

HBase 内存索引系统的研究与实现



重庆大学硕士学位论文 (专业学位)

学生姓名：卢文博

指导教师：刘卫宁 教授

学位类别：工程硕士（计算机技术领域）

重庆大学计算机学院

二〇一六年四月

Research and Implementation of HBase Memory Index System



A Thesis Submitted to Chongqing University
in Partial Fulfillment of the Requirement for the
Professional Degree

**By
Lu Wenbo**

**Supervised by Professor Weining Liu
Specialty: ME(Computer Technology Field)**

College of Computer Science of
Chongqing University , Chongqing, China

April, 2016

摘 要

随着计算机技术和数据库技术的快速发展, 人类需要储存的数据量极大的增长, 传统的储存、处理数据的理念已经不能适用于海量数据的环境。为了满足当前数据存储和处理的需求, 现在的数据系统是建立在分布式系统之上的。HBase 是目前最受欢迎开源分布式数据库软件之一。HBase 的设计主要目的是稳定的存储海量的数据, 在其它方面 HBase 的性能并没有我们期望的高。HBase 读取时对数据的行键 (rowkey) 有着很大的依赖性, 这点尤其限制 HBase 在复杂条件下的查询性能。

在传统的数据库中索引技术的使用可以极大的提高数据库的查询效率, 以此为鉴本文尝试建立 HBase 非 rowkey 列的索引来提高 HBase 在复杂条件下的查询性能。本文中索引树采用了重庆大学提出的 HT 树索引, 为了优化索引树的空间利用率对 HT 树的插入和删除算法进行了一些优化。本文中将索引树存储在 Spark 分布式内存计算系统中, Spark 是一个效率极高的分布式内存计算软件, 将索引树存储在 Spark 之上能快速提高索引的处理效率。在索引系统的实现时采用了二级索引的设计架构, 在这种设计下每个查询都是由两部分组成的。查询时我们可以先在索引系统中获取到 rowkey, 拿到 rowkey 之后再从 HBase 读取数据。这种分段查询的思想虽然在查询条件里有 rowkey 的情况下性能比 HBase 稍低, 但极大的提高了查询条件里没有 rowkey 时 HBase 的查询效率, 使得 HBase 在复杂查询条件下的适用性提高。

本文同时也实现了一个较为精简的索引系统。索引系统由索引中间件和应用程序接口 (API) 两部分组成。索引中间件是系统的核心所在, 实现了数据插入时索引树的建立, 数据查询时索引树的查询, 数据删除时索引树的修改, 同时也实现了索引树、HBase、Spark 以及应用程序接口之间的数据交互。应用程序接口主要有 Java 语言的访问接口和 Web Service 的访问接口两种。

关键词: HBase、HT 树、内存索引、Spark

ABSTRACT

With the rapid development of computer technology and database technology, the human need to store the amount of data that a great deal of growth, the concept of traditional storage and processing data is applicable is not measured data environment. In order to meet the needs of the current data storage and processing, the data system is based on distributed system. HBase is one of the most popular open source software distributed database. HBase purpose of design is stable to store vast amounts of data, HBase performance in other ways is not our expectations. HBase reads the data line (rowkey) have great dependence, this is particularly restricted HBase under the condition of complex query performance.

In the use of traditional database indexing technology can greatly improve the query efficiency of database, in order to guide this paper tries to establish a HBase non rowkey column index to improve HBase under the condition of complex query performance. In this article the index tree Luo Jun adopted in chongqing university professor and learn Chinese zodiac fuping HT tree index is put forward, in order to optimize the index tree of space utilization of HT tree insertion and deletion algorithm optimization. In this article, I will the index tree is stored in the Spark distributed memory computing system, the Spark is a high efficiency calculation software of distributed memory to store the index tree in Spark can quickly improve the processing efficiency of the index. During the implementation of the index system used the secondary indexes design architecture, every query in this design is composed of two parts. Query when we can get to the first in the index system rowkey, get to read the data from HBase after rowkey. The thought of the segmented query while in the query conditions have rowkey performance was slightly lower than HBase, but greatly improve the query conditions when there are no rowkey HBase query efficiency, makes the HBase under the condition of complex queries the applicability of the improved.

This article also implements a relatively compact index system. Index system by the index middleware and application program interface (API) of two parts. Index middleware is the core of the system, realized the data into the establishment of the index tree, index tree of data query is a query, modify data delete index tree, but also realized the index tree, HBase, Spark and data interaction between the application

program interface. Main application program interface with Java language access interface and two kinds of Web Service access interface.

Keywords: HBase、HT tree、Memory Index、Spark

目 录

中文摘要.....	I
英文摘要.....	II
1 绪 论.....	1
1.1 研究背景.....	1
1.2 研究现状.....	2
1.3 主要研究内容.....	3
1.4 论文内容结构.....	4
1.5 本章小结.....	5
2 相关技术介绍.....	6
2.1 基础技术介绍.....	6
2.1.1 Hadoop 简介.....	6
2.1.2 HBase 简介.....	6
2.1.3 Spark 简介.....	8
2.2 基础索引算法简介.....	10
2.2.1 Hash 索引.....	10
2.2.2 B 树索引.....	11
2.2.3 B+树索引.....	11
2.2.4 HT 树索引.....	12
2.3 HBase 查询优化方案.....	13
2.3.1 HBase rowkey 设计原则.....	13
2.3.2 二级索引架构.....	14
2.3.3 华为 HBase 索引方案.....	15
2.4 本章小结.....	16
3 索引系统的设计.....	18
3.1 索引系统方案.....	18
3.2 索引算法的选择.....	19
3.3 HT 树的操作.....	20
3.4 系统设计.....	22
3.4.1 系统架构图.....	23
3.4.2 主要操作流程.....	23
3.4.3 接口的架构图.....	25

3.5 本章总结.....	26
4 索引系统的实现.....	27
4.1 系统环境的搭建.....	27
4.1.1 软件版本的选择.....	27
4.1.2 Linux 基本系统安装.....	28
4.1.3 Hadoop 的安装.....	28
4.1.4 HBase 的安装.....	29
4.1.5 Spark 的安装.....	29
4.2 中间件的设计.....	30
4.3 中间件的实现.....	31
4.3.1 HBase 操作组件.....	31
4.3.2 Spark 操作组件.....	32
4.3.3 索引树核心组件.....	32
4.3.4 组件之间的耦合.....	34
4.4 API 设计.....	36
4.4.1 索引操作 API.....	36
4.4.2 数据操作 API.....	37
4.5 基于 Thrift 的 API 的实现.....	37
4.5.1 Thrift 简介.....	37
4.5.2 Java API 的实现.....	38
4.6 Web Service API 的实现.....	41
4.6.1 RESTful 简介.....	41
4.6.2 RESTful Web Service 的实现.....	41
4.7 本章总结.....	44
5 索引系统测试.....	45
5.1 总体设计.....	45
5.2 实验环境.....	46
5.3 数据入库测试.....	47
5.4 数据读取测试.....	49
5.4.1 查询条件 A 和 B 的查询时间对比.....	50
5.4.2 查询条件 A 和 C 的查询时间对比.....	51
5.4.3 查询条件 C 和 D 的查询时间对比.....	52
5.4.4 查询条件 B 和 D 的查询时间比较.....	52
5.4.5 查询条件 ABCD 的综合比较.....	53

5.5 本章小结.....	54
6 总结与展望.....	55
6.1 论文总结.....	55
6.2 今后展望.....	56
致 谢.....	57
参考文献.....	58

1 绪 论

1.1 研究背景

随着大数据技术在各个行业领域的广泛应用,越来越多的人认识到了数据的宝贵性,在这一认识的前提下日益积累起来了海量的数据。公开数据显示,Google 每天会产生 30 亿条的搜索记录,Facebook 每天会产生 30 亿的消息记录, Twitter 的用户每天要会发送出 4 亿条的推文^[1]。百度 2013 年拥有数据量已接近 EB 级别、阿里、腾讯申明其存储的数据总量都达到了百 PB 以上。此外,电信、医疗、金融、公共安全、交通、气象等各个方面保存的数据量也都达到数十甚至上百 PB 级别。据统计全世界的信息量大概以少于两年翻一倍的速度增长,数据的增长速度已经超过摩尔定律^[2,3]。由此估计 2013 年到 2020 年数据量将增长 10 倍,从 4ZB 到 40ZB。2013 年中国产生的数据总量超过 0.5ZB 占全球数据比例的 13%,预计到 2020 年中国产生的数据总量将超过 8ZB,占全球数据比例的 21%。从这些数据可以看出在数据的存储和处理技术方面,现有技术已经面临着比较大的挑战。

虽然传统的关系型数据库能够提供非常成熟稳定的数据存储、索引以及查询处理等功能,然而在不断增长的海量数据的情况下,关系型数据库面临着很大的挑战。主要体现在扩展性以及复杂查询情况下的查询效率等方面并不是很理想,无法实现高效灵活的动态扩展,及时复杂查询条件下的快速查询。虽然一些数据库公司提供了针对关系型数据库在分布式条件下的改进方案。但其由于其软件和服务价格的昂贵,以及设计上的复杂等因素,在部署和管理的代价是比较大的。在新的背景下传统的数据处理和存储的方法以及工具已不能满足当今时代对数据的存储和处理的需求。在这种条件下我们急需探索新条件下对数据的存储和处理的方式。

在面对复杂结构的海量数据的智能检索的需求下,人们急需寻找一种新的数据库代替传统数据库来存储、管理和处理数据^[4]。Google 在 2003 年的 SOSP (Symposium on Operating Systems Principles) 大会上发表了有关 GFS (Google File System^[5], Google 文件系统) 分布式存储系统的论文,在 2004 年的 OSDI (Operating Systems Design and Implementation) 大会上发表了有关 MapReduce^[6] 分布式处理技术的论文,在 2006 年的 OSDI 大会上发表了关于 BigTable^[7] 分布式数据库的论文。Google 这 3 篇具有深远意义的重量级论文的发表,不仅使大家了解 Google 搜索引擎背后强大的技术支撑,并且使得云计算成了解决海量数据条件下数据复杂的存储、管理和分析等问题的重要工具。并得到行业内众多大公司的广泛应用和深入

研究, 云计算已经成为大数据处理的一种基本解决方案。从而出现了很多优秀的开源的分布式数据存储和管理软件, 如 Hadoop、cassandra、HBase 和 ZooKeeper 等。在众多优秀的分布式数据存储和管理系统中, HBase 以其 key-value 的存储模式、海量数据条件下的高可靠性稳定性等特点脱颖而出, 占有了很大的一块市场。

HBase 是按照行健的顺序对数据进行逻辑组织和物理存储的, 然后在行健上建立类似于 B+树的索引结构, 所以 HBase 可以在行健上能够提供非常快速高效的查询。然而在实际应用中还有很多针对非行健的查询, 比如在个人医疗信息里每个人都会有非常多的生理指标, 每做一个检查都会有非常多的检查项, 我们会针对个人当前状况查看不同的信息; 比如在交通信息里会涉及到时间、地名、行驶方向甚至天气等很多种要素, 在大多数时候我们需要的不是一个单一的查询, 而是各种条件下的复杂的查询; 比如在气象数据里, 一条气象记录会涉及到经度、纬度、时间、海拔、气压、风速、风向、天气类型、可见度、降雨量等要素, 这种情况下我们需要的查询就更为复杂。由于 HBase 只有基于行健的索引, 在复杂条件下的查询只能通过全表扫描来实现, 这个缺点导致 HBase 难以满足复杂数据集在复杂查询条件下实施快速的查询。虽然我们可以利用 MapReduce 技术来实现数据访问的并行化, 在一定程度上提高查询速度。但是当数据量非常大的时候, MapReduce 并行查询所需的时间仍然比较长, 无法满足期望快速得到结果的应用的需求。因此, 在 HBase 基础上提供面向非主键的快速查询能力, 是目前 HBase 环境下急需研究和解决的一个重要问题。在本文中我们会继续分析这一问题, 并给出一个解决方案。

1.2 研究现状

目前国内外针对大数据环境下新的存储工具的索引优化做了很多研究, 这些研究总的来说可以分为索引系统架构的研究和索引算法的研究两个方向。这些研究相互之间是有紧密的依赖关系的, 索引架构的研究和索引算法的研究是相辅相成的, 一个优秀的索引系统必然在架构和算法两方面都是有独到之处的。下面我们简单介绍这两方面的研究成果。

索引架构的研究最初是基于外部软件的索引设计, 主要以 Lucene^[8]和 Solr^[9,10]这两个全文索引框架建立外部索引缓存。Lucene 是一个全文检索引擎的架构工具, 提供了完整的查询引擎和索引引擎以及部分文本分析引擎^[11]。Solr 是一个企业级搜索应用服务器, 对外提供类似于 Web Service 的访问的接口^[12]。用户可以通过 HTTP 请求向 Solr 提交一个具有一定格式的 xml 文件生成索引, 也可以通过 HTTP 提出查找请求, 并得到固定格式的返回结果。在这种架构中 Lucene 提供检索工具, 它包含了读写索引工具、索引结构、相关性工具、排序等功能, Solr 提供

索引的核心存储引擎服务，并可以配置缓存系统来提高查询效率。这种架构的设计思想类似于搜索引擎的设计思想，将 HBase 数据作为检索数据源，Solr 作为索引服务器解析 xml 配置文件中的索引项，Lucene 作为检索服务器来检索数据。这种架构的缺点在于 Solr 和 Lucene 的配置和使用是比较复杂的，且这两个组件需要较高的硬件支持。

外部索引的研究在一定程度上借鉴了传统数据库的索引设计的方法，比如二级索引的设计架构。二级索引架构主在 HBase 中的利用主要有以下几个方案：开源的 ITHBase 和 IHBBase；中国科学院提出的 CCIndex^[23]（complemental clustering index，互补聚簇式索引）；异步视图方案 Asynchronous views^[24]，以及华为公司基于 Coprocessor 技术的 Hindex 索引和中兴的 HiBase^[13]系统。二级索引方案设计的核心在于在主键之外提供其它列的索引，即我们在数据库之外维护额外的索引结构。查询数据库时根据查询条件先查询索引，得到索引查询的结果后再对数据库进行查询。

索引架构还有一些其它的方案，比如日本 NEC 公司的服务研究实验室和加利福尼亚大学的研究人员在 MD-HBase^[14,15]中提出了一种基于线性化技术的索引方案，其特点是不需要维护额外的外部索引就可以实现对内容的快速索引，但这种方案的实现比较复杂，且其存在一些难以解决的问题。

索引算法的研究有很多方向，在本文中我们主要关注内存索引算法的研究。针对内存索引算法的研究最早提出高性能内存索引数的是 T 树。后面随着硬件技术的快速发展，CPU 的频率不断提高而内存的访问速度却进步不是很多，导致了 CPU 和内存访问速度的失衡，为了弥补这种不足 CPU 缓存的容量也不不断的增大。在这种情况下研究发现高速缓存对数据库性能有着非常显著的影响，T 树在这种环境下由于较高的 TBL 失配率而降低了其效率。为了适应这种硬件的改变而提出了很多对 CPU 和高速缓存敏感的索引算法，主要有 CSS 树、CSB 树、CSB+树、pB+树、HT 树等。

1.3 主要研究内容

通过对研究背景和研究现状的描述，本文尝试性的提出了在 HBase 之上提供一个高效的分布式内存索引系统这个研究目的。为了更好的实现我们的研究目标，我们提出以下几个研究内容，也是本文的主要研究思路。

① 提出我们的设计目标。

在本文中分析了开源 HBase 系统在特定条件下的缺点，主要问题是 HBase 在查询条件里没有 rowkey 时的查询效率是极低的，在随机查询时的查询效率也是非常低的。而大多情况下的查询条件都是比较复杂的，不一定在每次查询里都能提

供 rowkey。针对上述缺点，本文提出了设计一个高效的 HBase 索引系统的目标。索引系统中可以给我们关注的列保持一个列值和 rowkey 的对应关系，在每次查询的时候都可以提供一个有效的 rowkey 给 HBase，从而实现 HBase 的快速查询。

② 研究相关基础技术

主要研究索引系统的算法和架构，为索引系统实现选择合适的索引算法和系统架构。算法主要包括一些基础的索引算法和 HT 树索引算法，架构主要是二级索引的架构。因为本文的索引系统是对于 HBase 的一种扩展，因此在写作本文的过程中研究了 HBase 的基本原理。HBase 的基本原理主要有 HBase 的数据模型和 HBase 检索原理，HBase 的数据模型又分为 HBase 的逻辑模型和 HBase 的物理模型。

③ 针对现有问题提出本文的解决方案

本文的设计目标是提供一个高效的 HBase 索引系统，针对这一目标我们提出了这样的解决方案：基于 Spark 的 HBase 内存二级索引方案。索引系统构建在 Spark 分布式内存计算之上可以使得索引系统的运行效率更好，索引算法选择 HT 树索引算法，索引架构采用二级索引架构。

④ 实现索引系统

索引系统的实现中包含了索引树的实现以及索引树、HBase、Spark 和索引系统访问接口的数据交互处理等。在索引树的查询时使用了 Spark streaming 流处理的方式实现任务在 Spark 中的执行。

⑤ 为索引系统提供 API

为了方便程序对索引系统的访问，我们提供了基于 Thrift 的 Java 应用程序接口和 Web Service 接口。其中 Thrift 的接口还可以很方便的扩展到其它的语言，降低了系统的跨语言跨平台的开发难度。Web Service 接口可以使得索引系统的适用范围更为宽广，只要支持网络访问就可以访问索引系统。

1.4 论文内容结构

本文的内容组织如下：

第一章为绪论。主要介绍了本文的研究背景及研究意义，分析了现有的 HBase 索引技术的研究现状，指出了论文的主要研究内容，对论文的行文结构做出安排，主要对后面的研究工作起一定的指导作用。

第二章是相关技术介绍。首先简要介绍了论文所需要的一些技术工具，而后介绍了一些基础索引算法以及目前关于大数据条件下一些新的索引架构技术及一些常见的解决方案。

第三章是索引系统的算法选择和架构设计。首先对第二章描述的一些索引算

法进行了概要性的对比,在此基础上选择 HT 树算法作为索引系统实现的核心算法,并对 HT 树的常用操作做了详细的介绍。最后描述了论文中索引系统的基本设计,对第四章索引系统的实现打下基础。

第四章时索引系统的实现。主要是开发相关的内容,首先是开发环境的搭建,其次是中间件的详细设计,而后是中间件的实现以及基于 Thrift 和 Web Service 的 API 的实现。

第五章是索引系统的测试。对索引系统的效率和 API 是否可以正常使用做简单的测试,并对测试结果进行分析,以分析结果论文索引系统设计的可行性以及其缺点。

第六章总结和展望。对本文的研究成果进行总结,提出论文的不足点,然后阐述了一些具有可行性的改进的想法。

1.5 本章小结

本章首先阐述了论文的研究背景及意义,分析了 HBase 索引系统的研究现状,然后指出了论文的主要研究的几个内容,最后对论文的行文结构做出了大致的安排,对后面的研究工作起到了引导作用。

2 相关技术介绍

2.1 基础技术介绍

2.1.1 Hadoop 简介

Hadoop 是一个由 Apache 软件基金会所开发的开源的分布式系统基础架构。Hadoop 在底层实现了分布式系统的功能,面向用户的是简单易用的使用接口,用户可以直接在这些接口之上开发分布式程序。降低了分布式计算的开发难度,使得大多数人可以充分利用集群的威力进行高速运算和存储。Hadoop 实现了一个类似于 Google 的 GFS 的分布式文件系统,简称 HDFS。HDFS 在设计最初就确立了单节点的存储是不可靠的这一理念,在设计的过程中采用冗余备份的方式实现了数据存储高容错性的特点,基于其存储的冗余备份产生的高可靠性,Hadoop 分布式环境可以部署在低廉的硬件上,这无疑降低了集群的部署成本;而且它提供高吞吐量来访问应用程序的数据,适合那些有着超大数据集的应用程序^[16]。Hadoop 的框架最核心的设计就是:HDFS 和 MapReduce。HDFS 是一个分布式的文件系统,提供了大数据的存储环境;MapReduce 是一个分布式的计算框架,提供了大数据的计算环境。

Hadoop 主要有以下几个优点:

- ① 高可靠性。Hadoop 中是按位存储数据的,同时其通过分布式平台按照位处理数据的可靠性是非常高的。
- ② 高扩展性。Hadoop 在运算时将数据和计算动态的分布在分布式集群的各个节点中同时也可以灵活的调整集群的大小,扩展性很高。
- ③ 高效性。Hadoop 中的数据是动态的存储于整个集群中的,可以根据节点的运行状态调整数据的存储和运算,其存储效率和运算效率都是非常高的。
- ④ 高容错性。Hadoop 容错机制会让每份数据在集群里保存为多个副本,不怕单一节点上数据的丢失。同时在计算过程中 Hadoop 会动态的处理失败的任务,将其换一个节点之后重新运行,最终保证每个任务都完成。
- ⑤ 低成本。于其它的解决方案相比,hadoop 是开源的,且其可以运行在廉价的主机之上,项目的成本因此会大大降低。

2.1.2 HBase 简介

① HBase 概述

HBase 是一个面向列的分布式的开源数据库,HBase 的设计理念来源于 Google 的论文“Bigtable: 一个结构化数据的分布式存储系统”,可以看做是 BigTable 在

Hadoop 平台的开源实现。HBase 不同于传统的关系数据库，它是一个适合于非结构化数据存储的分布式的数据库。HBase 的另一个特点是 HBase 基于列的而不是基于行的数据库。HBase 是运行在 Hadoop 上的非结构化数据库，其能够利用 HDFS 的分布式存储系统，和 Hadoop 的 MapReduce 分布式计算模式。这意味着 HBase 可以在一个庞大的计算集群里存储和处理一个具有上百万列和几乎没有上限行的特别大的表的数据。除了 Hadoop 带来的优势，HBase 本身也是一个十分强大的分布式数据库，它能够融合 key-value 存储模式带来性能较佳的实时查询的能力，以及通过 MapReduce 进行离线数据处理或者数据批处理的能力。简而言之，HBase 非常适合做海量数据的存储和处理，可以在较低的成本开销下获得较佳的性能和处理结果。

② HBase 数据模型

HBase 的数据模型和传统的关系型数据库的数据模型有些不同。传统的关系型数据库围绕表、列和数据类型使用严格且复杂的规则来约束数据之间的关系。遵守这些严格且复杂的规则的数据称为结构化数据。HBase 中存储的数据并没有严格复杂的数据关系。HBase 不提倡在表与表之间建立复杂的关联关系，而是将所有数据都放在一张大表中，表中数据记录的列、行等均是不确定的。这种没有复杂关联关系，且形态不确定的数据就是半结构化数据或者非结构化数据。

数据的逻辑模型会影响着数据系统物理模型的设计。关系型数据库存储在表中的记录一般都是结构化的数据，因此关系型数据库的物理模型也是适应结构化数据在内存和硬盘里存储结构的。HBase 因其存储着非结构化数据的特点而设计了其独特的物理模型。同时，随着存储硬件的发展，为了更好的适应硬件的结构我们需要提供更合适的物理模型，而为了便于实现物理模型我们也需要对逻辑模型做一些修改。因这种影响是双向的，所以合理的优化数据库系统必须深入理解数据的逻辑模型和物理模型。

除了存储非结构化数据这一特点外，HBase 还具有很强的可扩展性。在非结构化逻辑模型里数据之间没有复杂的关系去约束，这一优点非常便于数据在存储硬件里的分散存储，这一点也影响了 HBase 逻辑模型的建立。此外，这种物理模型设计使得 HBase 无法具有传统的关系型数据库的一些优点。比如说，HBase 不能实施数据之间的关系约束，不能支持多行的事物。这种关系影响了下面两个主题。

1) 逻辑模型：有序映射的映射集合

HBase 的逻辑数据模型有很多种描述方法。本文中我们采用有序映射的映射这一描述方法。有序映射的映射是一种映射的集合，可以把 HBase 看做有序映射的无限的、实体化的、嵌套的版本。HBase 中使用坐标系统来唯一标识其存储的数据，坐标值如下构成：[行健，列族，列限定符，时间版本]。

2) 物理模型：面向列族

和关系型数据库一样，HBase 中的数据表也是由行和列组成。HBase 中列是按照列族分组的。这种分组是 HBase 数据逻辑模型决定的。列族也表现在物理模型中，HBase 数据表的每个列独立存储在一个 HFile 文件中，随着数据量的增大一个 HFile 文件会分裂为多个 HFile 文件。可以这么认为，一个列族可以有多个连续存储的 HFile 文件，但每个 HFile 文件只属于一个列。而每个列组都是由多个列构成的，所以每个列族在硬盘上有自己的 HFile 集合。HFile 文件在物理存储上的隔离使得 HBase 可以很方便的在列族底层 HFile 层面上进行数据的管理。

HBase 的记录按照键值对存储在 HFile 里。HFile 只是二进制文件，是不可以直接读的。在 HFile 里这一行可以使用多条记录。每个列限定符和时间版本都有自己的记录。另外，文件里没有空记录（null）。如果没有数据，HBase 不会存储任何东西。所以列族的存储是面向列的，一行中一个列族的数据不会存放在同一个 HFile 里。

③ HBase 检索原理

和传统关系型数据库相同的是，HBase 也是通过每行数据的行键（即 rowkey）这个唯一的标识符来区分不同的行。HBase 对于数据的检索都是通过行键进行的。

HBase 对于数据的检索主要有三种方式：

1) 通过单个 rowkey 访问，按照某个 rowkey 值对 HBase 进行 get 操作，这样可以获取到唯一的一条记录。

2) 通过 rowkey 的范围进行扫描，通过设置 startRowkey 和 endRowkey，在这个范围内对 HBase 表进行 scan 操作。这样可以按指定的条件获取一批记录。

3) 没有 rowkey 值时进行全表扫描，即直接扫描整张表中所有行记录，然后获得想要的结果，很明显这种方式的效率是非常低下的。

2.1.3 Spark 简介

Spark 是 UC Berkeley AMP lab 所开源的类 Hadoop MapReduce 的通用并行内存计算框架。Spark 拥有 MapReduce 所具有的一切优点，避免了 MapReduce 的一些缺点。比如 Spark 不同于 MapReduce 的是 Spark 任务的中间输出结果是直接存储在内存中，避免了在计算工程中对 HDFS 的频繁读取，甚至是零读取。由于 HDFS 的 I/O 操作需要大量的耗时，所以就这一点来说 Spark 就比 MapReduce 快速不少。因这一特性，Spark 能更好地适用于需要大量迭代计算的算法。虽然创建 Spark 是为了支持分布式数据集上的快速迭代作业，但是实际上它是对 Hadoop 的补充。Spark 是用 Scala 语言中实现的，它将 Scala 用作其应用程序框架的基本实现语言。与 Hadoop 和 Java 的关系不同的是，Spark 和 Scala 能够紧密集成，在 Spark 中提供了丰富的 Scala 对象接口，使得 Scala 可以非常便利的操作和处理分布式数据集。

Spark 有以下几个特点：快速、简单易用、组件多、通用性强^[17]。

① 快速

Spark 之所以快主要有以下几个原因：

1) Spark 采用了 DAG（有向无环图）的计算模式，大多数计算在运算的过程中只是存储一个计算的过程，在累积到特定的计算时才进行计算。

2) Spark 是基于内存的计算框架，运算所产生的中间结果是直接写入内存的，减少了不必要的写入磁盘 I/O 读写。

3) Spark 将一切存入到 Spark 的数据抽象成了 RDD（resilient distributed dataset），RDD 即弹性分布式数据集。所有的计算都是对 RDD 的操作。

图 2.1 是 Spark 和 Hadoop MapReduce 框架的计算时间对比，可见和 Hadoop MapReduce 相比，Spark 的计算速度是非常快的。

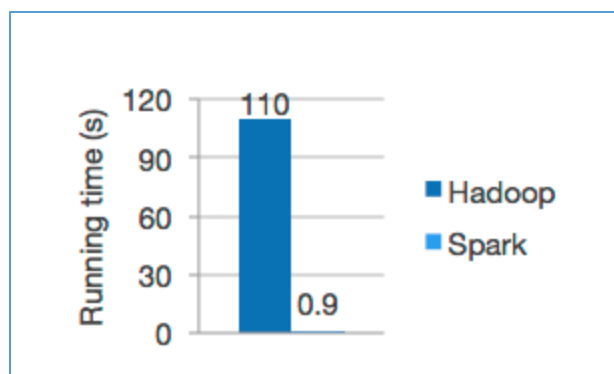


图 2.1 Spark Hadoop 速度对比

Figure 2.1 The speed comparison of Hadoop and Spark

② 简单易用

Spark 将计算划分抽象成了好多个算子，对 RDD 的大多数运算都可以直接调用已经包装好的算子，简化了工程的开发难度，极大的提高了开发效率。

图 2.2 是单词统计在 Spark 中的实现，和 MapReduce 极为复杂冗长的实现代码相比，可见 Spark 任务的实现是非常简洁的。

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
            .map(lambda word: (word, 1))
            .reduceByKey(lambda a, b: a+b)
```

图 2.2 Spark 单词统计示例

Figure 2.2 example code of spark word count

③ 组件多

得益于 Spark 优秀的计算模式, Spark 可以通过很简洁的代码开发出极其强大的功能, 在 Spark 的运行环境中直接包含了内存数据库组件 Spark SQL, 流计算组件 Spark Streaming, 机器学习库 MLlib, 图计算框架 GraphX。在实际开发项目的过程中可以在不添加任何其它组件的情况下直接调用 Spark 提供的功能, 且 Spark 提供的各种组件在类似产品中均属于佼佼者, 因此完全可以考虑采用 Spark 系列的组件实现全部的开发。

④ 通用性强

Spark 可以在 HDFS、Cassandra、HBase、MongoDB 和 S3 等文件系统或者数据库直接读取数据; 可以运行在 Hadoop Yarn、Mesos 等任务框架之上。不用在开发的过程中为了实现平台之间的兼容而编写复杂的组件。

2.2 基础索引算法简介

数据库最主要的功能是存储和查询数据, 使用者在查询数据库的时候希望能以最快的时间得到查询的结构。尤其是在当今海量数据的背景下, 随着数据量的成倍增加查询数据的响应时间也成倍的增加。所以优秀的数据库设计策略也变得越来越重要。索引技术能在很大程度上解决数据查询效率低下的问题。索引技术是围绕着计算机算法中的“以空间换时间”这一策略适当的增加必要的冗余数据来达到快速查询数据的目的。

索引是数据库的一种补充, 这意味着即使没有索引, 数据库仍然可以实现应有的功能。但索引可以在大多数场景下提升数据库的查询性能。在 OLAP 中尤其明显。常见的数据库索引算法都是基于最基本的数据结构中的“树”的不同变种来实现的。比如 MySQL 数据库提供了 B 树索引和 Hash 索引两种数据索引方式, Oracle 数据库中提供了 R 树和四叉索引树两种数据索引方式^[18]。

2.2.1 Hash 索引

Hash 索引包含以数组形式组织的桶集合。Hash 函数将索引键映射到 Hash 索引表中对应的桶。

用于 Hash 索引的 Hash 函数具有以下特征:

- ① 拥有一个用于所有 Hash 索引的 Hash 函数。
- ② Hash 函数具有确定性。同一索引键始终映射到 Hash 索引中的同一个桶中。
- ③ 多个索引键可能映射到同一个桶中, 这时就产生了 Hash 冲突。我们需要设计合理的 Hash 函数来尽量减少 Hash 冲突, 或者设计额外的策略来处理冲突。
- ④ 哈希函数经过数据的均衡处理, 这意味着索引键值在哈希桶上的分布通常符合泊松分布。泊松分布并非均匀分布。索引键值并非均匀地分布在哈希桶中。

如果两个索引键映射到同一个哈希桶，则产生哈希冲突。大量哈希冲突可影响读取操作的性能。

内存哈希索引结构包含一个内存指针数组。每个桶映射到该数组中的一个偏移位置。数组中的每个桶指向该哈希桶中的第一行。桶中的每行指向下行，因而形成了每个哈希桶的行链。

2.2.2 B 树索引

1970 年，R.Bayer 和 E.mccreight 提出了一种适用于外查找的平衡的多叉树，称为 B 树（或 B-树、B₊树）。B 树是二叉树的扩展，一棵 m 阶 B 树是一棵平衡的 m 路搜索树。它或者是空树，或者是满足下列性质的树：

- ① 根结点至少有两个子节点；
- ② 每个非根节点所包含的关键字个数 j 满足： $(m/2)-1 \leq j \leq m-1$ ；
- ③ 除根结点和叶子节点以外的所有结点的度数正好是关键字总数加 1，故内部子树个数 k 满足： $(m/2) \leq k \leq m$ ；
- ④ 所有的叶子结点都位于同一层。

如果 B 树的所有非叶子结点的左右子树的节点数目保持平衡，那么 B 树的搜索性能是接近于二分查找的，但它和连续内存空间的二分查找相比优点是：改变 B 树结构不需要移动大段的内存数据，甚至通常是常数开销^[19]，正是因为这个优点好多数据表和索引采取这种方式存储。

2.2.3 B+树索引

B+树是一种树数据结构，是一个 n 叉树，每个节点通常有多个孩子，一颗 B+树包含根节点、内部节点和叶子节点。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。B+树通常用于数据库和操作系统的文件系统中^[20]。NTFS、ReiserFS、NSS、XFS、JFS、ReFS 和 BFS 等文件系统都在使用 B+树作为元数据的索引。B+树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+树元素自底向上插入。

一棵 m 阶的 B+树和 m 阶的 B-树的差异在于：

- ① 有 n 棵子树的结点中含有 n 个关键字，关键字不保存具体数据，只用来作为索引，B+数中所有数据都保存在叶子节点上。
- ② B+树的叶子结点中包含了全部的关键字，和指向含有这些关键字记录的指针，且叶子结点本身也是有序的，叶子节点之间从小到大有指针链接。
- ③ 所有的非叶子结点都可以看成是 B+树的索引部分，索引节点的值为其子树的中的最大或者最小关键字。

④ 在 B+树上有两个头指针，一个指向 B+树的根结点，一个指向 B+树所有关键字中最小关键字所在的叶子结点。

2.2.4 HT 树索引

HT 树是针对 T 树、CSS 树、CBS 树等缓存敏感索引的不足，综合考虑了 B+ 树索引和哈希索引的优点而提出的索引结构^[21]。HT 树的结构如图 2.3 所示，可以分为两层的架构：非叶子节点为树结构，一般采用 B 树或 B+树实现；叶子节点为一个特定形式的 Hash 表结构，其中的数据已 Hash 表的形式组织。在树结构中因 B+树或 B 树的有序性的特征，HT 树的关键字之间是有序排序的，在叶子部分节点由 Hash 表的特性，数据是无序的。由于 Hash 表的访问速度是非常快的，所以本文中在叶子节点中实现的是一个较大的 Hash 表。较大的 Hash 表中可以存储较多的数据，一方面降低了树 HT 树的高度可以降低 TBL 失配率，同时也可以利用 Hash 表高速访问的特性增大来提高 HT 树的访问速度^[21]。

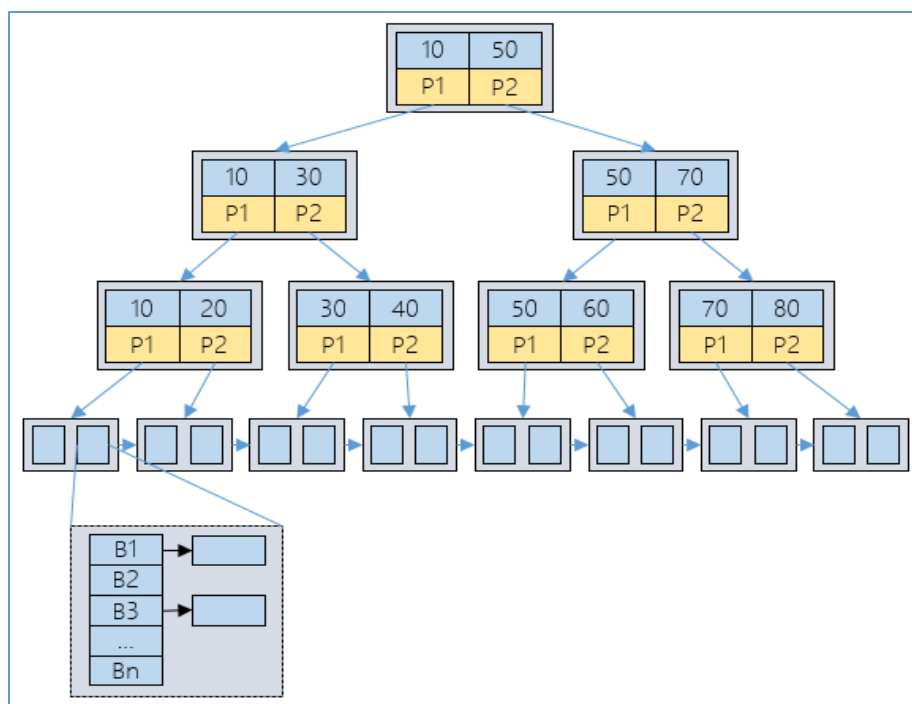


图 2.3 HT 树结构图

Figure 2.3 The architecture of HT Tree

基于 HT 树的以下特点，HT 树的内存查询是非常高效的

① HT 树由于其大量的数据是存储叶子节点的 Hash 表中，这一特点使得 HT 树的深度比较低。低深度的树的一般都会有较低的 TBL 失配率，有着较高的内存查询效率^[21]。

② 由于 HT 树整体的结构是一个 B+树或者类似 B+树的有序的树结构, 所以总体来说 HT 树的关键字之间也是的有序排列的, 另一方面虽然 Hash 表是无序的, 但其访问速度是特别快的。因此, 从整体来说 HT 树有着很良好的访问性、可查询性。

2.3 HBase 查询优化方案

在 HBase 中, 表格中每行的 rowkey 按照字典排序, Region 按照 rowkey 进行分块, 通过这种方式实现的全局、分布式索引。然而, 随着在 HBase 系统上应用的驱动, 人们发现单一的通过 rowkey 检索数据的方式, 不再满足更多应用的需求, 人们希望像 SQL 一样检索数据, 例如 `select * from table where col=val`。可是, HBase 之前的定位是大表的存储, 要进行这样的查询, 往往是要通过类似 Hive、Pig 等系统进行全表的 MapReduce 计算, 这种方式一方面需要大量的计算资源, 另一方面其查询速度达不到大多数使用者所期望的速度。

2.3.1 HBase rowkey 设计原则

HBase 设计的目标旨在能稳定存储大量的数据, 而数据检索方式比较单一, 只能通过 rowkey 来快速检索数据。可以采用以下两种方法来提高 HBase 的查询效率: 优化 rowkey 设计和提供基于 rowkey 的外部索引。由于 HBase 通过 Rowkey 可以快速的检索到数据, 所以在一般的设计中都会通过合理的 rowkey 设计来得到一个可以通过计算等方式能得到的 rowkey, 从而进行快速的检索。

HBase rowkey 的设计要遵循以下 3 个设计原则:

① rowkey 长度原则

rowkey 在实际存储中是 HFile 中的一个二进制字节流, 建议 rowkey 的长度越短越好, 最好不要超过 16 个字节。在 64 位的操作系统中内存是 8 字节对齐的。控制在 16 个字节, 8 字节的整数倍可以使得内存的利用率比较高。如果 rowkey 过长比如 100 个字节, 1000 万行数据光 rowkey 就要占用 $100 \times 1000 \text{ 万} = 10 \text{ 亿个字节}$, 将近 1G 的数据量, 这会极大浪费存储空间, 并且影响 HFile 的存储效率^[22]。

② rowkey 散列原则

rowkey 的散列原则是将数据比较平均的存储在 HFile 文件中。在 HBase 中, rowkey 值相近的数据行往往会被存储在同一个或者相近的节点中。这样在数据读取的过程中如果是读取 rowkey 值连续的行会给 RegionServer 带来比较大的压力, 不利于集群的负载均衡, 同时也会降低集群的查询效率。比如在一些场合中会使用时间戳作为 rowkey 值, 在这种情况下个别 RegionServer 会承受比较大的吞吐量, 而其它节点处于闲置状态, 这种负载的不均衡非常不利于发挥集群的强大性能。

③ rowkey 唯一原则

必须在设计上保证其唯一性。**rowkey** 的优化设计在数据库设计的时候会有比较大的难度, 这种难度来自于随着数据量的累积有意义的 **rowkey** 设计会带来极大的 **rowkey** 冲突。为了避免这种冲突在 **HBase** 数据库设计的时候会有比较大的难度, 可以考虑通过增加外部索引的方法来降低数据库设计的难度。

2.3.2 二级索引架构

二级索引架构类似于 **Oracle**、**MySQL** 等关系数据库中的二级索引, 主要应用于 **key-value** 存储形式的分布式数据库系统中^[3]。在 **HBase** 中, 所有的数据都是存储在一张大表里的。为了提高 **HBase** 的访问效率, 原生的 **HBase** 在 **rowkey** 列以 **rowkey** 值建立了一个类似于 **B+** 树的索引树。然而并没有提供非 **rowkey** 列的索引功能。为了提高非 **rowkey** 列的数据访问速度, 我们可以构建一个所需的非 **rowkey** 列的索引。在索引树里存储非 **rowkey** 列值和其所在行的 **rowkey** 值的键值对应关系, 大概会形成如图 2.4 所示的关系映射。在这一的查询系统中需要先对非 **rowkey** 列的索引表进行查询, 找到非 **rowkey** 列索引相对应的 **rowkey**, 然后在用这个 **rowkey** 在 **HBase** 中快速的获取到我们的目的数据。

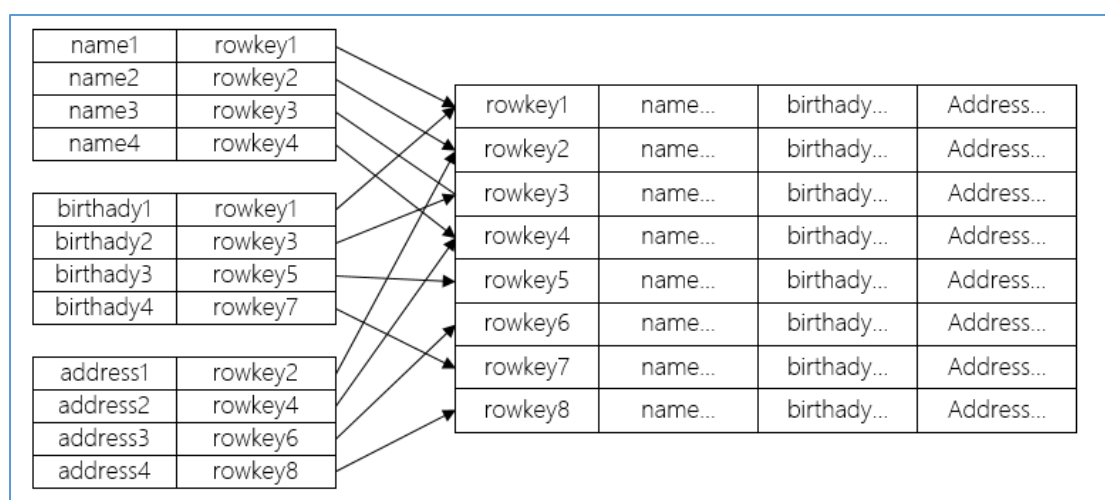


图 2.4 二级索引示例

Figure 2.4 The example of secondary index

现有的基于二级索引架构实现的 **HBase** 索引方案主要有以下几种, 分别是开源的实现方案 **IHBase** 和 **ITHBase**, 中国科学院提出的互补聚簇式索引 (complemental clustering index) **CCIndex**^[23], 异步视图方案 **Asynchronous views**^[24], 以及华为公司基于 **Coprocessor** 技术的 **Hindex** 索引和中兴的 **HiBase** 系统。

IHBase 主要特点是在 **HBase** 的 **region** 级别建立索引而不是在表级别。**IHBase** 在内存中为 **region** 维护了一份索引, 在数据扫描的时候首先在索引中查找数据,

得到目的 rowkey 列表,而在常规的查询时,能利用已得到的 rowkey 来快速的获取数据。ITHBase 的全称是 Indexed Transactional HBase,从名称可以看出事务性是它的重要特性。ITHBase 会建立一张事物表,每次进行事物操作的时候都会先在事物表里保存已操作的状态,知道遇到 ITHBase 定义的 commit 操作时才会对正真存储数据的表进行修改。IHBase 与 ITHBase 都是在 HBase 的源码上进行了一些修改和扩展而实现的,重新定义和实现了 HBase 的一些辑和组件,因此它们在实现上来说都有比较大的难度,且对 HBase 源码的一些修改也带来了很大的入侵性^[3]。

中国科学院提出的互补聚簇式索引 CCIndex 主要目的是为了满足不同海量数据条件下多维区间检索的需求^[25]。常规的二级索引均需要在索引中获取到 rowkey 值,之后再从 HBase 中读取数据。由于获取到的 rowkey 在 HBase 中的顺序是随机的,大量的根据 rowkey 的随机读取 HBase 的性能并不是很佳。CCIndex 解决这一问题的方法是将索引列的详细数据也存储在索引表里,但这一解决方案会带了存储空间的大量浪费。尤其是当系统中存储着大量的索引的时候,所需要的存储空间会出现成倍的增长。为了解决存储空间的问题,CCIndex 中抛弃了 HDFS 的冗余备份功能,将 HDFS 的备份数设置成了 1。由于 HDFS 本来就不保证单一文件的可靠性存储,这样一来 CCIndex 的容错性成了又一个需要解决的问题。为了实现良好的容错性,CCIndex 又设计了自己的容错方案。这样无疑又增加了系统的难度。总的来说 CCIndex 的设计理念比较简单,但其对原生系统的一些修改并不是良性的,反而会带出其它一些问题。最主要的问题是这种方案在索引较多的时候依然存在这对存储空间的极大浪费,且由于其索引中直接存储数据内容的设计是不利于索引的修改的。

Asynchronous View^[24]提出了一种似于二级索引的异步视图的方式来实现非 rowkey 列的索引查询^[3]。有两种主要的视图方案:远端视图表 RVTs 和局部视图表 LVTs。RVTs 有点类似于 Mysql 的视图方案,针对某个查询建立一个视图。这种方式对于查询来说效率比较高,但在频繁的数据更新后,视图和其索引的数据可能不会在同一个节点中,需要额外的操作来保持视图和其索引的数据所在节点的一致性。LVTs 视图是针对每个节点的数据建立的一个局部的视图,其特点是在数据库频繁更新时视图的维护代价是比较低的。但因视图只提供单一节点数据的索引,在涉及到大量节点查询的时候需要对每个节点的查询结果进行合并操作,在一定程度上降低了查询的效率。总体来说,LVTs 视图常用于聚集类型的查询,其他查询采用 RVTs 视图来实现,对于一些复杂的查询可以采用两者相结合的方式来实现综合的视图查询^[3]。

2.3.3 华为 HBase 索引方案

HBase 的 Coprocessor 特性提出以后,出现了一些基于 Coprocessor 特性的二级索引,比如华为的 Hindex 方案。下面对 Hindex 做一个简单的介绍。

华为 Hindex 在设计中主要有以下两个要点：保证主表和索引表在同一个 regionserver 上(通过自定义的 balancer 实现)；使用 coprocessor 实现索引表的创建和插入。HiBase 主要由三部分组成：Client Ext, Balancer, Coprocessor。这三部分各自的功能分别是：

① Client Ext

扩展的 HBase Client，主要是在创建 table 的时候，添加了指定特定 index 的细节，其他与 HBase 原生 API 没有差异。

② Balance

华为 HiBase 重写了 balance 的实现，主要是为了保障 user 表的 region 与其一对应的 index 表的 region 存放在一个 RegionServer 上。假如 user region 发生了迁移，那么与之对应的 index region 也需要迁移到同一个地方；同理，如果 index region 发生了迁移，那么其对应的 user region 也需要迁移到对应的地方。

③ Coprocessor

华为 HiBase 相对原始 HBase core 添加了更多的 coprocessor 函数，主要是为了保证 user table 进行数据更新时，index table 的 index 数据也及时的做出了对应的更新。这里需要面对一个问题：user table 更新操作与 index table 更新操作连接在一起是一个事物操作。这两个更新操作都成功则更新，若有一个失败则不更新任何 table。实际上，在 HBase 里面，对于一行的操作比如 put(包括 prePut, Put, postPut)是一个原子操作，那么通过 coprocessor 实现 index table 数据更新与 user table 数据更新一起满足事务性要求。

华为 HiBase 很好的利用了 HBase 的 coprocessor，通过利用 actual region 与 index region 始终保持存放在一个 RegionServer 上的核心思想，实现了 HBase 二级索引的功能。但是其对 HBase core 的一些侵入性改动，使得 HBase 的升级不得不考虑其是否适应，尤其是 HBase 0.96 进行了很多的改动，并且含有对以前版本不兼容的修改。另外 HiBase 刚开源不久，在具体的生产环境中还未经受考验，其是否成熟还是一个问题。还有从其 RoadMap 看出，华为 HiBase 索引目前是不支持动态加载和删除索引，索引只能在表创建的时候指定，这在实际运用中是不可接受的。最后还有个问题，华为 HiBase 索引是不能够对已有的表建立索引的，只能在表建立后，随着数据的增长，慢慢建立起索引。

2.4 本章小结

本章第一节介绍了 Hadoop、HBase 和 Spark 三种分布式数据存储和处理的软件，分析了其特点以及使用的场景，为第 4 章中间件的实现的工具选择打下基础。在第二节介绍了索引技术的原理以及一些索引算法并分析了每种索引算法的特点，

研究来其适用的情况。第三节主要介绍了现有的一些 HBase 索引的架构，并分析了每种架构的特点和适用情况，以及华为 HBase 索引解决方案作为我们索引系统设计的参考。

3 索引系统的设计

3.1 索引系统方案

上文我们较详细的描述了 HBase 的查询原理，从 HBase 的查询原理中可以看出要实现 HBase 的快速检索就必须在检索条件中提供一个有效的 rowkey。上文也介绍了 HBase 的 rowkey 设计原则，HBase 良好性能的 rowkey 设计必须要遵循长度有限性、散列性和唯一性三个原则。同时为了方便查询，查询之前我们就应该得到一个有效的 rowkey。综合上述条件我们可以发现构建一个合适的 rowkey 是有一定的难度的，且在大多数的业务中提供 rowkey 只是为了便于我们查询 HBase 数据库，和我们要处理的业务逻辑没多紧密的关系。提供非 rowkey 列的索引一方面可以优化 HBase 在复杂条件下的查询效率，另一方面也可以让系统在实际应用中的业务逻辑显得更加清楚。

在不支持索引的数据库中使用索引最为常见的策略就是提供外部的索引，外部索引的实现有多种方式。本文第二章所描述的二级索引架构是比较直观，且有着很高适用性的索引设计方案。设计索引最主要的目标是优化数据库的查询性能，但索引的处理也是需要一定的时间和空间的开销。因此，为了优化索引系统的性能，我们要采用比较合理的索引架构和索引算法来实现索引系统。索引的时间效率主要是由索引的运行平台和索引的算法综合决定。众所周知，内存中对数据直接运算的效率是高于从硬盘读取数据然后在进行运算的效率的。我们可以采用将索引数据直接缓存到内存的方式来避免频繁的 I/O 操作从而增加了索引的效率。但当需要运算的索引数据量比较大的时候单节点的内存并不能够完全的将所有的数据缓存到内存中。Spark 是一个性能非常高的分布式内存计算平台，我们可以将索引数据直接缓存到 Spark 分布式内存中进行计算，从而获得一个比较大的内存计算环境。同时，数据库的使用是不定时的，每次执行一次数据库操作的时候都需要进行一次计算。Spark 的流处理模式 Streaming 能够很好的支持这种需求。

综上所述，本文的索引系统方案可以总结为基于 Spark Streaming 的 HBase 内存索引系统。系统的方案图如图 3.1 所示。索引方案主要由索引核心组件（Index Core）、任务分配组件（Butler）以及对 HBase 和 Spark 的一些操作组件组成。在任务分配组件中将任务进行分流，分为可以直接在 HBase 中执行的任务和需要在索引中执行的任务。这种任务分流可以减少不必要的索引处理。索引核心组件是由索引树和相关的算法组成的，系统运行中的所有的索引树都存储在索引核心组件中。整个索引系统的数据是存储和运行在内存中的，同时在 HDFS 中生成备份。

索引系统和 HBase 中的 rowkey 索引共同组成了 HBase 的二级索引方案。

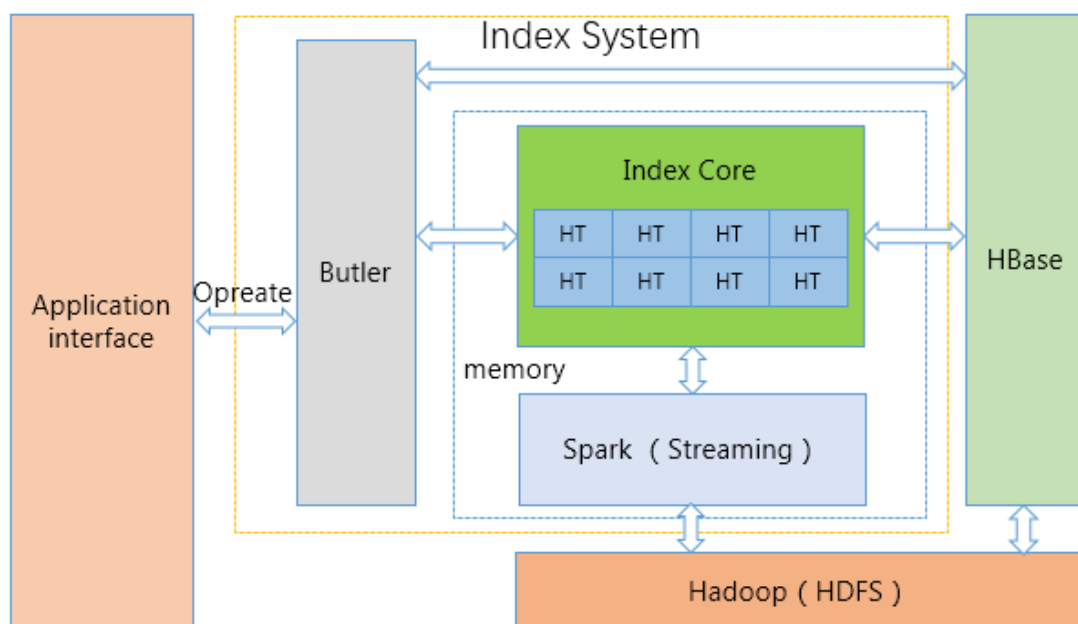


图 3.1 索引方案图

Figure 3.1 The scheme graph of index

3.2 索引算法的选择

系统设计的目标是实现一个快速有效的索引系统。因此在选择和设计索引算法时我们需要考虑以下两个原则：

- ① 索引结构要适用于分布式内存的存储。
- ② 索引算法要有较高的查询效率。

常见的索引算法只要有一下一些：Hash 索引、B 树索引、B+树索引、AVL 树、T 树、pB+树、fB+树、CSS 树、CSB 树、HT 树以及其它一些针对上述索引算法而提出的改进型的算法。下面我们分别简单描述各种算法性能的优点和缺点，结合上述的两个索引算法选择的原则来确定我们的内存索引系统中采用哪种算法合适。

B 树和 B+树是磁盘数据库中广泛使用的索引技术，能最大化地减少 I/O，空间的利用率为 60%左右^[26]，其 60%的空间利用率并不适于在其之上建立索引并存入内存中。

在 T 树最开始被提出的时候，其 CPU 效率被认为比 B 树要好^[27]。但是随着 CPU 的速度和内存的访问速度的发展不平衡，CPU 缓存的容量也在不断被扩充，研究发现高速缓存对数据库性能有着非常显著的影响^[28]，T 树的结构决定其会有比较深的树结构，而高度比较深的树由于其较高的 TBL 失配率不能很好的适用于

当前的硬件环境。

基于高速缓存对索引树的影响的研究导致了一系列缓存敏感树的提出,比如 pB+树、fB+树、CSS 树、CSB+树以及在 B+树和 Hash 索引的基础上提出了 HT 树^[21]。这些缓存敏感树在很大程度上减少了高速缓存对索引的影响,提高了内存索引的效率。

综合考虑上章提出基础索引算法和上述的一些索引算法可以得到以下结论:

① Hash 索引的随机查找具有极好的性能,但是 Hash 索引的存储效率是非常低的,不适合于内存中的存储。

② AVL 树的查找和更新效率都还不错,但其存储效率较低,同时也有着很高的 TBL 失配问题,不适合于内存中的存储。

③ T 树因其较高的 TBL 失配率导致其查询开销较大,降低了查询效率。

④ B 树的存储效率较高,查询效率在磁盘文件中是低于 B+树,在内存中高于 B+树。

⑤ B+树的 TBL 失配率比 AVL 树和 T 树都要好,但 B+树的关键字也会出现在叶子节点中会导致数据的冗余,且每个结点保存值和指针,所以其存储效率相对较低,不适合于存储在内存环境中。

⑥ pB+树、fB+树均是 B+树针对高速缓存的优化产生的。由于其会将全部节点载入缓存中的特性一方面缓存利用率较低,一方面占据大量缓存会引起缓存震荡的问题,且其限制于缓存的大小,程序移植也比较麻烦。

⑦ CSS 树和 CSB+树有着很高的缓存意识,缓存块利用率高,且查找速度要快于 B+树,同时也能节省结点空间^[29]。但 CSS 树和 CSB 树的更新效率比较差,次更新索引树的效率和重建索引树的效率相近。

⑧ HT 树的查询效率是高于 CSB+树的,且其插入性能和可维护性均比较好^[21]。

综上所述,我们采用 HT 树来作为 HBase 非 Rowkey 列的索引的实现算法。

3.3 HT 树的操作

用 HT 树作为索引方式主要完成三个工作:查找,插入,删除,其中插入和删除都是以查找为基础,HT 的树结构在插入和删除时操作有时候会影响到 HT 树的结构,为了维护 HT 的树结构在插入和删除后会根据情况对节点进行分裂或者合并的操作。因为 Hash 表会有 Hash 冲突的问题,在本文中采用分离链表法来解决这个问题。在本文中对 HT 树原来的插入和删除进行了一些调整,优化了 HT 树的空间利用率。下面分别介绍几种操作的实现算法。

① HT 树的查找

优秀的索引算法都会支持对指定值得精确查找和对一系列键值的范围查找。

HT 树的查找分为两个步骤：首先要得到值所在的叶子节点，即得到 Hash 表；然后从 Hash 表中得到数据。第一部分即 B+树的查找，第二部分即 Hash 表的查找。HT 树的查找是递归形式的查找，具体的查找过程为：

若当前所查询的节点为叶子节点，则通过 Hash 算法得到 Key 在 Hash 表中的桶，然后在桶中顺序查找可得到 Key 的具体位置，从而得到 value 值。

如果当前查询的节点不是叶子节点，根据 B+树的性质，该节点中的关键字是按照从小到大排列的，我们可以根据 Key 和关键字之间的大小关系定位到 B+树的下一级节点，然后在这一子节点上递归运用上述查找过程。

② HT 树的插入

首先通过 HT 树的查找算法找到该 Key 在 HT 树中所对应的位置，即找到其要插入的叶子节点。然后就是在 Hash 表中的插入，计算该 Key 在 Hash 表中具体的桶，然后将 Key 和对应的 value 值插入到该桶中。如果插入后 Hash 表的填装因子小于最大填装因子，则插入操作完成。如果该 Hash 表的填装因子最大填装因子，则引发节点的分裂。节点分裂的具体步骤下文将详细描述。节点分裂完成后即完成了 HT 树的插入操作。

HT 树的分裂操作可以分为叶子节点的分裂和非叶子节点的分裂。非叶子节点的分裂一般都是叶子节点的分裂所引发的。

假定插入后某个叶子节点 N 中的 Hash 表的当前填装因子大于最大填装因子，则引发叶子节点和 Hash 表的分裂操作。Hash 表的分裂方法按照 Hash 表中 Key 的个数将 Key 值从小到大分为两段，保证前半段的 Key 值均小于后半段的 Key 值，然后分别在两个 Hash 表中散列两断数据即可完成 Hash 表的分裂。最后创建一个新的叶子节点 M 来保存分裂出来的 Hash 表，由于 B+ 树是有序的，新创建的该节点 M 将会是在原来节点 N 的兄弟节点，位置在原节点的右边。

叶子节点的分裂操作有可能引起 HT 树结构出现问题，当插入一个新节点的时候需要判断结构是否合理。若合理则分裂完成，若不合理则进行 HT 树结构的调整，即引发 HT 树上层节点的分裂操作，HT 树上层节点的分裂操作即 B+树的分裂操作。

③ HT 树的删除

HT 树的所有数据都存储在叶子节点中，所有的删除操作最开始的步骤都是通过 HT 树的查找算法找到该 Key 在 HT 树中所对应的位置。若该 Hash 表中的 Key 值的数目为 1，则直接删除该叶子节点。若该 Hash 表中的 Key 值的数目大于 1 然后判断删除该 Key 后 Hash 表的填装因子是不是小于最小填装因子，若大于最小填装因子则删除该 Key 即其对应的 value 值。若小于最小填装因子则将该节点则判断该节点和其相邻节点填装因子的平均值是否大于最大填装因子，若小于则将两个节点合并，将两个节点的 Hash 表重新散列为一个 Hash 表。若该节点和其相邻节

点填装因子的平均值是否大于最大填装因子，则将两个 Hash 表的内容合并按照其中 Key 的个数平均后重新划分为两个 Hash 表，并求该关键字节点的 Key 值。完成上述操作即完成了 HT 树中 Key 的删除。

删除叶子节点之后有可能会破坏 HT 树的结构，这时会引发上层关键字节点的删除操作，关键字节点的删除算法参考 B+树的删除算法。

3.4 系统设计

HBase 数据管理系统是按照 rowkey 的顺序对数据进行组织，并在 rowkey 上建立类似 B+树的索引结构，所以在 rowkey 上能够提供高效的点查询或范围查询。而目前版本的 HBase 系统没有提供非 rowkey 的二级索引功能，当用户基于非 rowkey 查询 HBase 数据的时候，只能通过 Scan 全表扫描或者使用 MapReduce 架构全表扫描获取满足条件的数据，但这两种方式效率太低，无法满足实时查询的需要。

虽然我们可以利用 MapReduce 技术来实现数据访问的并行化，在一定程度上提高查询速度，但是当数据量非常大的时候，对于时间延迟要求比较高的应用来说，全表扫描所需的时间仍然比较长。因此我们需要通过其它的方法来设计。如何建立既能提高查询效率，又不会导致系统结构复杂和管理困难的二级索引，这既是本文的研究重点和关键问题。本文借鉴和分析已有的方案和等方案，并分析系统的特性，最终采用了基于 Spark 建立二级索引。

在网络环境中，建立在具有基本通信协议的操作系统之上，支持应用软件有效开发、部署、运行和管理的支撑软件称为分布计算中间件^[30]。分布计算中间件是一种起承上（应用软件）启下（操作系统）作用的支撑软件，它支持一体化网络计算，故又称为网络计算中间件，或称为软件中间件，简称中间件。中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。中间件位于 C/S 操作系统之上，管理计算机资源和网络通讯。是连接两个独立应用程序或独立系统的软件。相连接的系统，即使它们具有不同的接口，但通过中间件相互之间仍能交换信息。执行中间件的一个关键途径是信息传递。通过中间件，应用程序可以工作于多平台或 OS 环境。

本系统旨在与提供一个 HBase 与 Spark 之间的非侵入式的中间件。采用非侵入式设计的优点在于中间件是基于 HBase 和 Spark 平台的，但运行中间件时不需要对 HBase、Spark 和原有数据做出任何修改。这样可以在不修改原有环境和数据的前提下将中间件直接运行在现有的数据库系统之上。我们可以根据是否有需求来选择性的使用中间件提供的接口，当中间件出现任何问题的时候也不会造成数据丢失和环境的损伤。

3.4.1 系统架构图

索引系统架构如图 3.2 所示。整个系统架构在 Hadoop、HBase、Spark 等分布式软件之上。其中 Hadoop 的 HDFS 提供分布式文件系统支持；HBase 提供分布式数据库支持；Spark 提供分布式内存计算支持，索引系统的基本运算都是基于 Spark 的内存计算的，同时采用了 Spark 的 Streaming 组件实现了任务的交互和实时处理。在 HBase 和 Spark 之间提供了索引系统，通过 Spark 强大的内存计算能力可以快速的得到 rowkey，拿到 rowkey 后我们可以在 HBase 中快速的读取数据。在 Index System 之上提供外部访问的 API 接口，实现外部系统对索引数据库的访问。

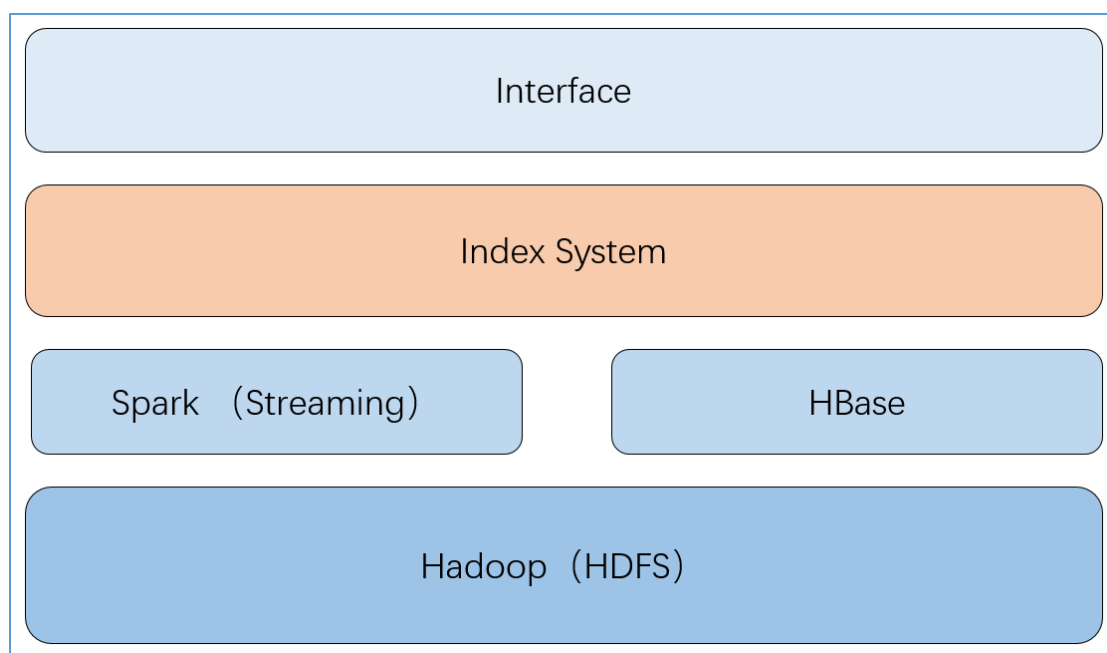


图 3.2 本文索引系统架构图

Figure 3.2 Architecture of the index system in this paper

在索引系统启动的时候首先要在 HDFS 中加载序列化后的索引树到内存中。之后所有关于索引树的操作都是直接在内存中进行的，系统每隔一段时间会将内存中的索引树序列化保存到 HDFS 中，实现索引树的备份，防止系统在非正常退出等情况下丢失索引。

3.4.2 主要操作流程

① 写入数据

流程如图 3.3 所示：

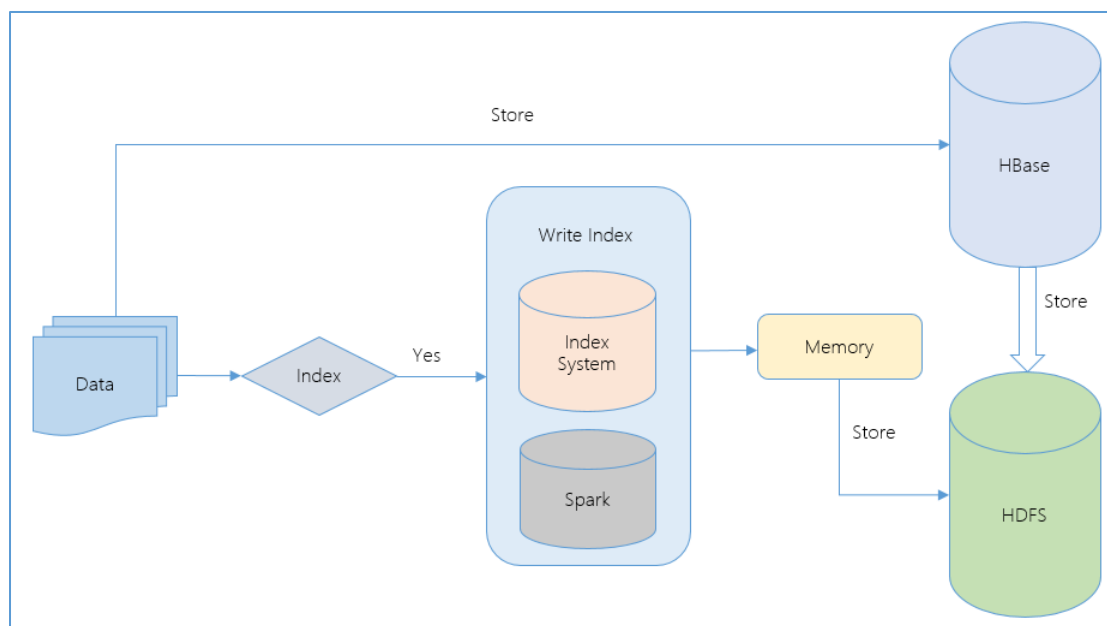


图 3.3 存储数据流程图

Figure 3.3 flow of data store

在写入数据的过程中首先要判断数据中是否有我们要创建索引的列，如果没有直接写入 **HBase** 数据即可，如果有还需要在索引树中写入索引项。此时的索引树是保存在内存中的，系统每隔一段时间会将索引树序列化保存到 **HDFS** 中，防止因为硬件或者软件的原因丢失索引数据。

② 查询数据

数据查询流程如图 3.4 所示：

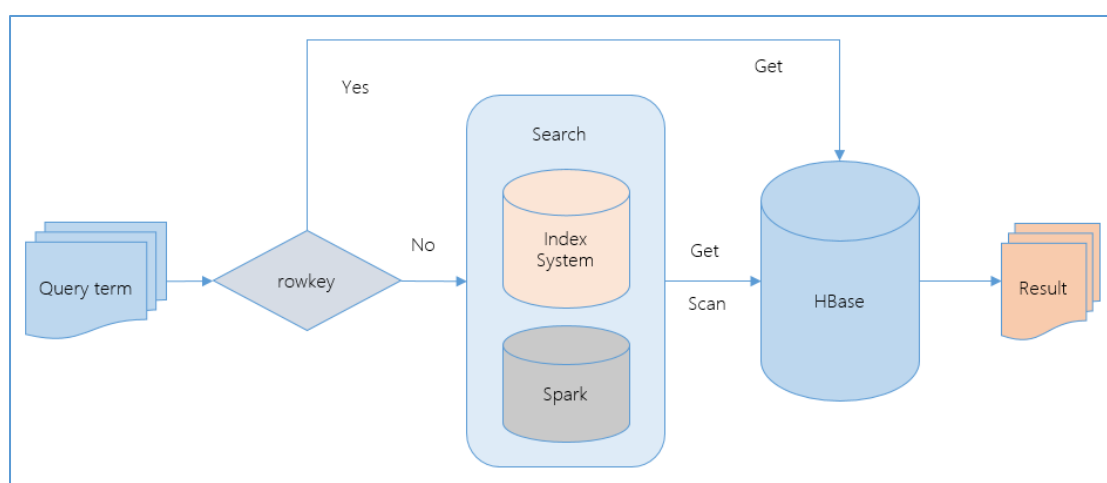


图 3.4 读取数据流程图

Figure 3.4 flow of data read

在读取数据时先判断查询条件里是否有 rowkey，如果有 rowkey 则直接从 HBase 中读取数据；如果查询条件里没有 rowkey，则先在索引系统中根据查询条件读取 rowkey，如果能读取到 rowkey 则根据 rowkey 在 HBase 中快速的读取数据；如果在索引树中没有读取到 rowkey 则只能对 HBase 进行一次全表扫描来读取数据。

③ 删除数据

数据删除流程如图 3.5 所示：

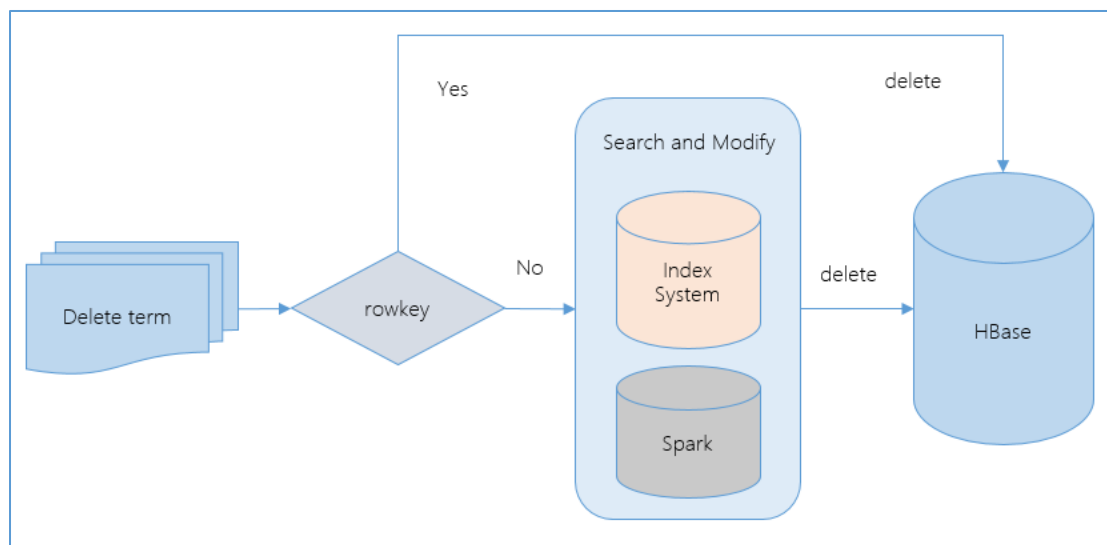


图 3.5 删除数据流程图

Figure 3.5 flow of data delete

在删除 HBase 数据的时候先判断删除条件中有没有 rowkey，若有 rowkey 则直接在 HBase 数据库中删除行。若删除条件中没有 rowkey 则先在索引系统中查询删除条件对应的 rowkey，然后在用这个 rowkey 在 HBase 中删除数据，之后再修改索引树，在索引树中删除已删除数据的索引项。

3.4.3 接口的架构图

接口实现的架构图如图 3.6 所示：

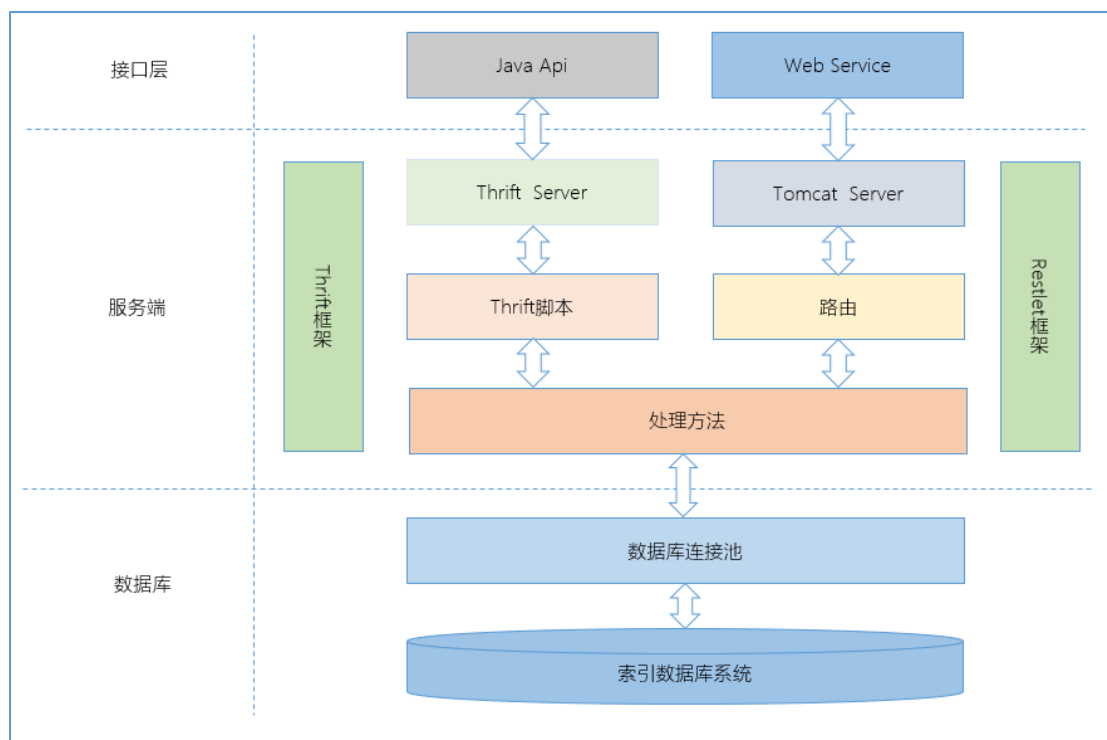


图 3.6 API 架构图

Figure 3.6 The architecture of API

系统的接口实现主要依赖于 thrift 和 Spring 的 Boot 框架，在服务端通过中间件的访问接口访问数据库，将数据库里拿到的结果进行 json 格式的包装，然后以 Thrift 和 Restfull 的形式提供接口，实现用户对数据的多种 API 形式的访问。

3.5 本章总结

本章第一节分析了一些常见索引算法的优点和缺点，最终选择 HT 树索引算法作为我们索引系统的实现算法；第二节对第二节选择的 HT 树索引算法的操作进行详细的查找、插入和删除操作做详细的描述；第三节介绍了系统整体的设计理念，经过分析选择在 Spark 上搭建 HBase 索引系统。

4 索引系统的实现

4.1 系统环境的搭建

4.1.1 软件版本的选择

① 硬件配置

CPU: Intel Core i5 M520 2.40GHz

内存: 6G

硬盘: 三星 850 EVO 120G SATA3 固态硬盘

② 核心软件版本

由于整个项目会用到比较多的开源软件，我们要考虑组件之间的兼容性，主要是 Hadoop、HBase 和 Spark 之间的兼容性。查询上述软件的官网文档可以得出如

表 4.1 所示软件版本配置：

表 4.1 主要软件版本

Table 4.1 The version of main softwares

软件	版本
Linux	CentOS-7-x86_64-LiveKDE-1511
Java	jdk-7u79-linux-x64
Hadoop	hadoop-2.6.1
HBase	hbase-1.2.0
Spark	spark-1.6.0-bin-hadoop2.6

其它工具的版本如

表 4.2 所示：

表 4.2 次要软件版本

Table 4.2 The version of minor softwares

软件	版本
----	----

Window	Windows 10 Professional
虚拟机	VMware-workstation -12.0.1
连接工具	Xshell 5.0.0.31
Thrift	thrift-0.9.3
Eclipse	Eclipse Mars.1 Release (4.5.1)

4.1.2 Linux 基本系统安装

基本系统的安装过程如下：

- ① 安装虚拟机软件 VMware Workstation。
- ② 安装 CentOS 到虚拟机。
- ③ 配置 Host。
- ④ 配置 SSH。
- ⑤ 安装 Java。

上述步骤的具体操作略。

4.1.3 Hadoop 的安装

在官网下载 `hadoop-2.6.1`，上传到 `centos` 的 `/usr/local` 文件夹下。解压缩 `hadoop-2.6.1.tar.gz`。hadoop 的主要配置文件有以下几个，都在 `/hadoop/etc/hadoop` 文件夹下：`hadoop-env.sh`、`yarn-env.sh`、`slaves`、`core-site.xml`、`hdfs-site.xml`、`mapred-site.xml`、`yarn-site.xml`

下面逐一进行配置。

- ① 配置 `hadoop-env.sh` 文件

此文件记录脚本要运行的环境变量，我们需要修改 `JAVA_HOME` 为系统的

- ② 配置 `yarn-env.sh` 文件

此文件记录脚本要运行的环境变量，修改 `JAVA_HOME`

- ③ 配置 `slaves` 文件

此文件用来保存 `slave` 节点，需要注意的是每行只能添加一个 `hostname` 或者 `ip`。

- ④ 配置 `core-site.xml` 文件

这个是 `hadoop` 核心配置文件，里面包含 `HDFS` 和 `MapReduce` 常用的 `I/O` 设置。这里需要配置的就这两个属性，`fs.default.name` 配置了 `hadoop` 的 `HDFS` 系统的命名，位置为主机的 `9000` 端口；`hadoop.tmp.dir` 配置了 `hadoop` 的 `tmp` 目录的根位置。这里使用了一个文件系统中没有的位置，所以要先用 `mkdir` 命令新建一下。

- ⑤ 配置 `hdfs-site.xml` 文件。

这个是 `hdfs` 的配置文件，`dfs.http.address` 配置了 `hdfs` 的 `http` 的访问位置；`dfs.replication` 配置了文件块的备份副本数，一般默认配置为 `3`。

- ⑥ 配置 `mapred-site.xml` 文件

这个是 mapreduce 任务的配置, 由于 hadoop2.x 版本使用了 yarn 框架来管理和调度任务, 所以要配置相关信息实现任务的分布式处理。具体的配置项的详细内容: mapreduce.framework.name 属性下配置为 yarn, mapred.map.tasks 和 mapred.reduce.tasks 分别为 map 和 reduce 的任务数。其它属性为一些进程的端口配置, 均配在主机下。

⑦ 配置 yarn-site.xml 文件

yarn-site.xml 为 yarn 框架的配置信息, 主要是 yarn 的启动初始化相关的配置

⑧ 测试 Hadoop

格式化 hdfs, 启动 hdfs 和 yar

4.1.4 HBase 的安装

从官网下载 hbase-1.2.0, 上传到 centos 的 /usr/ local 文件夹下。Hbase 主要的配置文件有这三个: hbase-env.sh、hbase-site.xml、regionservers.xml。均在 ~/hbase-0.98.17-hadoop2/conf/路径下, 下面将逐一进行配置

① 配置 hbase-env.sh 文件

JAVA_HOME 表示系统的 jdk 路径; HBASE_CLASSPATH 是 habse 配置文件路径; HBASE_MANAGES_ZK 配置系统是否启用默认的 ZooKeeper, 若采用 hbase 自带的 ZooKeeper 则设置为 true, 若采用独立的 ZooKeeper 则设置为 false。本系统采用 hbase 自带的 ZooKeeper。

② 配置 hbase-site.xml 文件

hbase.rootdir 表示 HBase 在 HDFS 中的路径, hbase.master 表示 HBase master 节点的主机名或者 IP 地址, hbase.zookeeper.property.clientPort 属性表示 zookeeper 运行的的端口号, hbase.zookeeper.property.dataDir 属性表示 zookeeper 的缓存目录。

③ 配置 regionservers 文件

此文件用来保存 slave 节点, 需要注意的是每行只能添加一个 hostname 或者 ip。

④ 启动 HBase

4.1.5 Spark 的安装

① 从官网下载 scala-2.11.7, 上传到 centos 的 /usr/ local 文件夹下。解压 scala-2.11.7.tgz。

② 配置 scala 的环境变量。

③ 从官网下载 Spark-1.6.0-bin-hadoop2.6, 上传到 centos 的 /usr/ local 文件夹下。解压缩 Spark-1.6.0-bin-hadoop2.6.tgz。Spark 的配置文件主要有 Spark-env.sh 和 slave, 都在 conf 文件夹下。

④ Spark-env.sh 文件

是 Spark 运行时读取的环境变量, 其中 JAVA_HOME 为 jdk 路径,

SCALA_HOME 为 scala sdk 路径, SPARK_MASTER_IP 为 master 主机的 ip, SPARK_WORKER_MEMORY 为每个 Spark work 节点可使用的最大内存, HADOOP_CONF_DIR 为 Hadoop 配置文件所在的目录。

⑤ Slave 文件, 主要用来保存 hostname。

⑥ 执行 sbin/start-all.sh 启动 Spark。

4.2 中间件的设计

设计本系统最重要的环节是在 HBase 之上借助 Spark 强大的内存计算能力生成非 rowkey 列的索引树。考虑到基于 HBase 和 Spark 源码的二次开发的复杂性, 为了降低开发的难度我们可以在 Spark 和 HBase 之间采用非入侵式、低耦合的中间件模式来实现非 rowkey 列的索引树。中间件的架构图如图 4.1 所示:

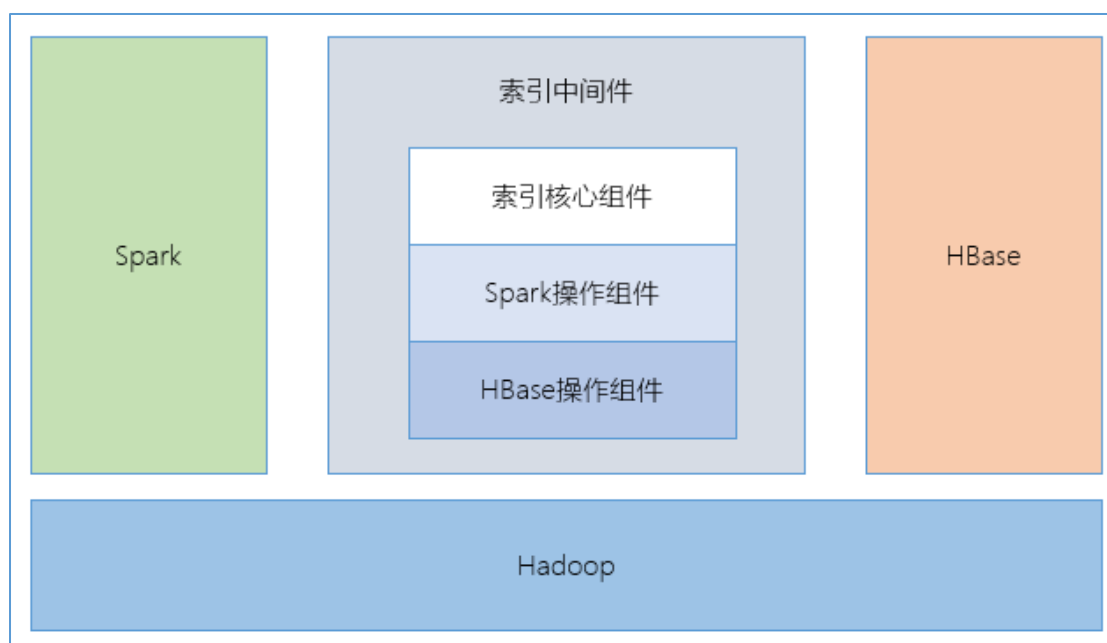


图 4.1 中间件架构图

Figure 4.1 Architecture of middleware

中间件分为以下几个组件:

索引数核心组件, Spark 操作组件, HBase 操作组件。其中索引树核心组件是 HT 索引树的实现; Spark 操作组件是将我们所需要的 Spark 的功能封装成易用的接口, 并且负责组件之间的相互耦合; HBase 操作组件主要是实现 HBase 常用操作的封装。

4.3 中间件的实现

4.3.1 HBase 操作组件

HBase 操作类主要方法如下所示：

```
1 public class HBaseUtils {  
2     public static void setConfiguration (List<Map<String,String>> config){...}  
3     public static boolean createTable(TableName tablename, String[] families) {...}  
4     public static boolean deleteTable(TableName tablename) {...}  
5     public static boolean putRow(HTable table, String rowKey, String columnFamily, String  
    identifier, String data) {...}  
6     public static Result getRow(HTable table, String rowKey) {...}  
7     public static boolean deleteRow(HTable table, String rowKey) {...}  
8     public static ResultScanner scanAll(HTable table) {...}  
9     public static ResultScanner scanRange(HTable table, String startrow, String endrow)  
    {...}  
10    public static ResultScanner scanFilter(HTable table, String startrow, Filter filter) {...}  
11 }
```

① `setConfiguration` 方法用来设置 HBase 的一些基础连接信息，其参数为 HBase 的配置信息列表。

② `createTable` 方法方法用来创建表，返回值为 `boolean` 类型，其参数为表名称和表中的列族列表。

③ `deleteTable` 方法用来删除表，返回值为 `boolean` 类型，其参数为表名称和表中的列族列表。

④ `putRow` 方法用来添加一个数据，其参数为表名称、列族名称、标识符和数据值。

⑤ `getRow` 方法用来获取一行数据，返回值 `Result` 类型的结果，其参数为表名称和列族名称。

⑥ `deleteRow` 方法用来删除一行数据，返回值为 `boolean` 类型，其参数为表名称和列族名称。

⑦ `scanAll` 方法用来获取所有数据，返回 `ResultScanner` 类型的一个结果集，其参数为表名称。

⑧ `scanRange` 方法用来返回保存指定 Rowkey 范围的结果集的 `ResultScanner`，其参数为表名称、起始 Rowkey 和结束 Rowkey。

⑨ `scanFilter` 方法用来返回保存满足指定 Rowkey 过滤条件的扫描结果集的

ResultScanner 其参数为表名称、起始 Rowkey 和过滤条件。

4.3.2 Spark 操作组件

在文中我们采用 Spark Streaming 来实现任务的实时处理。Spark 操作类的实现如下：

```

1  public class SparkUtils {
2      public static JavaStreamingContext getContext(){...};
3      public static JavaReceiverInputDStream<String> getDatas(JavaStreamingContext
    jsc,String hostname,int port) {...};
4      public static List<String> indexOpreate (JavaReceiverInputDStream<String> datas) {...};
5      public static void returnDatas(List<String> keywords) {...};
6      public static void taskStart(JavaStreamingContext jsc) {...};
7  }
```

① JavaStreamingContext 方法用来获取了 Spark 的上下文环境。

② getDatas 方法用于在指定端口接收传入的查询条件并返回一个 JavaReceiverInputDStream，其参数列表中需要传入 Spark 的上下文环境、主机名和端口。

③ indexOpreate 方法内部操作索引树，并返回 json 字符串格式的操作结果，其参数为 getDatas 方法中返回的操作数据。

④ returnDatas 方法将 dataProcess 方法中获取到的结果字符串写入指定的端口，其参数为 dataProcess 中返回的 keyword 列表。

⑤ taskStart 方法用来打包运行上述方法在指定端口运行。

4.3.3 索引树操作组件

主要有 HT 树的节点类、HT 树类和 HT 索引系统的外部操作类。

① 节点类提供两个构造函数，分别如下：

1) 非根节点的构造函数

```

1  public HTNode(boolean isLeaf) {
2      this.isLeaf = isLeaf;
3      entries = new ArrayList<Entry<Comparable, Object>>();
4      if (!isLeaf) {
5          children = new ArrayList<HTNode >();
6      }else{
7          children =new Object[n];
```



```
8      }  
9  }
```

每次新建一个节点的时候都会判断当前节点是否为叶子节点，如果不是则在此之上保存叶子节点的列表，如果是叶子节点则在此节点中构造 Hash 表来存储数据。在本文中我们采用的是定长的 Hash 表，不能直接使用 Java 中提供的 HashMap 或者 HashTable。因此根据这个特殊的需求，本文需要编写了自己的 Hash 表的实现。

2) 根节点的构造函数

```
1  public HTNode (boolean isLeaf, boolean isRoot) {  
2      this(isLeaf);  
3      this.isRoot = isRoot;  
4  }
```

根节点从某种程度上来说也是一个叶子节点，和叶子节点唯一的不同是在构造根节点的时候会表明该节点是否为 root，此构造函数在构造整个索引树的时候只会调用一次，详细代码如下：

② 索引树的构造函数

```
1  public HTTree(int m,int n){  
2      if (m < 3 || n<10) {  
3          System.out.print("m 值必须大于 2， n 值必须大于 10");  
4          System.exit(0);  
5      }  
6      this.m = m;  
7      this.n = n;  
8      root = new HTNode (true, true);  
9      head = root;  
10 }
```

索引树的构建需要指定节点的 M 值以及叶子节点中 Hash 表的容量 N，索引树的构造是以根节点的构造开始的，详细代码如下：

③ HT 外部操作类的接口实现如下：

```
1 public class HTUtils {  
2     public void creatIndex(String name){...}  
3     public void deleteIndex(String name){...}  
4     public Object get(Comparable key){...}  
5     public void remove(Comparable key){...}  
6     public void insertOrUpdate(Comparable key, Object obj){...}  
7     public void refactorIndex(String name){...}  
8 }
```

- 1) `creatIndex` 方法用来创建一个索引，其参数是索引名称。
- 2) `deleteIndex` 方法用来删除一个索引，其参数是索引名称。
- 3) `get` 方法用来获取索引的值，其参数是一个 `Comparable` 类型的关键字，即 HT 树的节点关键字。
- 4) `remove` 方法是来删除一个索引项，其参数为 `Comparable` 类型的关键字，即 HT 树的节点关键字。
- 5) `insertOrUpdate` 方法是插入或者修改一个节点索引项，其参数为关键字和索引值。
- 6) `refactorIndex` 方法实现了索引树的重构，索引树的重构可以解决大量的增删后索引的性能降低的问题，也可以在一个已有的数据库上创建索引，其参数为索引名称。

4.3.4 组件之间的耦合

组件之间的耦合是在 `Butler` 类和 `SparkUtils` 类中 `indexOpreate` 方法中实现。

① `Butler` 类

`Butler` 类主要负责判断一项操作是应该在 `HBase` 中执行还是索引系统中执行或者是先在索引系统中执行再到 `HBase` 中执行。比如创建表应该直接在 `HBase` 中执行，创建索引应该在索引系统中执行，在没有 `rowkey` 的读取操作中需要先在索引系统中执行然后在 `HBase` 中执行。并且判断在进行查找和删除等操作时操作条件中是否有 `rowkey`，根据不同的情况进行不同的操作，最大化的节约执行时间。`Butler` 类的示意图如图 4.2 所示：

比如 API 中的“`queryData`”操作在 `Butler` 类中是这样的处理的，首先判断查询条件中是否有 `rowkey`，若有则直接在此处调用 `HBase` 组件中的查询方法，获取并返回数据。如果没有则生成一个 `Json` 操作字符串在索引系统中查找 `rowkey`，然后根据返回的 `rowkey` 在 `HBase` 中进行查询。

比如 API 中“`addData`”操作在 `Butler` 类是这样的处理的，首先写入 `HBase`，

然后根据写入的数据中是否有索引列来判断是否需要写索引，如果需要生成一个Json 操作字符串来写入索引树。其它操作的处理是相类似的。

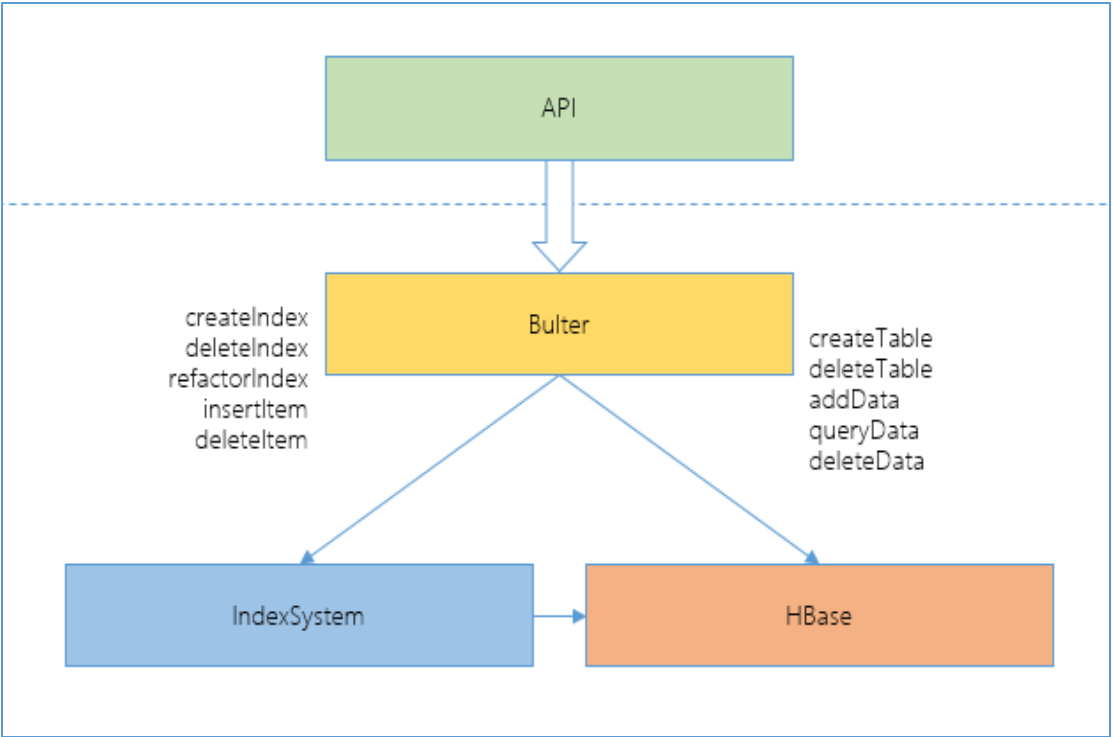


图 4.2 Butler 类示意图

Figure 4.2 The sketch of Butler class

② indexOpreate 方法

SparkUtils 类中 indexOpreate 方法中进行索引树相关操作调用的入口，其参数为包装在 JavaReceiverInputDStream 类中的字符串。根据传入的 json 字符串判断具体操作的类型，然后根据操作类型的不同读出不同的参数列表，从而对索引系统进行相应的操作。

CreateIndex、DeleteIndex、InsertItem 和 DeleteItem 四种操作的具体格式如下所示，其它操作字符串的格式是相似的：

CreateIndex	DeleteIndex
1 {	1 {
2 "indexname" : " table001"	2 "indexname" : " table001"
3 "opreate" : " createIndex"	3 "opreate" : " deleteIndex"
4 "indexs" : [4 }
5 "index0" ,	
6 "index1" ,	
7 "index2"	

```
8      ]
9  }
```

InsertItem	Deleteltem
1 {	1 {
2 “indexname” :” table001”	2 “indexname” :” table001”
3 “opreate” :” insertItem”	3 “opreate” :” deleteltem”
4 “condition” :[4 “condition” :[
5 [5 [
6 “index0” ,	6 “index0” ,
7 [7 [
8 “key1” ,	8 “item0” ,
9 “rowkey1”	9 “item1”
10]	10]
11]	11]
12]	12]
13 }	13 }

每个 Json 操作字符串的第一项都是索引的名称，第二项是具体的操作类型，之后是该操作所需的数据，操作所需的数据会根据操作类型的不同而不同。比如上述的 createIndex 操作，就是创建一个名为” table001” 的索引。其索引列有” index0” 、” index1” 、” index2” 三项。在具体的实现中我们会创建一个 List<HTTree>列表来存储索引，每个索引列为一行数据，具体的索引列是 HT 树。在 InsertItem 操作中，该操作字符串表示在” table001” 索引中的” index0” 这个列中增加了一个索引项，索引的键值关系为” key1” 对应” rowkey1” 。

4.4 API 设计

考虑到我们会提供不同技术方案的 API，所以在写具体的 API 实现之前对要实现的接口做一个大概的规划，统一接口的命名风格。这样做的好处是我们熟悉了一套 API 之后可以很方便的使用其它的 API，因为方法的名称是相同的。基于系统所提供的功能主要设计以下两个类型的 API：索引操作 API、数据操作 API。

4.4.1 索引操作 API

① 创建索引结构

为了优化数据库的查询效率，设置一些查询的关键字，在这些关键字之上建立索引结构。

API 名称：createIndex

参数：Index 类，主要字段为索引名称，索引描述列表包

② 删除索引

删除废弃的索引

API 名称: `deleteIndex`

参数: `Index` 类, 主要字段为索引名称

③ 重构索引

在建立新的索引后, 做一次全表扫描, 根据指定的字段重新生成索引树。

API 名称: `refactorIndex`

参数: `Index` 类型, 主要字段为索引名称

4.4.2 数据操作 API

① 创建数据库

在指定服务器上根据规则创建适用的数据库。

API 名称: `createTable`

参数: 表名称、列族名称列表

② 删除数据库

根据数据库名称删除指定的数据库。

API 名称: `deleteTable`

参数: 表名称

③ 查询数据

根据指定的查询条件在 `HBase` 中查找数据, 并返回查询的结果集。

API 名称: `queryData`

参数: 表名称、查询条件列表

④ 添加数据

将新的数据添加至 `HBase` 数据库中。

API 名称: `addData`

参数: 表名称、行键、列族、标识符、数据

⑤ 删除数据

根据需求删除数据库中存储的数据。

API 名称: `deleteData`

参数: 表名称

4.5 基于 Thrift 的 API 的实现

4.5.1 Thrift 简介

Thrift 最初是 Facebook 内部的一个项目, 后来 Facebook 把 Thrift 提交给了 Apache 基金会作为一个开源项目, Facebook 创造 thrift 是为了解决 Facebook

系统中各组件间大数据量的传输通信以及系统之间语言环境不同需要跨平台的特性^[31]。Thrift 是一个 C/S（服务端和客户端）的架构体系，Thrift 是一个类似 XML-RPC+Java-to-IDL+Serialization Tools=Thrift 的架构，其可以支持多种程序语言，例如：C++，C#，Cocoa，Erlang，Haskell，Java，Ocami，Perl，PHP，Python，Ruby，Smalltalk 等。因此 Thrift 可以作为不同的语言之间二进制形式的高性能通讯中间件，支持数据序列化和多种类型的远程过程调用协议服务。

使用 Thrift 作为程序之间数据传输的工具的首要任务是通过 Thrift 语言定义组件之间数据交换的接口和格式，即定义 Thrift 的 IDL 文件。定义好 IDL 文件之后可以通过 Thrift 工具将其编译为指定语言的代码文件，若修改了传输的接口或者数据接口则需要重新定义和编译 IDL 文件。Thrift 适用于搭建大型数据交换及存储的通用工具，对于大型系统中的内部数据传输相对于 JSON 和 xml 无论在性能、传输大小上有明显的优势^[32]。

4.5.2 Java API 的实现

用 Thrift 实现 Java 接口有几个步骤：

① 编写 Thrift 脚本

首先根据 IDL（The Thrift interface definition language^[33]）编写脚本文件 myIndex.thrift，部分代码如下：

```
1 namespace java com.rambo.myIndex.service.thrift
2 service MyIndexService {
3     void setConfiguration(1:string host),
4     bool createTable(1:string tableName,2: list<string> familys),
5     bool deletetable(1:string tableName),
6     bool createIndex(1:string indexName),
7     bool deleteIndex(1:string indexName),
8     bool refactorIndex(1:string indexName),
9     bool addData(1:string tableName,2:string rowkey,3:string family,4:string
    identifier,5:string data);
10    string queryData(1:string tableName,2:list<string> conditions),
11    bool deleteRow(1:string tableName,2:list<string> rowkeys)
12 }
```

以上定义了常用的几个 Java API 的访问方法

- 1) setConfiguration 方法用来设置 HBase 集群的 zookeeper 管理主机地址。
- 2) createTable 方法用来创建数据库，其参数为 string 类型，主要包含表名称

和列族名称列表。

- 3) `deleteTable` 方法用来删除数据库, 其参数为 `string` 类型, 主要包含表名称。
- 4) `createIndex` 方法用来创建索引, 其参数为 `string` 类型, 主要包含表名称和索引列表。
- 5) `deleteIndex` 方法用来删除索引, 其参数为 `string` 类型, 主要包含表名称。
- 6) `refactorIndex` 方法用来重构索引, 其参数为 `string` 类型, 主要包含表名称。
- 7) `addData` 方法用来给数据库添加一条记录, 其参数为 `string` 类型, `rowkey` 值, 列族名称, 标志位名称, 和数据值。
- 8) `queryData` 方法用来查询数据, 其参数为 `string` 类型, 查询条件列表。
- 9) `deleteRow` 方法用来删除一行数据, 其参数为 `string` 类型以及要删除的 `rowkey` 列表。

② 生成 Java 代码

运行命令生成 Java 代码, 生成最主要的 `MyIndexService.java` 类和其它的一些数据类型类。

③ 实现 Thrift 接口

新建 `MyIndexImpl` 类, 实现 `MyIndexService` 类中的 `Iface` 接口中所有方法, 即根据方法名的不同调用 `Butler` 不同的方法。下面给出 `createTable` 的实现示例, 其它接口的实现略。

```
1 public class MyIndexImpl implements MyIndexService.Iface{
2     private static Butler butler = new Butler();
3     public boolean createTable(String tableName,String[] familys ){
4         return butler.createTable(tableName, families);
5     }
6 }
```

④ 编写 Thrift 服务端

我们分别实现多线程服务端和 `NIO` 服务端, 根据应用场景的不同选用不同的服务端。在并发度不高, 但个体请求吞吐量大的情况下选择使用多线程的服务端; 在并发度很高, 但个体请求吞吐量小的情况下使用 `NIO` 服务端。需要注意的是在客户端和服务端采用的数据传输协议必须一致, 在多线程服务端采用 `TBinaryProtocol` 协议, 在 `NIO` 服务端中采用 `TFramedTransport` 协议。

多线程服务端的实现如下:

```
1 public class MultiThreadsServer{
2     public static final int SERVER_PORT = 8090;
3     public void startServer() {
4         try {
5             System.out.println("MyIndex MultiThreadsServer start ....");
6             TProcessor tprocessor = new MyIndexService.Processor<MyIndexService.Ifce>(new MyIndexImpl());
7             TServerSocket serverTransport = new TServerSocket(SERVER_PORT);
8             TThreadPoolServer.Args ttpsArgs = new TThreadPoolServer.Args(serverTransport);
9             ttpsArgs.processor(tprocessor);
10            ttpsArgs.protocolFactory(new TBinaryProtocol.Factory());
11            TServer server = new TThreadPoolServer(ttpsArgs);
12            server.serve();
13        } catch (Exception e) {
14            System.out.println("Server start error!!!");
15            e.printStackTrace();
16        }
17    }
18 }
```

NIO 服务端的实现如下：

```
1 public class NIOServer{
2     public static final int SERVER_PORT = 8090;
3     public void startServer() {
4         try {
5             System.out.println("MyIndex NIOServer start ....");
6             TProcessor tprocessor = new MyIndexService.Processor<MyIndexService.Ifce>(new MyIndexImpl());
7             TNonblockingServerSocket tnbSocketTransport = new TNonblockingServerSocket(SERVER_PORT);
8             TNonblockingServer.Args tnbArgs = new TNonblockingServer.Args(tnbSocketTransport);
9             tnbArgs.processor(tprocessor);
10            tnbArgs.transportFactory(new TFramedTransport.Factory());
11            tnbArgs.protocolFactory(new TCompactProtocol.Factory());
12            TServer server = new TNonblockingServer(tnbArgs);
13            server.serve();
14        } catch (Exception e) {
15            System.out.println("Server start error!!!");
16        }
17    }
18 }
```



```
16         e.printStackTrace();
17     }
18 }
19 }
```

4.6 Web Service API 的实现

4.6.1 RESTful 简介

2000 年, Roy Fielding 博士在其发表的论文《Architectural Styles and the Design of Network-based Software Architectures》^[34]中提出 REST 的概念, 它的意思是表述性状态转移, 它代表的是一种架构风格, 还代表着一组特征, 在软件开发领域称其为架构约束。与传统的客户端和服务端通信模式的区别是, REST 在 Web 层面上对模型进行了抽象, 总的来说, REST 是一种设计和实现分布式超媒体系统的软件架构风格。REST 具有以下五大设计准则:

- ① 任何事物都可以称为资源。
- ② 每个资源都可以有一个唯一的资源标识符匹配。
- ③ 对这些资源的相关操作可以使用连接器接口进行操作。
- ④ 资源标识是不会发生变化的, 和具体资源操作没有关系。
- ⑤ 对操作的增删改查都应该是无状态的。

由于轻量级以及通过 HTTP 直接传输数据的特性, Web 务的 RESTful 方法已经成为最常见的 REST 的实现方法, 可以使用各种语言实现客户端^[35]。RESTful Web 服务通常可以通过自动客户端或代表用户的应用程序访问。但是, 这种服务的简便性让用户能够与之直接交互, 使用它们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

Spring 是一个得到广泛应用的 Java EE 框架, 在版本 Spring 3.0 以后就增加了 RESTful Web Services 开发的支持。REST 支持被无缝整合到 Spring 的 MVC 层, 它可以很容易应用到使用 Spring 构建的应用中。Spring 及其系列框架虽然是非常强大的, 但在软件的正式开发前存在着很复杂的配置过程。为了解决这一问题 Spring 推出了 Spring Boot 框架。Spring Boot 这个框架在经历了不断的演变之后, 如今已经能够用于开发 Java 微服务了。Boot 是基于 Spring 框架进行开发的, 也继承了 Spring 的成熟性。它通过一些内置的固件封装了底层框架的复杂性, 以帮助使用者进行微服务的开发。

4.6.2 RESTful Web Service 的实现

具体开发步骤如下:

- ① Maven 基本配置

在 Spring Boot 的官方网站复制 Boot Restfull 服务的基本 Maven 配置信息到项目的 pom.xml 文件中。这时 Maven 会自动加载一切所需要的 jar 包。基本配置信息如图 4.3 Spring Boot 基本配置 所示：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5
6    <groupId>org.springframework</groupId>
7    <artifactId>gs-rest-service</artifactId>
8    <version>0.1.0</version>
9    <parent>
10     <groupId>org.springframework.boot</groupId>
11     <artifactId>spring-boot-starter-parent</artifactId>
12     <version>1.3.3.RELEASE</version>
13   </parent>
14   <dependencies>
15     <dependency>
16       <groupId>org.springframework.boot</groupId>
17       <artifactId>spring-boot-starter-web</artifactId>
18     </dependency>
19     <dependency>
20       <groupId>junit</groupId>
21       <artifactId>junit</artifactId>
22       <version>4.12</version>
23       <scope>test</scope>
24     </dependency>
25   </dependencies>
26   <properties>
27     <java.version>1.8</java.version>
28   </properties>
29   <build>
30     <plugins>
31       <plugin>
32         <groupId>org.springframework.boot</groupId>
33         <artifactId>spring-boot-maven-plugin</artifactId>
34       </plugin>
35     </plugins>
36   </build>
37   <repositories>
38     <repository>
39       <id>spring-releases</id>
40       <url>https://repo.spring.io/libs-release</url>
41     </repository>
42   </repositories>
43   <pluginRepositories>
44     <pluginRepository>
45       <id>spring-releases</id>
46       <url>https://repo.spring.io/libs-release</url>
47     </pluginRepository>
48   </pluginRepositories>
49 </project>
50

```

图 4.3 Spring Boot 基本配置

Figure 4.3 The base configuration of Spring Boot

② 编写 MVC 控制类 MyIndexController

在 MyIndexController 类定义之前一行通过注解 `@RestController` 将该类描述为 REST 的路由控制文件。在 MyIndexController 中我们要实现所有的 Web Service API，实现的主要方式是对 Butler 类中各个方法的调用。下面给出 createTable 的实现示例，其它接口的实现略。

```
1  @RestController
```

```
2 public class MyIndexController {
3     private static Butler butler = new Butler();
4     @RequestMapping(value={"/createTable"},method=RequestMethod.POST)
5     public boolean createTable(@RequestBody String tableName,String[] familys ){
6         return butler.createTable(tableName, familys);
7     }
8 }
```

MVC 路由控制的实现类似于 SpringMVC 中路由的实现，需要注意的是我们提交的参数不是基本类型，只能通过 POST 传递参数，所以需要指定方法为 POST 类型。

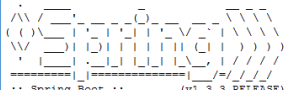
③ 编写 Boot 程序的入口文件 Application 类

在 Application 类定义之前通过注解 `@SpringBootApplication` 将该类描述为一个 Spring Boot 应用程序。Application 文件的内容基本上是固定不变的，任何基于 Boot 开发的服务所开发的软件的 Application 类均是以下这样的。

```
1 @SpringBootApplication
2 public class Application {
3     public static void main(String[] args) {
4         SpringApplication.run(Application.class, args);
5     }
6 }
```

④ 启动 Boot 程序

系统 log 如图 4.4 所示，最后一行显示 `Started Application in...` 则表示我们的 REST API 已经启动，我们可以在浏览器或者程序中通过简单的 http 请求来访问我们的索引数据库了。



```

:: Spring Boot :: (v1.3.3.RELEASE)

2016-03-22 22:22:26.123 INFO 20696 --- [main] com.rambo.SpringBoot.Application : Starting Application on Rambo with PID 20696 (D:\CodeSpace\Java
2016-03-22 22:22:26.132 INFO 20696 --- [main] com.rambo.SpringBoot.Application : No active profile set, falling back to default profiles: default
2016-03-22 22:22:26.301 INFO 20696 --- [main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.Annotation
2016-03-22 22:22:29.581 INFO 20696 --- [main] o.s.b.f.s.DefaultListableBeanFactory : Overriding bean definition for bean 'beanNameViewResolver' with
2016-03-22 22:22:31.727 INFO 20696 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2016-03-22 22:22:31.772 INFO 20696 --- [main] o.apache.catalina.core.StandardService : Starting service Tomcat
2016-03-22 22:22:31.775 INFO 20696 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.0.32
2016-03-22 22:22:32.218 INFO 20696 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2016-03-22 22:22:32.219 INFO 20696 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 5927 ms
2016-03-22 22:22:33.021 INFO 20696 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean : Mapping servlet: 'dispatcherServlet' to [/]
2016-03-22 22:22:33.040 INFO 20696 --- [ost-startStop-1] o.s.b.c.e.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [/]
2016-03-22 22:22:33.041 INFO 20696 --- [ost-startStop-1] o.s.b.c.e.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [/]
2016-03-22 22:22:33.041 INFO 20696 --- [ost-startStop-1] o.s.b.c.e.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [/]
2016-03-22 22:22:33.041 INFO 20696 --- [ost-startStop-1] o.s.b.c.e.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [/]
2016-03-22 22:22:34.169 INFO 20696 --- [main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context
2016-03-22 22:22:34.499 INFO 20696 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "([/greeting])" onto public com.rambo.SpringBoot.Greetin
2016-03-22 22:22:34.532 INFO 20696 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "([/error])" onto public org.springframework.http.Respon
2016-03-22 22:22:34.534 INFO 20696 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "([/error],produces=[text/html])" onto public org.spring
2016-03-22 22:22:34.662 INFO 20696 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.s
2016-03-22 22:22:34.666 INFO 20696 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springfra
2016-03-22 22:22:34.861 INFO 20696 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class o
2016-03-22 22:22:35.245 INFO 20696 --- [main] o.s.j.e.a.AnnotationBeanExporter : Registering beans for JMX exposure on startup
2016-03-22 22:22:35.513 INFO 20696 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2016-03-22 22:22:35.524 INFO 20696 --- [main] com.rambo.SpringBoot.Application : Started Application in 10.287 seconds (JVM running for 11.258)

```

图 4.4 启动成功时的 Boot Log 记录

Figure 4.4 The log record when boot started success

4.7 本章总结

本章主要详细介绍了系统的开发环节。第一小节简单描述了系统开发的基本环境的搭建；第二小节介绍了基于 Spark 的 HBase 索引中间件的设计理念；第三小节描述了中间件的实现过程；第四小节简单介绍了 API 的概要性设计；第五小节是基于 Thrift 的 Java API 的实现的描述；第六小节是基于 Rest Web Service 的 API 的实现的描述。

5 索引系统测试

为了评估文章中所实现系统的性能，分析系统的优点和不足，从而指明未来研究的重点，本章将搭建文章上述的 HBase 索引系统，进行了一系列实验测试，并对实验结果进行比较和分析。

5.1 总体设计

测试数据采用 NCDC (National Climatic Data Center 美国国家气候中心) 提供的数据^[36]。这些数据按行并以 ASCLL 编码存储，其中每一行是一条记录。该存储格式能够支持众多气象数据，其中许多要素可以有选择性地列入收集范围或其数据所需的存储长度是可变的。为了简单起见，我们重点研究一些基本要素，这些要素始终都是有且固定长度的。采集数据如图 5.1 所示，在实际文件中这些数据被整合成了一行并且没有任何分隔符：

1	0067011990999991950051507004+68750+023550FM-12+038299999V0203301N00671220001CN9999999N9+00001+99999999999
2	0043011990999991950051512004+68750+023550FM-12+038299999V0203201N00671220001CN9999999N9+00221+99999999999
3	0043011990999991950051518004+68750+023550FM-12+038299999V0203201N00261220001CN9999999N9-00111+99999999999
4	0043012650999991949032412004+62300+010750FM-12+048599999V0202701N00461220001CN0500001N9+01111+99999999999
5	0043012650999991949032418004+62300+010750FM-12+048599999V0202701N00461220001CN0500001N9+00781+99999999999

图 5.1 数据示例

Figure 5.1 Data example

数据解析如表 5.1 所示：

表 5.1 数据字段解析

Table 5.1 Resolve of data field			
字段编号	位数	实例	含义
01	01-15	00670119909999999999	气象站标识符
02	16-27	195005150700	采集时间
03	28-41	4+68750+023550	经纬度
04	42-46	FM-12	
05	47-51	+0382	海拔（米）
06	52-56	99999	
07	57-60	V020	
08	61-63	330	风向（度）

字段编号	位数	实例	含义
09	64	1	Quality code
10	65	N	
11	66-69	0067	
12	70	1	
13	71-75	22000	采集高度（米）
14	76	1	Quality code
15	77	C	
16	78	N	
17	79-84	999999	可见距离（米）
18	85	9	Quality code
19	86	N	
20	87-92	9+0000	空气温度
21	93	1	Quality code
22	94-99	+99999	露点温度（摄氏度*10）
23	100	9	Quality code
24	101-105	99999	气压（百帕斯卡）

注：数据全被 9 填充则表示该记录缺失。

数据库的设计有以下几个要点：

① 选取气象站标识符、采集时间、经纬度、海拔、风向、采集高度、可见距离、空气温度、露点温度以及气压这 11 项有意义的数据做测试。

② 在入库时我们选取采集时间加经纬度作为每行数据的 rowkey，在遇到数据缺失的时候我们直接存储空值。

③ 在建立数据库的时候我们选取采集时间、经纬度、海拔、风向、空气温度 5 项比较常用的数据在带有索引的 HBase 中建立索引系统。

④ 由于个人使用的计算机的性能和测试数据的数据量均是有限的，我只做了 1000 万条内数量级的数据测试，不做更大规模数据的测试。

下文我们分别在 HBase 和文章中所设计的带有索引的 HBase 进行数据的入库和读取测试。然后对比测试结果，并对结果进行分析。

5.2 实验环境

① 物理机环境

CPU: Intel Core i5 M520 2.40GHz

内存: 6G

硬盘：三星 850 EVO 120G SATA3 固态硬盘

操作系统：Windows 10 Professional

虚拟机：VMware-workstation -12.0.1

② 虚拟机环境

Linux：CentOS-7-x86_64-LiveKDE-1511

CPU：1 核心

内存：3G

5.3 数据入库测试

考虑到计算机的性能问题，我们在写数据的时候每 1 万条记录作为一个最小写入单元，每写入 1w 条数据记录一次写入时间。图 5.2 和图 5.4 是连续写入 1000w 条时 HBase 和带有索引的 HBase 的入库测试时的两个对比图对比图。

① 每万条记录的入库时间对比如图 5.2 所示：

图中线条不是很平滑是因为每次入库的时间不是完全相同的，总体来说数据库的入库耗时在一定的范围内波动。在图中可以看到在本机的硬件条件下在 300 多万条记录内 HBase 和带有索引的 HBase 每万条记录的入库时间相差无几，大概是 0.1 秒多一点。在 300 多万条记录以后由于带有索引系统的 HBase 会维护一个越来越大的索引结构，其插入性能有所下降。在插入最后的 1 万条记录，即 1000 万数量级的时候 HBase 和带有索引的 HBase 的插入时间差约为 0.3 秒。可以看到虽然数据库的性能有所下降，但仍然在一个可接受的范围之内。

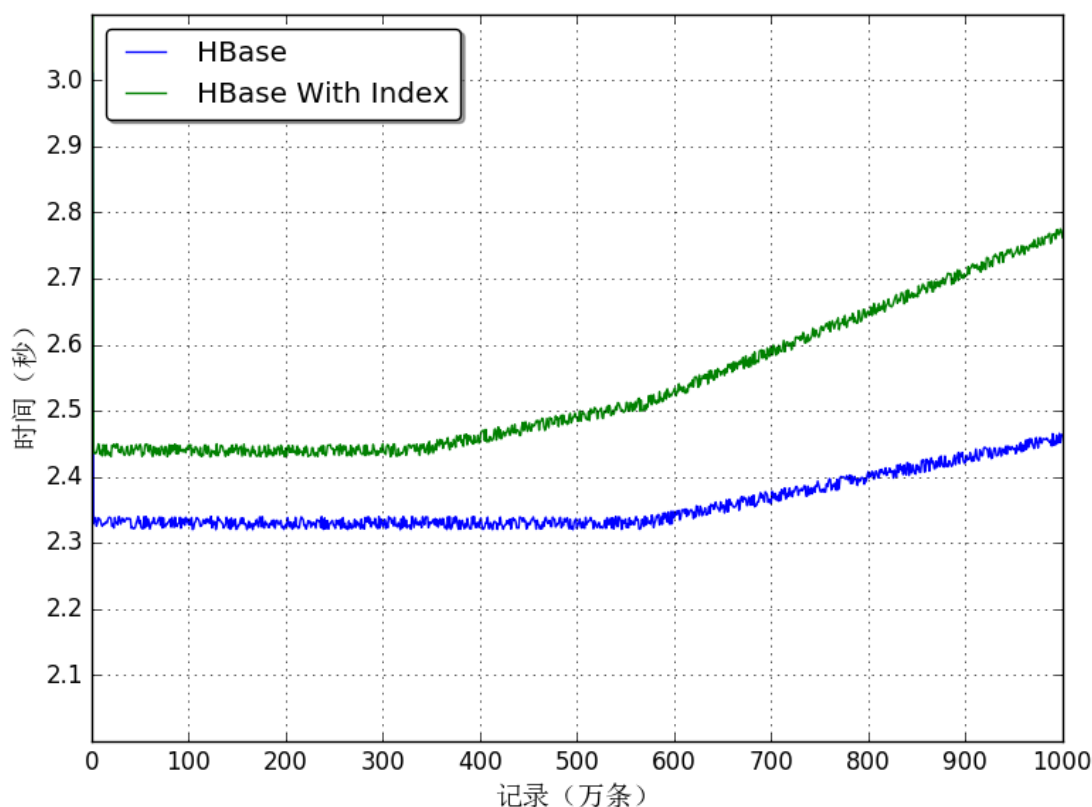


图 5.2 每万条记录写入时间对比

Figure 5.2 The time-consuming compare of each 10,000 records writed

② 总的入库时间对比如图 5.4 所示：

首先我们利用 HBase 自带的系统性能测试工具对数据库的顺序入库性能进行测试，测试结果如图 5.3 所示。对比可以发现在我们的入库测试 100w 条记录的入库时间比 HBase 的性能测试工具给出的结果稍慢，但也在一个正常的范围内。稍慢的原因是我们的测试数据比系统测试的入库数据稍微复杂。

从总的入库时间对比可以看出随着入库记录条数的增加带有索引的 HBase 和 HBase 相比耗时逐渐增加，累积到 1000 万条记录的时候时间差距约为 200 秒。这两百秒就是我们维护 1000 万条记录的索引所需要的时间。


```
HBase Performance Evaluation
Elapsed time in milliseconds=198658
Row count=1048570
File Input Format Counters
Bytes Read=34515
File Output Format Counters
Bytes Written=137
[root@localhost hbase-1.2.0]#
```

图 5.3 HBase 自带性能测试结果

Figure 5.3 The result of HBase performance tool test

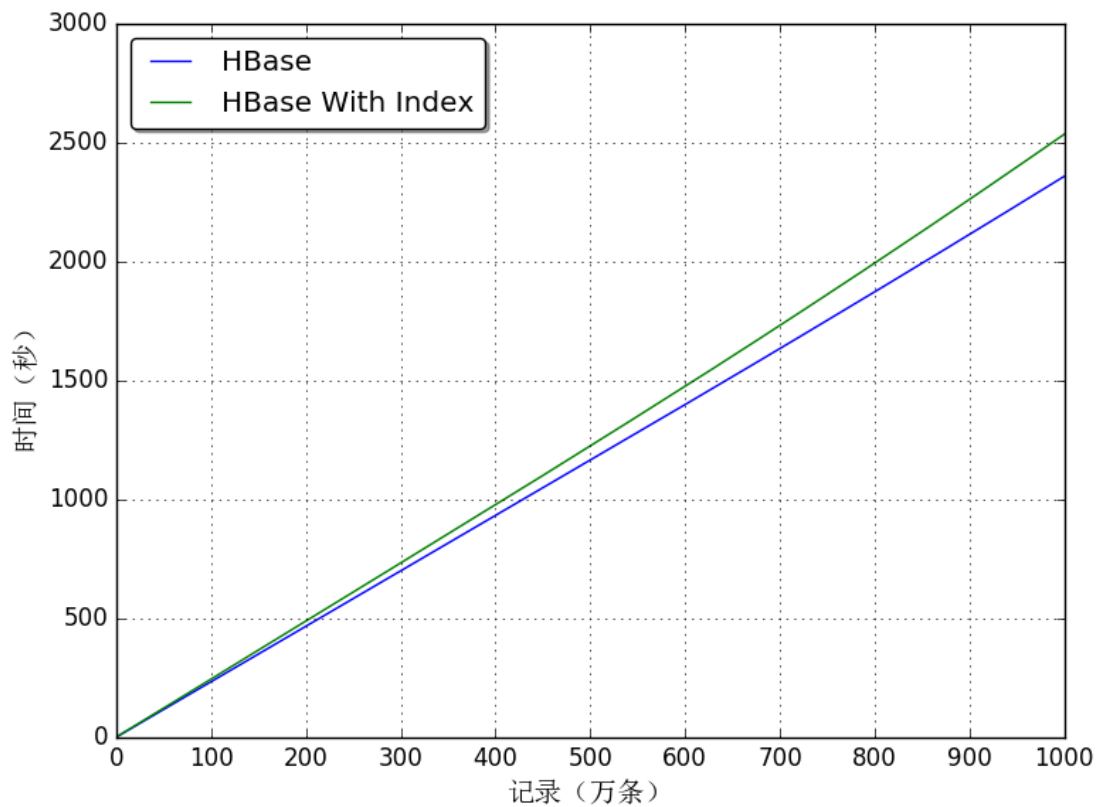


图 5.4 写入总时间对比

Figure 5.4 The time-consuming compare of all records written

5.4 数据读取测试

本文分别在 200w、400w、600w、800w、1000w 这几个数量级下进行对比试验，每个数量级读取十条记录然后求平均值。分别记录了以下四种情况下的查询时间。

A: 有 rowkey 时 HBase 的查询

B: 没 rowkey 时 HBase 的查询

C: 有 rowkey 时有索引的 HBase 的查询

D: 没 rowkey 但有索引时有索引的 HBase 的查询

在上述查询条件中 A 和 C 都能很精准的返回一条记录。B 在查询时只能通过设置过滤条件做全标扫描来实现，有可能在查询结束的时候会返回多条记录。D 在查询的时候先查询索引树，有可能会返回多个 rowkey，同理也会导致返回多条记录。由于 B 和 D 在查询时有可能返回多条记录，不利于在试验对比，因此在试验中将返回结果限定为 1 行。在下文我们将分别对比这四种查询条件在不同数量级时的查询时间的对比。

5.4.1 查询条件 A 和 B 的查询时间对比

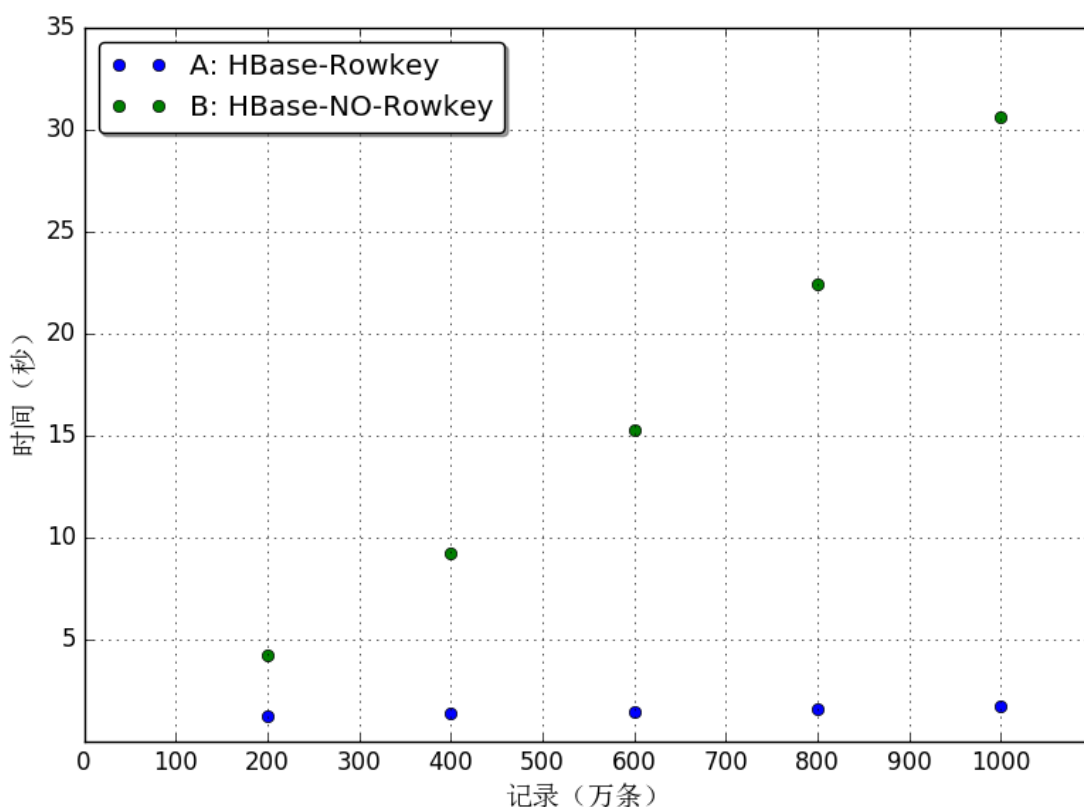


图 5.5 HBase 有无 rowkey 时的读取时间对比

Figure 5.5 The time-consuming comparison when is there rowkey in hbase

从图 5.5 可以看出虽然随着数据规模的增大，查询条件中有 rowkey 的查询也有一定的增长的趋势，但这种增长量很小。反而是查询条件中没有 rowkey 时查询的耗时随着数据规模的增大变得越来越高。由此可以看出在查询 HBase 时 rowkey

的重要性，有 rowkey 时的查询速度要比没 rowkey 时快的太多。在图中只是随机读取一条记录的时间延迟，如果是随机读取多条记录时，HBase 的性能将会表现的特别差。在查询条件没有 rowkey 时 HBase 的性能变得很低并且随着数据规模的扩大性能线性降低的因素是 HBase 在没有 rowkey 的情况下需要进行 scan 操作进行全表扫描来获取数据，scan 操作需要花费大量的时间。

5.4.2 查询条件 A 和 C 的查询时间对比

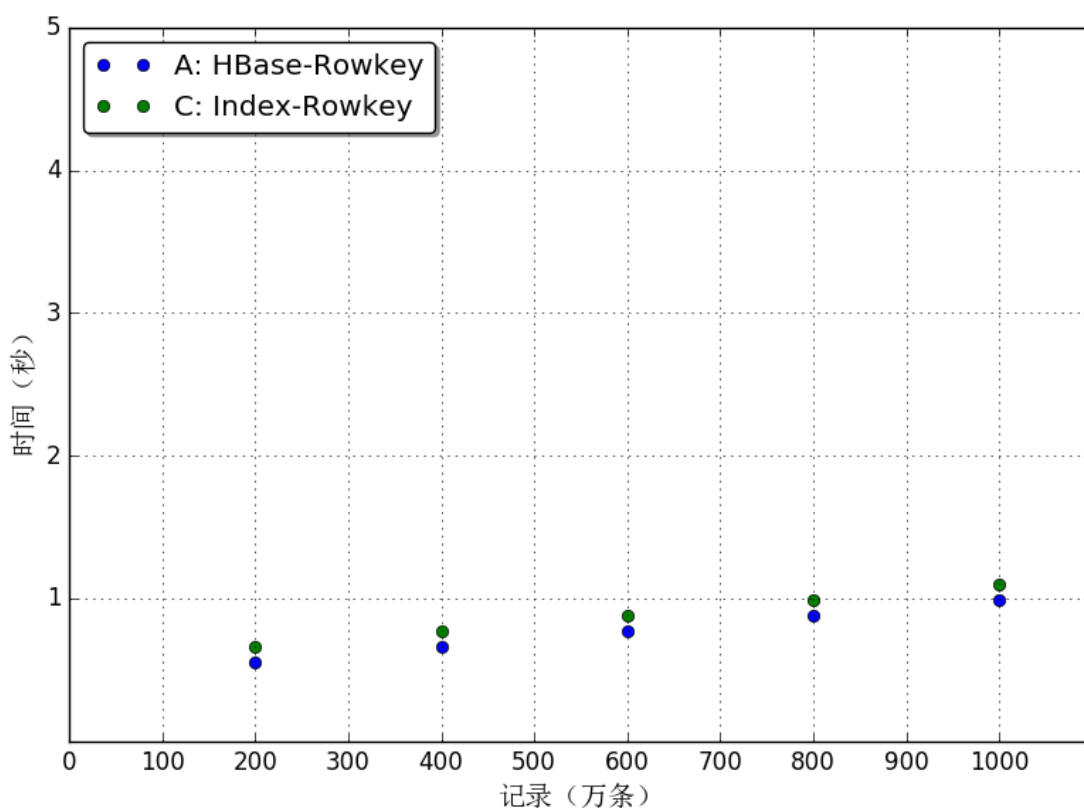


图 5.6 有 rowkey 的 HBase 和有索引的 HBase 的读取时间对比

Figure 5.6 The time-consuming comparison in hbase and hbase with index when is not there rokey

从图 5.6 可以看出在查询条件都有 rowkey 时的查询测试中 HBase 略快于有索引的 HBase，存在这点问题的主要原因是在有索引的 HBase 中所有的查询都要经过系统的 Butler 组建判断查询条件中是否含有 rowkey 的环节，增加了一些额外的时延，且这个延迟也是比较固定的，大概为 0.1s。总体来说并没有对系统的性能带来较大的影响。

5.4.3 查询条件 C 和 D 的查询时间对比

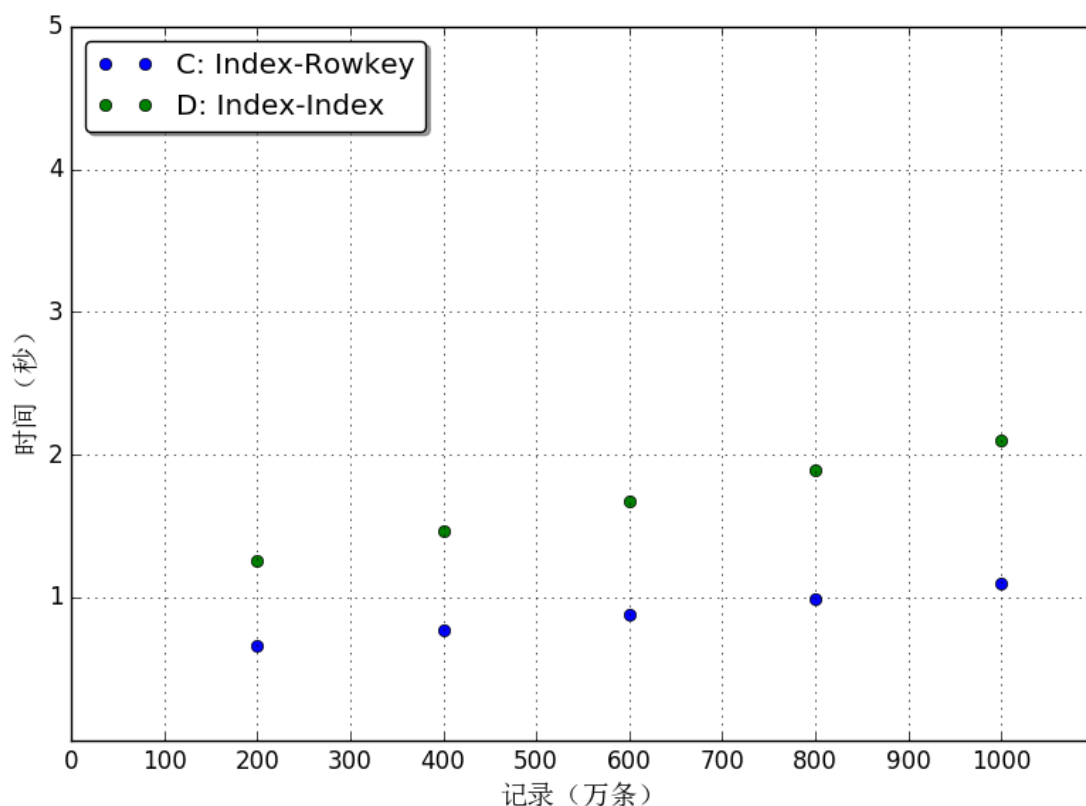


图 5.7 有索引的 HBase 通过 rowkey 查询和索引查询的读取时间对比

Figure 5.7 The time-consuming comparison in hbase with index

图 5.7 是带有索引的 HBase 在查询条件中有 rowkey 和没 rowkey 但有索引项的查询时间的对比，可以看出这两种不同查询条件在不同数量级下的查询时间均有增长，在索引查询时的时间增长略快。在 200w 条数据库时两种查询的时间差约为 0.6 秒，在 1000w 条数据库时的查询时间差约为 1 秒。产生时差的主要原因是带有索引系统的 HBase 在查询条件有索引时会经过 Spark Streaming 去查询索引树，Spark Streaming 是一个流处理框架，有着最小的处理延迟。且随着数据量的增大，索引树的规模也会越来越大，需要的查询时间也会越来越大。但索引树是存储在内存中的，通过 Spark 强大的内存计算能力这个延迟会很低，能够很快的获取到 rowkey。

5.4.4 查询条件 B 和 D 的查询时间比较

从图 5.8 可以看出有索引系统的 HBase 在查询条件有索引时的查询和 HBase

在查询条件没有 `rowkey` 的查询相比是有着很大优势的。虽然随着数据规模的增大有索引的 `HBase` 的查询速度会有一些延迟，但和 `HBase` 下的查询的延迟相比几乎可以忽略不计。

本文的主要目的是解决 `HBase` 在没有 `rowkey` 时或者直接对非 `rowkey` 列时查询的性能问题。从上图我们可以看出我们较好的完成了这一目标。

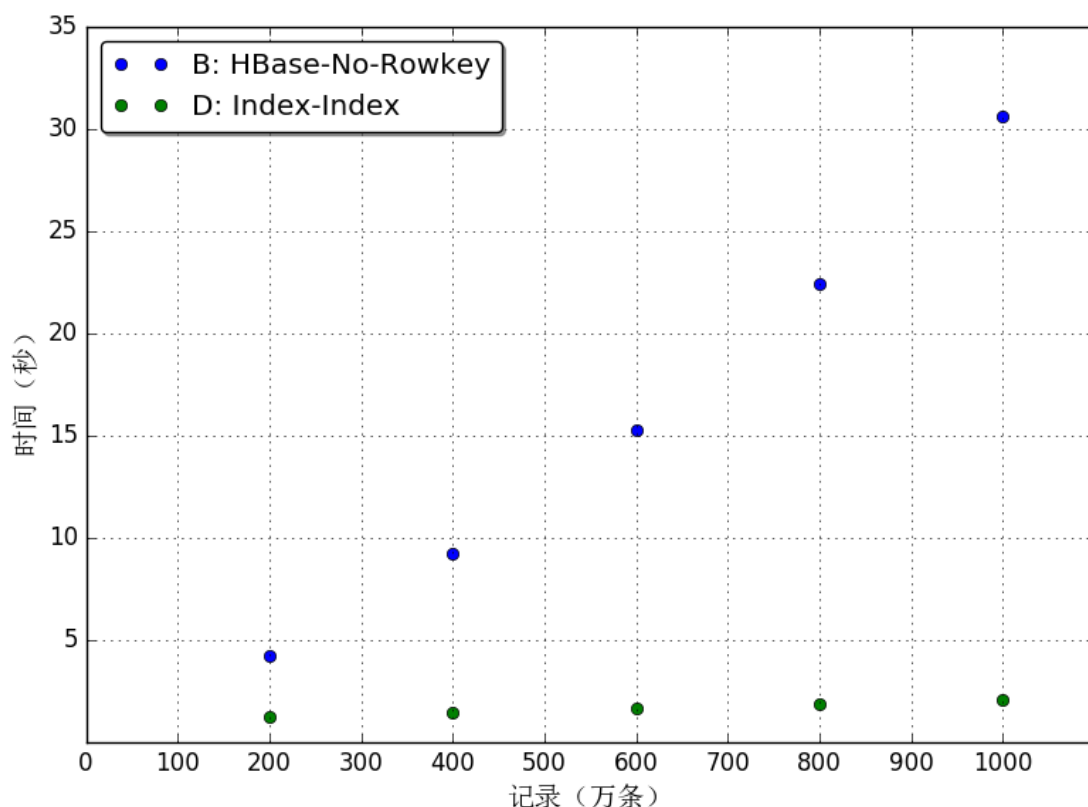


图 5.8 HBase 在没有 `rowkey` 和有索引的 HBase 在没有 `rowkey` 但有索引时查询时间对比

Figure 5.8 The time-consuming compare in hbase between hbase with index when there are rowkey

5.4.5 查询条件 ABCD 的综合比较

从图 5.9 可以更清楚的看到各种情况下的查询效率的对比。带有索引的 `HBase` 的查询性能受查询条件的影响并不是很大，在各种情况下均有接近 `HBase` 直接使用 `rowkey` 查询时的性能。

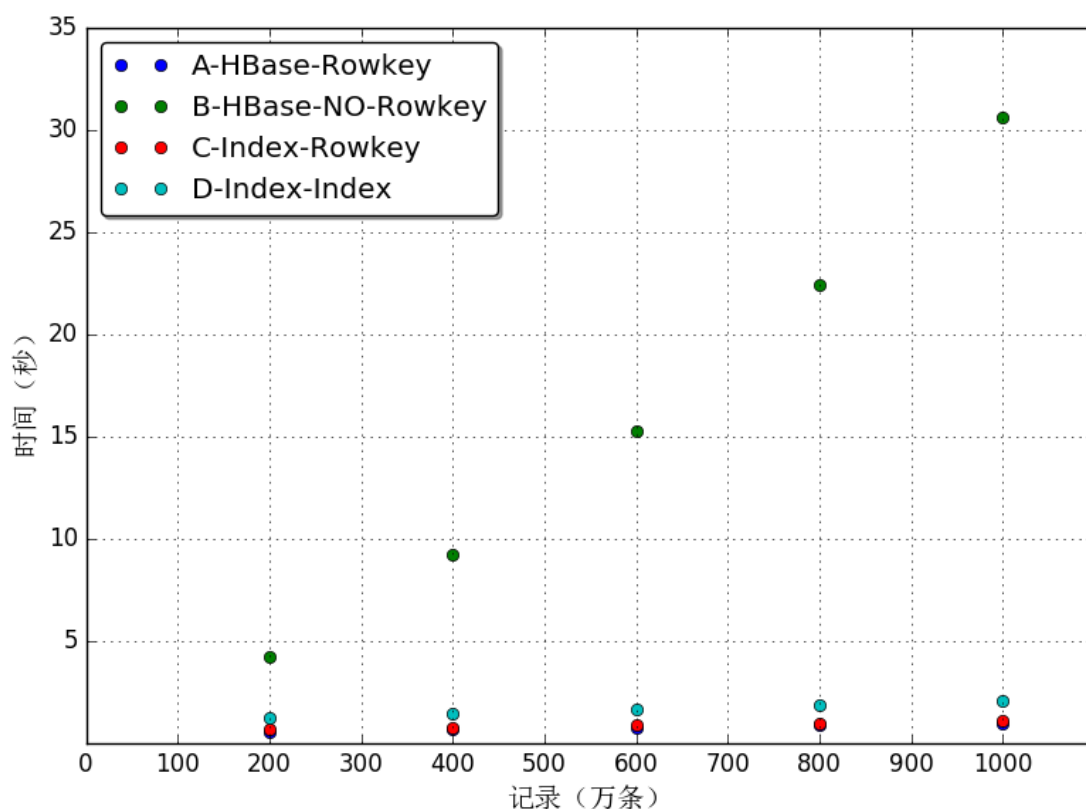


图 5.9 综合对比

Figure 5.9 The time-consuming compare of very condition

5.5 本章小结

本章首先简单介绍了我们测试数据库的数据集，并选取了其中一部分具有明显含义的数据来测试我们的数据库。数据库的测试主要分为入库测试和读取测试量大模块。因为在入库时比 HBase 多一个索引构建的环节，所以数据的入库速度略慢于 HBase。在数据库的读取环节来看读取条件中有 rowkey 的情况下我们的系统读取速度略慢于 HBase，但速度直接的差距是非常小的。读取条件中没有 rowkey 但有索引时我们的系统查询速度比 HBase 快很多。从上述结论可以看出本文所做的一些工作还是有一定的效果的。

6 总结与展望

6.1 论文总结

相对传统的数据库来说, HBase 在海量数据库的存储方面来说具有更好的存储性能和集群的可扩充性。但 HBase 的设计目标只是用来存储数据, 其数据访问的性能、易用性和传统的数据库相比具有一定的劣势。本文中我们主要在 HBase 数据库访问性这一方面能做了一些研究和改进。我们采用了建立类似传统数据库中的非主键索引的方法来提高 HBase 数据库在复杂查询条件下的可访问性。在建立索引的过程中我们借助 Spark 的内存计算功能设计了一个存储在内存中的二级索引来建立和存储非 rowkey 的索引。在一定程度上改善了 HBase 数据访问的性能。在完成本文的过程中做了一些学习和研究的工作, 主要有一下几个方面。

① 基础技术的研究

在本文中我们剖析了 HBase 的基本原理, 主要有 HBase 的逻辑模型, HBase 的物理模型和 HBase 的检索原理。并在此基础上论证了 HBase 的一些不足。在本文中我们研究了索引技术的基本原理和一些基础索引算法, 主要有 Hash 索引、AVL 树、B 树、B+树、T 树和 HT 树等。并详细解释了 HT 树的基本原理和 HT 树的常用操作的实现。在本文中我们研究了 HBase 的几个索引设计架构, 主要有二级索引架构和线性化索引架构, 并详细剖析了这些架构的优缺点。同时也简单介绍了两个完整的索引的设计方案, 主要有中移动的 HugeTable 方案和华为 HBase 索引方案。

② 设计 HBase 索引系统

在研究了常用的索引算法和 HBase 索引设计方案的基础之上我们提出了自己的索引设计方案, 在本文中对此方案做了详细的描述。我们的设计方案主要是基于 Spark 快速的内存计算功能实现基于内存 HBase 索引系统, 索引算法选择了文章详细描述 HT 树。

③ 完成 HBase 索引系统

本文还详细介绍了我们索引系统的具体实现的过程, 主要是系统的一些逻辑关系和类之间的耦合关系。为了便于索引数据库的访问, 在 HBase 索引系统之上提供了基于 Thrift 的 Java API 和基于 REST Web Service 的 API, 文章中我们也详细的介绍了这些接口的实现过程。

④ 系统的测试

最后做了一些测试性的实验, 主要从数据的存储和读取两个方面做了有索引系统的 HBase 和基本的 HBase 的对比, 从实验的结果来看我们的索引系统在一定

程度上提高了 HBase 的访问性能。

6.2 今后展望

随着大数据技术的发展和数据量的累积,数据存储软件逐渐的由 SQL Server、Oracle、Mysql 等传统软件向 HBase、Cassandra、MongoDB 等分布式数据库软件过度越来越明显。同时,我们也期望着分布式数据库存储软件的功能越来越完善。可以提供传统结构化数据库里很常用的一些功能,比如复杂条件下的查询、事物等功能。在本文中我们探索性的做了一些 HBase 非主键索引的研究。但由于本人研究水平和时间按有限,论文里的工作只是一个初步的研究。只完成了最基本的功能,没有做更多的功能扩展。在现有研究的基础上还可以做很多其他的研究,主要总结如下:

① 提供数据管理界面

现有的数据管理主要是通过 Java 语言和 RESTful 的接口实现的,在数据库管理时有一定的复杂性。提供命令行或者图形化的数据库管理界面能降低数据库的使用难度。

② 提供多线程的系统

本文目前实现的系统只能使用在单线程的条件下,在多线程运行时会有一些问题。为了实现系统的高性能和硬件资源的高利用率以后可以在此基础上实现多线程的版本。

③ 提供多表之间的查询

虽然 HBase 不提倡在一个数据库里建立多张表,但有时候实际应用中处于各方面的考虑也会有多表查询的需求,为了满足数据库在更多业务下的适用性,我们可以在现有的基础之上提供多表查询的功能。

④ 提供 SQL 语句的解析

和 HBase 相比,大多数人更为熟悉用传统的 SQL 语句来操作数据库,以后以提供一个 SQL 语句的解析组件来兼容传统的 SQL 语句,让更多的开发人员跟容易的切换到我们的数据库来。

⑤ 索引算法的优化

本文中选取的 HT 树的索引算法在内存的空间利用率方面不是很好,查询和维护效率也有提升的空间,我们可以进一步优化索引算法来降低索引数据库的内存占用率提高数据库的维护性能。

⑥ 引入权限系统

本文的研究中没有考虑到数据库的用户和表之间权限问题,在以后的研究中应该引入权限系统,实现多用户环境下用户数据的安全。

致 谢

在不经意之间这一段求学之路即将画上句号，好多事还没做完就要踏上另一段旅途，不得不让人感慨岁月如流水一般一去不复还。回首来路，虽谈不上虚度光阴，但多少浪费了一些时间，不免有几分自责。这几年有些遗憾但更多的是收获；有些惆怅但更多的是欢乐；虽有过彷徨，但依旧坚毅这向着自己的目标前进。虽然这一段学习生涯即将结束，但这并不意味着往后不用再学习的，在今后的工作生活中我依然会努力学习心得知识，不辜负师长得嘱托。在毕业论文即将成文之际，我在此对给予我帮助和关心的老师表示由衷的感谢。

首先要感谢我的导师刘卫宁教授，感谢刘老师在本文写作过程中给予我的悉心指导和帮助。在我三年的读研生活中，刘老师渊博的学识、广阔的研究视野和高屋建瓴的学术造诣使得我学术研究能力不断的提高，也让我对自己所学和即将从事的行业有了全新的认识。同时，刘老师严谨务实、一丝不苟的治学态度也深深地影响了我，豁达的人生态度、睿智的人格魅力更是让我受益匪浅。很开心在以后的工作中我还有继续向刘老师学习的机会。另外我也要感谢郑林江老师这几年对我的一些指导，让我学习到了很多学术以及学术之外的事。

非常感谢冯琦森、徐仁和、王海涛、邓晨晨、廖理和赵欣以及其他的同门师兄，是你们陪我探讨了许多的学术问题，也是你们给了我很多学术之余的生活中的快乐，使我的学习生活更加的丰富多彩。

非常感谢崔晓冬、宋元隆和赵振崇等好友，是你们在我迷茫是给予我不断的鼓励，让我更加坚定了努力的信念。

非常感谢我的父母和其他家人这么多年对我的支持，是你们给了我继续努力的动力。特别感谢我的女朋友赵晓婕，这么多年一直不离不弃默默的陪着我，从一个天真烂漫的女孩变成了一个依旧天真烂漫的大龄女青年，我相信以后我们会更加幸福。

最后我再次感谢大家！感谢所有关心我、帮助过我的人们！祝大家天天快乐、永远幸福！

卢文博

二〇一六年四月

参考文献

- [1] 盛扬燕,周涛.大数据时代[M]:浙江人民出版社,2012:11.
- [2] EMC 数字宇宙研究报告 2014 版[EB/OL].
<http://wenku.baidu.com/view/5b885f7efc4ffe473268ab40.html>.2014
- [3] 马友忠,孟小峰.云数据管理索引技术研究[J].软件学报.2015,26(1):145-166.
- [4] 王子健.海量多结构数据智能检索中的存取方法研究[D].武汉:华中科技大学.2013.
- [5] Ghemawat S, Gobioff H, Leung ST. The google file system[J]. In: Proc. of the 19th ACM Symp. on Operating Systems Principles. New York: ACM, 2003. 29-43.
- [6] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters[J]. In: Brewer EA, Chen P, eds. Proc. of the 6th Conf. on Symp. on Operating Systems Design & Implementation. New York: ACM, 2004. 137-149.
- [7] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data[J]. ACM Trans. on Computer Systems, 2008,26(2):1-26.
- [8] 李永春,丁华福.Lucene 的全文检索的研究与应用[J].2010(02):12-15.
- [9] 陈新鹏.基于 HBase 的数据生成与索引方法的研究[D].北京:北京邮电大学,2013.12.
- [10] 卓海艺.基于的海量数据实时查询系统设计与实现[D]. 北京:北京邮电大学, 2013.02.
- [11] 李卫江,侯国.Lucene 应用研究[J].2011(16): 291-291
- [12] 马会.基于 Nutch 和 Solr 的企业级搜索引擎的研究与实现[D].西安:西安电子科技大学, 2014.
- [13] 葛微,罗圣美. HiBase:一种基于分层式索引的高效 HBase 查询技术与系统 [J].2016(01):140-153.
- [14] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, Amr El Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services[J].
- [15] Nishimura S, Das S, Agrawal D, Abbadi AE. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services[J]. In: Proc. of the 11th Int'l Conf. on Mobile Data Management. Washington: IEEE, 2011. 7-16.
- [16] 闫伟,陈满林,白云鹤,班海涛. 基于 Hadoop 的校园网盘的设计与实现[J]. 科技创新与应用.2015(16).
- [17] Spark 官方网站.<http://spark.apache.org/>. [EB/OL].
- [18] 唐向红.移动实时数据库数据广播与索引技术[D].武汉:华中科技大学.2006.

- [19] 何水霞.基于 B-Tree 索引和 BerkeleyDB 的中文词库的设计和实现案[D].2009.
- [20] 李萌萌,赵勇. 一种支持活动标记的访问控制标识方法[J]. 计算机工程与应用.2012.03.
- [21] 肖富平.内存数据库存储及索引技术研究[D].重庆:重庆大学.2009.
- [22] 鲍培明.基于 HBase 的空间数据分布式存储和并行查询算法研究[D].2014.
- [23] Zou YQ, Liu J, Wang SC, Zha L, Xu ZW. CCIndex: A complemental clustering index on distributed ordered tables formulti-dimensional range queries[J]. In: Proc. of the 7th IFIP Int'l Conf. on Network and Parallel Computing. Berlin: Springer-Verlag,2010. 247-261.
- [24] Agrawal P, Silberstein A, Cooper BF. Asynchronous view maintenance for VLSD databases[J]. In: Proc. of the 2009 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2009. 179-192.
- [25] 查礼.基于 Hadoop 的大数据计算技术[J].科研信息化技术与应用,2012 (6):26-33.
- [26] 袁培森,皮德常.用于内存数据库的 Hash 索引的设计与实现[J].计算机工程,2007,33(18):69-71.
- [27] T.J.Lehman, M.J.Carey,A Study of Index Structures for Main Memory Database Management Systems[J], in Proceedings 12th Int. Conf. On Very Large Database, Kyoto, August 1986, pp. 294-303.
- [28] Ailamaki, A. DBMSs on A Modern Processor: Where Does Time Go[C]Proceedings of the 25th VLDB Conference.Edinburgh, UK: [s. n.], 1999.
- [29] 万鹏. OSS 中嵌入式内存数据库研究[D].哈尔滨:哈尔滨工业大学.2008.
- [30] 吴泉源. 网络计算中间件[J]. 软件学报, 2013,24(1):67-76.
- [31] 刘书楠. Thrift 的入门简介[J]. 青年与社会, 2013. VOL. 511. NO. 1.
- [32] Thrift 官方网站[EB/OL].<http://thrift.apache.org/>.2016
- [33] Thrift 接口定义规范[EB/OL].<http://thrift.apache.org/docs/idl/>.2016
- [34] Fielding R T. Architectural styles and the design of network-based software architectures[D]. University of California, Irvine, 2000.
- [35] 龚向阳.移动环境下图片多维组织与管理系统服务器端的研究与实现[D].北京:北京邮电大学.2010.
- [36] 美国国家气候数据中心.[http:// www.ncdc.noaa.gov/](http://www.ncdc.noaa.gov/). [EB/OL].