoadRunner，该工具通过模拟数以千万计的用户并发访问进行性能监测来发现和确认问题，LoadRunner详情参考文献[[[71]](#endnote-71)]。

部分数据格式如图5.1所示。各个字段所代表意义前文已经介绍过，此处不再赘述。

图5.1 部分实验数据

5.2 实验分析

本节针对第四章提出的交通流数据实时存储与查询系统的实现进行实验分析，主要包括数据写入性能对比实验、缓冲区阈值对性能影响对比实验、数据查询速度对比实验、替换对比实验和SQL解析验证。

5.2.1 数据写入性能对比实验

本节使用三组对比实验，分别比较传统关系型数据库MySQL、原生HBase集群和本文设计的交通流数据实时存储与查询系统在数据写入频率为2k、4k、6k…20k时的延迟时间。实验结果延迟时间越小的方案，数据写入性能越好。为了更真实的模拟不同数据源同时发出写入数据请求，使用LoadRunner配置数据请求频率，分别为2千条/秒、4千条/秒直至2万条/秒，每条记录大小约为150字节，在本次对比实验中，缓冲区队列阈值设置为6M。实验结果如图5.2所示。

图5.2 写入性能测试对比图

如图5.2所示，横坐标为经由LoadRunner发送的数据的频率，纵坐标表示数据库响应的延迟时间，所谓响应延迟时间，是指经过两小时持续的数据写入之后，剩余未写入的数据写入对应数据库中所需要的时间。由图中可知，对于传统关系型数据库的代表MySQL，响应延迟时间随着数据发送速率成倍增长。对于原生HBase集群数据库，在五千条每秒数据发送频率以下时效率较高，当每秒数据量超过五千条时，原生HBase性能下降较快。而对于本文提出的HBase集群方案，由于缓冲队列和并发写入的存在，响应延迟时间随着每秒发送速率变化的曲线较平缓，直到每秒数据发送速率达到2万条时，才出现了较高的响应延迟时间。实验数据证明了本文提出的数据写入方案的可行性。

5.2.2 缓冲队列写入阈值测试实验

数据写入缓冲区共有若干个缓冲队列，每个缓冲队列独立负责数据的写入，需要为其设置合适的数据写入阈值。若阈值设置太低，则队列写入I/O操作频繁，影响性能，从而导致写入延迟变大。若阈值设置太高，则数据留在缓冲区时间过长，写入延迟过大，系统实时性不能保证。本节实验主要验证不同的写入阈值对数据库写入延迟的影响。与上节相同，使用LoadRunner配置数据请求频率，分别为2千条/秒、4千条/秒直至2万条/秒，每条记录大小约为150字节，在本次对比实验中，设置不同的缓冲区容量为2M、4M、6M、8M和10M。实验结果如图5.3所示。

图5.3 缓冲队列写入阈值对性能影响对比图

如图5.3所示，当数据发送速率不断增加时，不同的缓冲队列写入阈值对性能影响较大。在数据发送速率低于一万条每秒时，不同的阈值对性能影响较小，当超过一万条每秒时，阈值为6M的曲线较缓，是理想的性能状态。低于6M的阈值由于I/O频繁导致性能降低，高于6M导致数据长时间滞留缓冲区，性能也会降低。

5.2.3 数据读取效率对比实验

本节使用两组对比实验，分别比较不使用任何缓存读取从原生HBase集群数据库和利用本文所使用的缓存集群服务器进行数据读取时效率差异，为了使实验结果更符合实际业务需求，设置缓存容量占总数据量的40%，并在实验前先随机从数据集中抽取部分数据组成待查询数据集（可能重复），约占总量的10%，分别记录下从第一条记录查询开始直至最后一条记录查询结果返回总共占用的时间，为了数据准确性，我们计算单条数据查询所需要的平均时间，实验结果如图5.4所示。

图5.4 数据读取效率对比实验

如图5.4所示，随着数据量的逐步增加，查询时间也越长。在数据量少于300万条的时候，直接进入HBase集群查询时间比使用缓存服务器更少，原因是在数据量较少时，查询数据集较少，此时大量时间被用来扫描HBase集群并构建缓存，而不使用缓存服务器则没有构建缓存环节，因此查询效率更高。随着数据量的变大，查询数据集的不断扩大，缓存服务器的优势也逐渐体现，当数据量达到5亿条时，使用缓存数据库的查询时间比不使用缓存数据库节约了近20%，查询效率提升了25%。数据证明，使用基于Redis缓存服务器的HBase集群在查询效率上更高。

5.2.4 缓存淘汰策略对比实验

通常衡量一个缓存淘汰策略的重要指标为缓存索引命中率，命中率越高，则说明缓存效率越高，命中率计算公式如公式5.1所示：

|  |  |  |
| --- | --- | --- |
|  |  | （5.1） |

本节实验分别对比不同缓存容量对缓存命中率的影响。具体来说，通过设置缓存容量占缓存空间的20%、30%直至80%，分别对比LRU算法和本文提出的基于热度的缓存淘汰算法的命中率，为了数据准确性，统一设置缓存总体容量占总数据量的40%。实验结果如图5.5所示。

图5.5 缓存容量与缓存命中率关系对比图

如图5.5所示，缓存命中率与缓存容量关系密切，随着缓存容量的不断变大，缓存命中率越高，因此，在部署交通流数据实时存储与查询系统时，建议提供足够大的缓存容量空间。同时，该实验也证明了基于热度缓存淘汰策略命中率比LRU算法更高，更适合本系统。

5.3 本章小结

在本章中，对交通流数据实时存储与查询系统进行了一系列实验。首先介绍了实验所使用的服务器的硬件配置、软件环境和相关数据来源。接着，进行了数据写入性能对比实验、缓冲队列写入阈值测试实验、数据读取效率对比实验和缓存淘汰策略对比实验。经由实验数据证明，本文所提到的基于HBase的交通流数据实时存储与查询系统在大部分情况下拥有较好的存储性能和读取性能。

第六章 总结与展望

6.1 工作总结

针对交通流数据中来源丰富、数据量大、读写频繁等特点，本文设计并实现了一套交通流数据实时存储与查询系统，该系统采用HBase作为持久化数据存储媒介，采用Redis作为缓存集群数据库。

在数据写入方面，设计了预处理模块、缓冲队列模块和写入层模块。数据写入时，通过对数据“清洗”，将多源数据进行分类、筛选、修复之后，通过缓冲队列并发写入HBase数据库中，防止大量数据涌入导致写入阻塞，系统性能降低。

在数据查询方面，非主键查询请求不再直接进入HBase进行Scan操作，在HBase集群之上搭建了多级索引缓存。本地缓存用于存储数据量小，但访问热度高的数据。Redis缓存服务器使用分布式集群，可以存储大量缓存数据。为了对缓存数据进行统一管理，提高命中率，本文提出了一种缓存淘汰策略，将热度值较低的数据记录淘汰出Redis缓存服务器。

最后，依据交通流数据实时存储与查询系统的理论，设计实验论证了本文提出的交通流数据实时存储与查询系统的可行性。

6.2 展望

虽然本文提出了一种基于HBase的交通流数据实时存储与查询系统的实现方案，并使用数据论证了可行性，但由于本人的研究水平和研究时间的有限，该工作依然还存在着下一步改进的问题：

1. 本文提出的SQL解析方案基于Phoenix解析器，Phoenix提供了丰富的SQL解析语法，对于交通流数据实时存储与查询系统来说，大部分解析语法是冗余功能，这些功能会导致解析速度降低。因此，后续工作中，可以考虑实现一套基于ANTLR的适用于交通流数据实时存储与查询系统的SQL解析模块。
2. 数据实时写入方面，由于数据写入时需要经过预处理、缓冲层和写入层，即使多个并发队列同时写入，但对于部分数据仍然存在延迟问题。所以如何在数据写入时减少延迟是需要进一步研究的问题。
3. 缓存设计方面，交通流数据实时存储与查询系统目前只支持Redis作为缓存集群数据库。随着技术日益成熟，涌现出越来越多的缓存数据库设计方案，在后续研究中，会开放支持其他缓存设计方案。

致 谢

时间如流水，不舍昼夜匆匆，转眼又到了江南草长莺飞的季节了。花谢花开，人来人往，三年时间转瞬即逝却又慢慢悠悠，在这种快与慢的矛盾中，我也终于走过了二十五个春秋，来到了花儿开满春天的路口。

论文从最初的构想，慢慢从脑海里走出轮廓，慢慢从轮廓中化出虚实，慢慢从虚实中生出眉目。语言在这一刻显得如此苍白无力，只有一步步走来，才能知道导师对于我的帮助，竟至于如此；总会想起那一遍遍的教诲和指导。就这么，在导师的指导、同门的关心和朋友的帮助里，写出了这不算漂亮，却有鼻子有眼的论文。

感谢时光，感谢每一个清晨食堂里飘满油条青菜香的时光，感谢每一个晚实验后都有月光一路相伴的时光。感谢每一个在图书馆抚摸书脊的静谧时光，感谢每一个穿梭于各个教室间聆听知识的时光。

感谢亲人，感谢年过半百苍苍白发、盛年不再佝偻身躯的爸妈，半辈子的辛劳和面朝黄土背朝天的勤勤恳恳，写出了满脸满身的印记和沧桑。

感谢我的小伙伴们，我们共同为青春奋斗，我们共同在黑夜祈求黎明，我们共同把酒言欢，我们共同相知相交，然而，前路漫漫，无论未来我们携手同行还是相忘江湖，我都祝福你们，我将永远珍藏我们之间的点滴回忆，因为这是我一生最宝贵的财富！

最要感谢的是我的导师李星毅副教授，研究从选题立题、实验设计到具体实验、结果分析直至文章撰写和论文的修改都凝结了导师的心血和智慧结晶。李老师是一位良师益友，他严谨的治学态度、渊博的知识、创新的思维和高尚的人格给我留下了深刻的印象，并将使我受益终身。

感谢江苏大学，感谢镇江这座城，把青春安放在这里让我觉得幸福。三年前，一个来自远方的小伙一脚踩进江南的烟雨朦胧，他曾经以为这是一次放逐，而今回望始知，它是一场成全。

最后，还要感谢一个人，张旺旺，感谢时光荏苒，三年如白驹过隙，教会我珍惜。

“逝者如斯夫，不舍昼夜”．成长亦复如是，不断的和自己告别，也不免有与你们的分离．但是，一路上有你们，真好！相会是缘，同行是乐，共事是福！

在校期间发表论文

1. 环境监测实验室信息管理系统.[软件著作权].2016SR178397 第一作者
2. 一种基于TwemProxy的HBase索引缓存方案[J].信息技术（已发表） 第一作者

参考文献

1. []李国杰, 程学旗. 大数据研究:未来科技及经济社会发展的重大战略领域——[大数据的研究现状与科学思考[J]. 中国科学院院刊, 2012, 27(6):5-15. [↑](#endnote-ref-1)
2. []王珊, 王会举, 覃雄派,等. 架构大数据:挑战、现状与展望[J]. 计算机学报, 2011, 34(10):1741-1752. [↑](#endnote-ref-2)
3. []George L. HBase : The Definitive Guide[J]. Andre, 2011, 12(1):1 - 4. [↑](#endnote-ref-3)
4. []Chang F, Dean J, Ghemawat S, et al. Bigtable:A Distributed Storage System for Structured Data[J]. Acm Transactions on Computer Systems, 2008, 26(2):1-26. [↑](#endnote-ref-4)
5. []史新宏, 蔡伯根, 穆建成. 智能交通系统的发展[J]. 北京交通大学学报, 2002, 26(1):29-34. [↑](#endnote-ref-5)
6. []蔡翠. 我国智慧交通发展的现状分析与建议[J]. 公路交通科技:应用技术版, 2013(6):230-233. [↑](#endnote-ref-6)
7. []MySQL Database[EB/OL]. https://www.mysql.com/ [↑](#endnote-ref-7)
8. []Oracle Database[EB/OL]. http://www.oracle.com/ [↑](#endnote-ref-8)
9. []DB2 Database[EB/OL]. http://www.ibm.com/analytics/us/en/technology/db2/ [↑](#endnote-ref-9)
10. []Altinel M, Luo Q, Krishnamurthy S, et al. DBCache:database caching for web application servers[C]// ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June. DBLP, 2002:612. [↑](#endnote-ref-10)
11. []Anton J, Jacobs L, Liu X, et al. Web caching for database applications with Oracle Web Cache[C]// ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June. DBLP, 2002:594-599. [↑](#endnote-ref-11)
12. []Big Data[EB/OL]. Big data. 2011. http://en.wikipedia.org/wiki/Big\_data. [↑](#endnote-ref-12)
13. []Chen J, Chen Y, Xiaoyong D U, et al. Big data challenge: a data management perspective[J]. Frontiers of Computer Science, 2013, 7(2):157-164. [↑](#endnote-ref-13)
14. []NoSQL Database[EB/OL]. http://nosql-database.org/. [↑](#endnote-ref-14)
15. []Decandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[J]. Acm Sigops Operating Systems Review, 2007, 41(6):205-220. [↑](#endnote-ref-15)
16. []Redis Database[EB/OL]. https://redis.io/. [↑](#endnote-ref-16)
17. []Lakshman A, Malik P. Cassandra:a decentralized structured storage system[J]. Acm Sigops Operating Systems Review, 2010, 44(2):35-40. [↑](#endnote-ref-17)
18. []Banker K. MongoDB in Action[M]. Manning Publications Co. 2011. [↑](#endnote-ref-18)
19. []Anderson J C, Lehnardt J, Slater N. CouchDB: The Definitive Guide[J]. Andre, 2010, 215(1):págs. 76-80. [↑](#endnote-ref-19)
20. []Vukotic A, Watt N, Abedrabbo T, et al. Neo4j in Action[C]// Manning Publications Co. 2014. [↑](#endnote-ref-20)
21. []Hindex[EB/OL]. :https://github.com/Huawei-Hadoop/hindex. [↑](#endnote-ref-21)
22. []HBase-Indexer[EB/OL]. https://github.com/NGDATA/hbase-indexer. [↑](#endnote-ref-22)
23. []Sfakianakis G, Patlakas I, Ntarmos N, et al. Interval indexing and querying on key-value cloud stores[C]//Data Engineering (ICDE), 2013 IEEE 29th International Conference on. IEEE, 2013: 805-816. [↑](#endnote-ref-23)
24. []Grainger T, Potter T. Solr in Action[M]. Manning Publications Co. 2014. [↑](#endnote-ref-24)
25. []Bentley J L. Solutions to Klee’s rectangle problems[R]. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, 1977. [↑](#endnote-ref-25)
26. []Borthakur D. HDFS architecture guide[J]. Hadoop Apache Project, 2008, 53. [↑](#endnote-ref-26)
27. []He B, Fang W, Luo Q, et al. Mars: a MapReduce framework on graphics processors[C]//Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008: 260-269. [↑](#endnote-ref-27)
28. []Borthakur D. The hadoop distributed file system: Architecture and design[J]. Hadoop Project Website, 2007, 11(2007): 21. [↑](#endnote-ref-28)
29. []陆嘉恒. Hadoop 实战[M]. 机械工业出版社, 2012. [↑](#endnote-ref-29)
30. []Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113. [↑](#endnote-ref-30)
31. []Mmel R. Google's MapReduce programming model — Revisited[J]. Science of Computer Programming, 2008, 70(1):1-30. [↑](#endnote-ref-31)
32. []Junqueira F, Reed B. ZooKeeper: Distributed Process Coordination[M]. O'Reilly Media, Inc. 2013. [↑](#endnote-ref-32)
33. []O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4):351-385. [↑](#endnote-ref-33)
34. []Howard J H, Stein C A. Data storage system and method employing a write-ahead hash log: US, US6629198[P]. 2003. [↑](#endnote-ref-34)
35. []叶小飞. 浅谈HBASE数据结构设计[J]. 信息通信, 2016(11). [↑](#endnote-ref-35)
36. []杨艳, 李炜, 王纯. 内存数据库在高速缓存方面的应用[J]. 现代电信科技, 2011, 41(12):59-64. [↑](#endnote-ref-36)
37. []Carlson J L. Redis in Action[J]. Media.johnwiley.com.au, 2013. [↑](#endnote-ref-37)
38. []李子骅. Redis入门指南.第2版[M]. 人民邮电出版社, 2015. [↑](#endnote-ref-38)
39. []数据字典[EB/OL]. http://baike.baidu.com/item/数据字典. [↑](#endnote-ref-39)
40. []Ren Z, Cai X, Bai L, et al. Double-linked List Simulation of Insertion Sort Algorithm[J]. Computer Programming Skills & Maintenance, 2010. [↑](#endnote-ref-40)
41. []符璨. 一种基于Redis的轻量级消息中间件的设计与实现[D]. 北京大学, 2013. [↑](#endnote-ref-41)
42. []潘国浩. 一种新的内存数据库快速日志恢复技术[J]. 计算机与信息技术, 2007(5):83-85. [↑](#endnote-ref-42)
43. []张文帅. Redis基于RDB+AOF的数据恢复策略研究[J]. 电脑知识与技术, 2016, 12(14):7-10. [↑](#endnote-ref-43)
44. []智能交通系统[EB/OL]. http://baike.baidu.com/item/智能交通系统. [↑](#endnote-ref-44)
45. []王晓原, 张敬磊, 吴芳. 交通流数据清洗规则研究[J]. 计算机工程, 2011, 37(20):191-193. [↑](#endnote-ref-45)
46. []韩卫国, 王劲峰, 胡建军. 交通流量数据缺失值的插补方法[J]. 交通信息与安全, 2005, 23(1):39-42. [↑](#endnote-ref-46)
47. []Gao J. A Sensory Data Integrating Bus under the Intelligent Transportation Environment[M]. 2013. [↑](#endnote-ref-47)
48. []王金全. 一种多租户数据管理方法及其在智能交通中的应用[D]. 北方工业大学, 2014. [↑](#endnote-ref-48)
49. []陆婷, 房俊, 乔彦克. 基于HBase的交通流数据实时存储系统[J]. 计算机应用, 2015, 35(1):103-107. [↑](#endnote-ref-49)
50. []Zdravevski E, Lameski P, Kulakov A. Row Key Designs of NoSQL Database Tables and Their Impact on Write Performance[C]// Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2016:10-17. [↑](#endnote-ref-50)
51. []柳皓亮, 王丽, 周阳辰. Redis集群性能测试分析[J]. 微型机与应用, 2016, 35(10):70-71. [↑](#endnote-ref-51)
52. []王心妍, 毛莉君. 基于Twemproxy的Redis集群解决方案的设计与实现[J]. 电子测试, 2016(6). [↑](#endnote-ref-52)
53. []Wang X, Mao L, Xian Peihua University. Design and implementation of Redis cluster solution based on Twemproxy[J]. Electronic Test, 2016. [↑](#endnote-ref-53)
54. []柳皓亮, 王丽, 周阳辰. Redis集群性能测试分析[J]. 微型机与应用, 2016, 35(10):70-71. [↑](#endnote-ref-54)
55. []刘敏娜, 张继涛. 基于LVS+KEEPALIVED的高可用负载均衡研究与应用[J]. 自动化技术与应用, 2014, 33(11):22-27. [↑](#endnote-ref-55)
56. []钱景辉, 廖锂, QIANJing-hui,等. 基于Keepalived的动态浮动IP集群实现[J]. 化工自动化及仪表, 2012, 39(7):926-928. [↑](#endnote-ref-56)
57. []杨波, 武波. 虚拟路由冗余协议及应用[J]. 通信世界, 2002(19):42-42. [↑](#endnote-ref-57)
58. []瞿龙俊，李星毅. 一种基于Redis TwemProxy索引缓存的HBase查询方案[J]. 信息技术，2017(09) [↑](#endnote-ref-58)
59. []葛微, 罗圣美, 周文辉,等. HiBase:一种基于分层式索引的高效HBase查询技术与系统[C]// ccf大数据学术会议. 2014. [↑](#endnote-ref-59)
60. []先入先出队列[EB/OL]. http://baike.baidu.com/item/先入先出队列. [↑](#endnote-ref-60)
61. []张震波, 杨鹤标, 马振华. 基于LRU算法的Web系统缓存机制[J]. 计算机工程, 2006, 32(19):68-70. [↑](#endnote-ref-61)
62. []Karakostas G, Serpanos D N. Practical LFU Implementation for Web Caching[J]. 2000. [↑](#endnote-ref-62)
63. []Devine R. Design and implementation of DDH: A distributed dynamic hashing algorithm[M]// Foundations of Data Organization and Algorithms. Springer Berlin Heidelberg, 1993:101-114. [↑](#endnote-ref-63)
64. []杨彧剑, 林波. 分布式存储系统中一致性哈希算法的研究[J]. 电脑知识与技术, 2011, 07(22):5295-5296. [↑](#endnote-ref-64)
65. []汪然. 基于ActiveMQ的消息中间件的设计与实现[D]. 西安电子科技大学, 2013. [↑](#endnote-ref-65)
66. []Videla A, Videla A. RabbitMQ in Action[M]. Manning Publications, 2012. [↑](#endnote-ref-66)
67. []Melnik S, Gubarev A, Long J J, et al. Dremel: Interactive Analysis of Web-Scale Datasets[J]. Communications of the Acm, 2010, 3(12):114-123. [↑](#endnote-ref-67)
68. []Vashishtha H, Stroulia E. Enhancing Query Support in HBase via an Extended Coprocessors Framework[M]// Towards a Service-Based Internet. DBLP, 2011:75-87. [↑](#endnote-ref-68)
69. []Phoenix[EB/OL]. http://phoenix.apache.org/. [↑](#endnote-ref-69)
70. []Clustering redis to maximize uptime and scale[EB/OL]. https://blog.recurly.com/2014/05/clustering-redis-maximize-uptime-scale. [↑](#endnote-ref-70)
71. []戴晓婧, 张宁. 基于LoadRunner的数据分析平台的性能测试及优化[J]. 计算机技术与发展, 2013, 23(7):202-206. [↑](#endnote-ref-71)