2018 届硕士专业学位研究生学位论文（全日制研究生）

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51151500113

華東師範大學

East China Normal University

硕士专业学位论文

MASTRER'S DEGREE THESIS (PROFESSIONAL)

论文题目：面向读写分离架构的分布式数据库连接算子设计与实现

院　　系：计算机科学与软件工程

专　　业：软件工程

研究方向：分布式数据库

指导教师：张召　副教授

学位申请人：隆飞

2017 年 11 月 13 日

Dissertation for professional master's degree In 2018

University Code: 10269

Student ID: 51151500113

East China Normal University

Title: Design and Implementation of A Join Operator In Reading/Writing Separation Database System

Department:	**School of Computer Science and Software Engineering**
Major:	**Software Engineering**
Research direction:	**Distributed Databases**
Supervisor:	**A/Prof Zhang Zhao**
Candidate:	**Long Fei**

November，2017

华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《面向读写分离架构的分布式数据库连接算子设计与实现》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名：隆飞　　日期：2017 年 11 月 28 日

华东师范大学学位论文著作权使用声明

《面向读写分离架构的分布式数据库连接算子设计与实现》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的著作权归本人所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和学校指定的相关机构送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

（　）1.经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文*，于　　年　　月　　日解密，解密后适用上述授权。

（√）2.不保密，适用上述授权。

导师签名　　本人签名　隆飞

2017 年 11 月 28 日

* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

隆 飞 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
周烜	教授	华东师范大学	主席
董启文	副教授	华东师范大学	
王晔	高级工程师	中国人民解放军第二军医大学	

摘　要

近年来，一些“现象级”事件，类似于淘宝的“双十一”，微博的“鹿晗事件”等，都造成了数据的“井喷”。这些事件会在短时间内产生大量的数据并写入到数据库，这就要求数据库拥有十分强大的写入能力。基于日志结构合并树（The Log-Structured Merge Tree，简称 LSM）引擎的读写分离架构是一种新型的分布式系统架构，具有良好的写入性能，可以很好的处理写密集型业务场景。但是，利用这种架构来提升写性能是以牺牲部分查询性能为前提的。

Join 算子是复杂查询中最重要的算子，也是影响查询性能最关键的算子。为了改善上述新型架构的查询性能，本文从 Join 算子入手，结合了传统的查询优化技术，提出了一种面向读写分离体系的自优化 Join 算子的实现方法。此 Join 算子将数据库中许多其他算子包含其中，并提供一套基于规则和统计信息的选择策略。此外，本文还针对这种特殊的架构实现了 ParallelScan，BtwFilter 等一系列其他算子，用于支撑 Join 算子的自优化过程。

本文主要贡献现总结如下：

1. 设计了一种面向读写分离新型体系的自优化 Join 算子，可以很好地改善此种架构下分布式数据库复杂查询的性能问题
2. 提出了面向分布式数据库的统计信息的收集方法，并将其用于自优化的选择策略当中
3. 提出了 ParallelScan 等一系列面向新型体系的物理算子的实现方法，为自优化提供更多地可选性

本文在一款读写分离分布式数据库系统 Cedar[2]中实现，并设计实施了针对性实验验证了本文提出的自优化 Join 算子优化过程的有效性。

关键词：读写分离架构、LSM 引擎、统计信息、并行扫描算子、自优化连接算子

ABSTRACT

In recent years, some "phenomenal" event, similar to Taobao's "Double 11 event", micro-blog's "Luhan incident", have caused the data "blowout". These events generate large amounts of data and write them into database in a short period of time, which requires the database to have very powerful write capability. Read / write separation architecture based on The Log-Structured Merge Tree engine is a new distributed system architecture, which has good writing performance and can handle write intensive business scenarios very well. However, the use of this architecture to improve write performance is based on sacrificing partial query performance.

Join operator is the most important operator in complex query, and it is also the most important operator that affects query performance. In order to improve the query performance of the new architecture, this thesis starts with the Join operator and combines the traditional query optimization technology, and proposes a method to implement the self-tuning Join operator for the read write separation system. The Join operator contains many other operators, and provides a set of selection policies based on rules and statistical information. In addition, a series of other operators such as ParallelScan, BtwFilter and so on are implemented to support the self-tuning process of the Join operators.

The main contributions of this paper are summarized as follows:

1. a self-tuning Join operator for reading and writing separation system is designed, which can improve the performance of complex query in distributed database under the new architecture
2. the method of collecting statistical information for distributed database is proposed, and it is used in the self-tuning selection strategy
3. A series of physical operators, such as ParallelScan, are proposed to provide more alternatives for self-tuning

We have implemented the technique mentioned in this thesis in Cedar[2], a read-write distributed database system designed, and implemented a targeted experiment to verify the effectiveness of the self-tuning Join operator optimization process.

Keywords: read write separation architecture, LSM engine, statistical information, parallel scan operator, self-tunnig join operator.

目 录

第一章 绪论

1.1 研究背景及意义

随着科技进步，智能手机、掌上电脑以及其他智能穿戴等高科技设备越来越普及，人们在使用这些工具辅助生产生活时产生了大量的数据。数据爆炸式增长不仅仅给数据的管理方式带来了严峻的考验，也对检索有价值信息的效率提出了更高要求。数据库的组织形式也由传统的集中式数据库渐渐向分布式架构演变，以更好地处理这种海量数据场景。但是，无论数据库技术怎么发展，查询始终是数据库领域中最重要的议题之一。

1996 年 Patrick O'Neil, Edward Cheng 等提出 LSM-Tree [3]存储架构。这种架构的核心思想就是牺牲数据的一些读性能，对写入变更操作进行延迟及批量处理，以提供高效的写入性能。最近几年，应用逐渐由读密集型向写密集型倾斜[1]，也使得 NoSQL 数据库，如 Google 的 BigTable[4]，Facebook 的 Cassandra[5]，以及开源社区的 HBase[6]等流行起来，LSM-Tree 架构也逐渐走进人们的视线。

另外一方面，读写分离架构是一种新型的分布式系统架构，其含义是在数据库系统中将提供读写的节点分离开，即一些节点只响应读请求，另外一些节点只响应写请求。在这种读写分离架构下，数据被分为基线数据和增量数据，数据的读取操作需要获取写节点上的增量数据，其流程更加复杂。另外，这种架构独有的每日合并模块：将增量数据冻结合并成为基线数据，也给查询优化工作带来新的机遇和挑战。

基于 LSM-Tree 架构的读写分离式分布式数据库系统固然拥有很高的写入性能，非常优秀的扩展能力，能够很好得应对数据的爆炸式增长，但是，利用这种

架构的分布式数据库系统在海量数据中获取有价值信息的效率却变得很低。另外，相对于传统的集中式数据库，分布式数据库系统查询还可能涉及到网络上的多个节点，特别是在表连接的场景下，跨网络传输数据更为普遍。

为了解决这种架构下的数据的读取性能，很多学者以及工业界的实践者也从不同角度对这些问题进行了很多思考。

i. 从架构本身

论文[1]中结合了 LSM-Tree 和 B-Tree 的优势，使用布隆过滤器保证读取操作在读到一个版本的数据之后能够及时返回。提出了一种新的“Spring and Gear”合并调度器，它通过确保树的每个级别的合并稳定地进行，而不采用降低读取性能的技术来实现。

ii. 从硬件角度

近些年来，硬件的发展十分迅速：固态硬盘逐渐取代了机械硬盘，前者能够提供更好的读取性能；新兴的非易失存储器（Non-Volatile Memory，NVM)[7，8，9]能够提供可靠的内存存储；GPU[10，11，12]的发展也为高吞吐读取操作提供了硬件基础。基于这些新型硬件的一些研究工作也在如火如荼得展开，这些工作从另外一个方面促进了数据库的发展。

iii. 从软件层面

在软件层面，基于多核硬件基础，将数据库中的算子并行起来可以大大缩短数据的访取流程。分布式的另外一个特点即查询很大可能会跨节点，通过使查询场地本地化可以大大减少网络上数据传输的量。论文[13]是软件层面的一项优化工作，可以弥补硬件的不足，降低数据库系统的成本。

本文也尝试从软件层面出发切实提升读写分离架构下分布式数据库的查询性能：提出了一种可能的此种架构下的统计信息的收集方式；实现了并行扫描（ParallelScan）算子，充分利用多核处理器的优势，将扫描（Scan）操作并行起来。最后，以收集的统计信息为基础实现了一个可以自我优化的连接（Self-tuning Join）算子，此算子可以调节查询使用的算子以及其执行顺序来获得一个较优的执行性能。

1.2 本文主要工作

随着越来越多的业务从传统的集中式数据库迁移到基于 LSM-Tree 架构的读写分离分布式数据库，导致了这种新型架构在数据查询方面的性能表现与用户需求之间的矛盾不断激化。特别的，在账务系统中，这种数据查询性能往往指的是表与表之间的连接操作的性能。据统计，某银行历史库业务的所有查询（Query）当中，连接（Join）操作占了 88.7%[14]。为了使传统业务能够顺利完成数据管理方式的转型，优化 LSM 数据库的查询性能迫在眉睫，关键是要优化表 Join 的性能。

下文介绍的 Cedar 数据库是一款典型的 LSM 架构的读写分离分布式数据库系统，也是本文工作的原型系统。本文围绕优化 Cedar 数据库中的 Join 性能展开了一系列工作：

首先，本文实现了分布式数据库系统中统计信息的收集和维护工作：在读写分离的架构下，增量数据必须依靠合并模块冻结成为基线数据。利用合并模块可以达到收集统计信息的目的。但是，这样做会产生一个问题，就是在两次合并期间如果数据发生了很大的变化，会导致统计信息的失真。这个问题会在后面详细讨论。在有了数据的统计信息之后，其他基于代价的一些优化工作才能够展开。

其次，本文的另一项工作是完善 Cedar 数据库中的算子。ParallelScan 算子是 Scan 算子的并行版本。在实际业务当中，我们发现某些情况下，如图 1.1 所示，黑色方块部分代表实际访问数据所花费的时间，箭头上的数据表示网络传输的时间，在整个取数据的流程当中网络的时间远远小于取数据实际花费的时间。ParallelScan 算子充分利用了多核架构的优势，并且结合了读写分离体系下副本提供可读性的特点将 Scan 算子并行执行，加速了数据的读取速度。在连接当中，不同的数据模式会严重影响到连接的执行效率。本文结合了不同的数据模式设计实现了 InFilter 和 BtwFilter 算子：InFilter 算子基于 In 表达式，适用于左表数据量较小的场景，BtwFilter 算子基于 Between And 表达式，适用于左表对右表过滤

性较好，并且左表的范围较大的情况。除此之外，本文还设计并实现了 IndexScan 算子，向用户提供索引查询，能够大大加速取数据的流程。

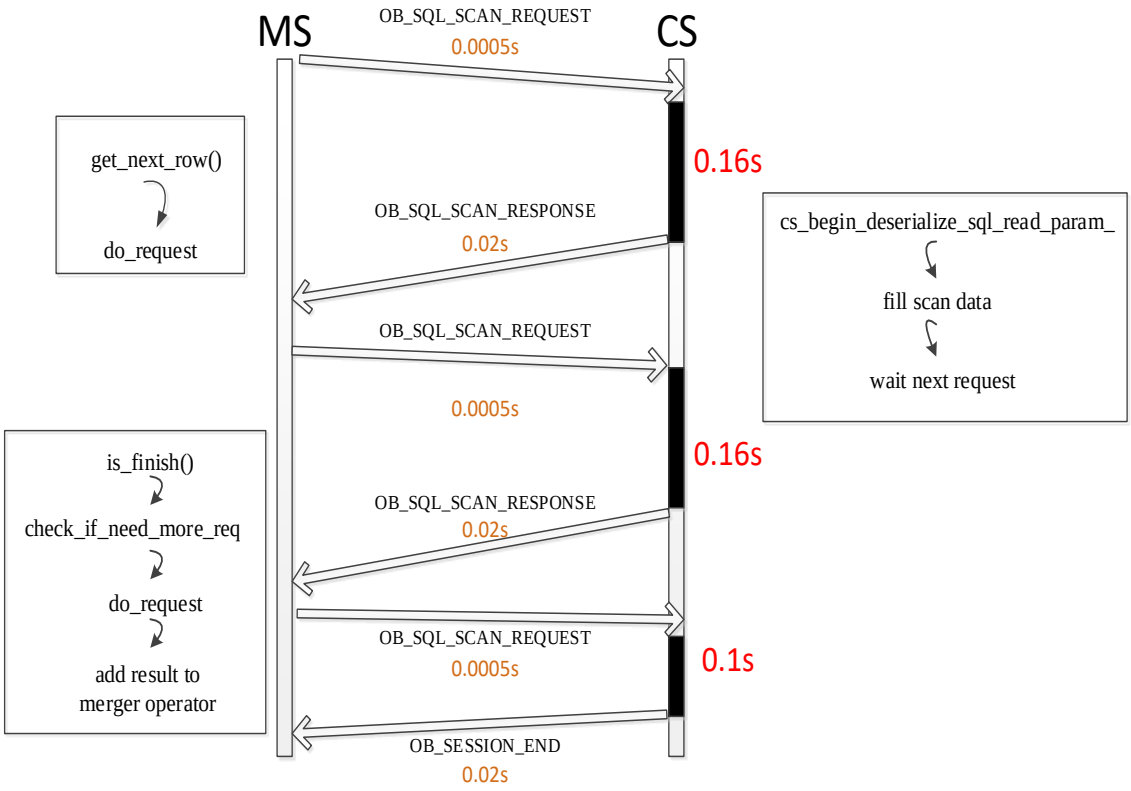

图 1.1 取数据网络时间比例图

接着，本文结合了传统数据库优化的技术，设计并实现了自优化的连接算子。自优化连接算子实现的难点在于产生可能的查询计划，并对每一个查询计划的性能进行预估，最终生成一个较优的查询树。目前，自优化算子能够自我调节实际使用的算子，Join 表的顺序以及是否加载多 SemiJoin 模块等等。而在这个过程中，统计信息所起到的作用至关重要。

最后，本文设计了一系列实验，从各个角度验证了自优化算子的可行性以及性能。同时，也对参与自优化过程的各个决策算法进行了功能性验证。

1.3 本文结构

如图 1.2 所示，本文在基于 LSM 引擎的读写分离分布式数据库系统下实现了一款自优化的连接算子。该算子主要包括两个部分：基础设施和选择策略。自优化的过程是 Join 算子根据用户 Query 输出最优执行计划的过程，基础设施为自优化算子进行自我优化时，提供更多的可选项。基础设施越丰富，意味着选择性更多，能够自优化的场景越广。对于每个算子，都匹配一套对应的基于规则或者代价的选择策略。

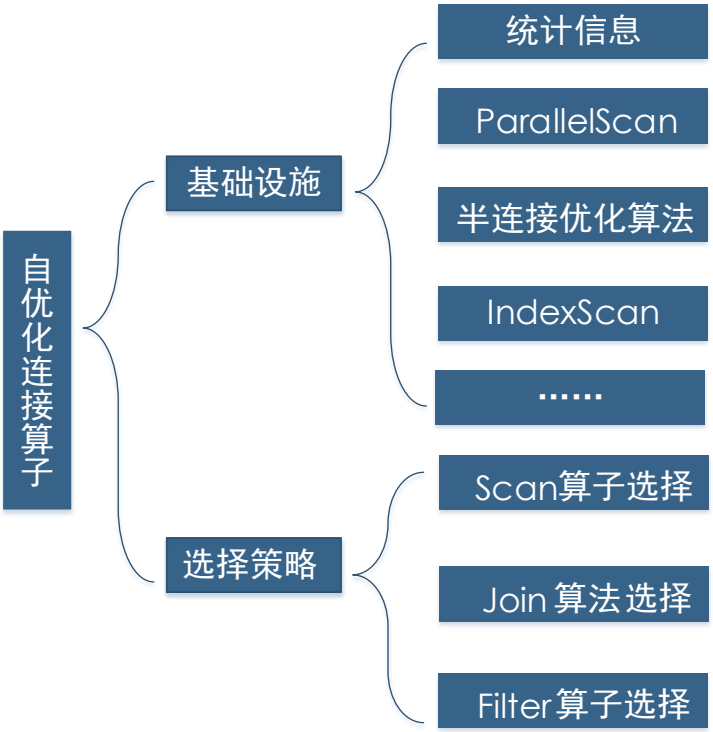

图 1.2 自优化算子组成

本文章节组织如下：

第二章首先向读者介绍了国内外在分布式系统查询领域的工作现状；接着介绍了分布式数据库系统的一般架构：LSM 存储引擎加上读写分离架构；然后，描述了这种架构下分布式数据库系统的组织方式及其读写的基本流程；再接着，简

单介绍经典数据库系统当中的连接算法，以更好地在后文中阐述自优化连接算子的原理；再接着，介绍了 Oracle，MySQL 以及 PostgreSQL 中统计信息的收集和使用方式，并提出了分布式架构下统计信息的一种实现方式；最后，说明了读写分离架构下的副本可读性。

第三章介绍了本文的研究背景，描述了本文需要解决的问题。首先，介绍了 Cedar 数据库，给出 Cedar 中主控服务器，基线数据服务器，增量数据服务器以及 SQL 代理服务器的概念；接着，介绍在 Cedar 数据库二级索引功能模块的原理及实现，并在此基础上实现了 IndexScan 算子；最后介绍在 Cedar 上实现半连接优化算法的相关细节以及其使用场景。

合并模块是读写分离架构下的独有模块，第四章首先介绍了这种架构下的系统中每日合并流程；接着介绍了读写分离架构下数据库的统计信息的收集过程和使用方法，归纳了统计信息要收集的统计量维度，并设计了统计信息的存储方式。

第五章介绍了基于统计信息的 ParallelScan 算子的实现方式。通过使用并行技术，可以充分利用到副本可读性，提升基线服务器的资源利用率，切实加快数据访问速度。接着，围绕了并行粒度进行了讨论，阐述了对扫描范围的不同的切分粒度对查询性能的影响。最后，分析了 ParallelScan 算子和 TableScan 算子之间的关系。

第六章介绍了自优化连接算子的实现思想。并介绍了在自优化连接算子内部对 IndexScan 和 TableScan 算子对选择策略；描述了 ParallelScan 的使用场景；比较了 InFilter 和 BtwFilter 的功能特点，阐述了 In 和 Btw 的选择策略。

最后，在第七章中对全文对工作进行了总结，并对未来实现一款较为通用的分布式查询优化器工作进行了展望，阐述了在 Cedar 中实现查询优化器还需要做哪些工作。

第二章 国内外研究现状

2.1 国内外数据库研究现状

在数据库发展的初始阶段，无论是企业还是政府，其业务规模都还比较小，数据量也比较少，这种情况下，应用程序，数据库，文件等一切的资源都在一台服务器上。最典型的情况是：操作系统使用 Linux，应用程序使用 PHP 开发，然后部署在 Apache 上，数据库使用 MySQL，汇集各种免费开源软件以及一台廉价服务器就可以开始系统的发展之路了[15]。

随着业务的发展，数据量不断增加，用户访问量也逐渐增多，应用系统的压力不断上升，将所有服务部署在一台机器上的方案与业务需求之间的矛盾越来越突出。使用集群的方式减轻高并发带来的压力是这一阶段人们的解决方案：将多台廉价的 PC 机器按照一定策略部署在一起，并通过负载均衡、分布式一致性协议等技术，从软件层面上对外提供一致服务。很自然地，数据库的发展进入了分布式的阶段。

相对传统集中式数据库，分布式数据库有如下这些优势：

1. 更高的数据访问速度：分布式数据库为了保证数据的高可靠性，往往采用备份的策略实现容错，所以，在读取数据的时候，客户端可以并发地从多个备份服务器同时读取，从而提高了数据访问速度。
2. 更强的可扩展性：分布式数据库可以动态增添存储节点来实现存储容量的线性扩展，而集中式数据库的可扩展性十分有限。
3. 更高的并发访问量：分布式数据库由于采用多台主机组成存储集群，所以相对集中式数据库，它可以提供更高的用户并发访问量[16]。

在 2003 年，Google 发布 Google File System[17]论文，以分布式文件系统为

例，阐述了如何在廉价的普通硬件的基础上提供可靠服务。GFS 为后续的 NoSQL 系统，大数据计算工作打下了牢固的基石。2004 年 Google 发布 MapReduce[18] 论文，描述了大数据的分布式计算方式，其思想是将任务（task）进行分解，然后在多台处理能力很弱的计算节点中同时处理，最后将结果合并从而完成整个 task。并行计算正是 MapReduce 思想的一种体现，本文设计 ParallelScan 的思想也是如此：在多核处理器的基础上，并行进行数据的扫描操作，可以大大减少数据的访问时间。2006 年，Google 发布 Bigtable[4]论文，开启了数据库 NoSQL 时代，国内外掀起了一股 NoSQL 研究风潮。Bigtable 是构建于 GFS 之上的一款分布式存储系统。

在国外，除了 Google 以外，其他公司也诞生了许多知名的分布式数据库产品：构建于 Hadoop 之上的 Hbase[6]，它是 Bigtable 的开源实现，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统[19]。如同 Hbase 这样的 NoSQL 系统是为了解决大数据的存储和计算而诞生的，虽然其具有很多优势，但是其也放弃了对于 SQL 的支持，这使得应用从传统数据库迁移过来的成本非常高；并且，大多数 NoSQL 系统都不支持事务，这也使得 OLTP 的业务无法迁移到 NoSQL 上；VoltDB[20]是 Postgres 和 Ingres 联合创始人 Mike Stonebraker 领导开发的内存数据库管理系统，属于 NewSQL 系统，既解决了传统集中式数据库扩展难的问题，同时又保证事务的原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）等特性，简称 ACID 特性；F1[21]是 Google 继 Bigtable 之后的另一款产品，F1 融合了 Bigtable 的高可扩展性和 SQL 数据库的可用性和功能性。F1 建立在 Spanner[22]之上，而 Spanner 通过利用一个全球位置服务（GPS）和原子钟实现了分布事务间强一致性（2PC）、无锁读事务、原子 schema 修改、基于时间戳的整体排序、通过 Paxos 协议进行同步复制、数据的自动均衡等功能。F1 是一个全新的数据库系统，目前主要服务于 Google 的广告系统；与 VoltDB 和 F1 不同，HAWQ[23]是 Pivotal 公司推出的一款主要针对分析型应用（OLAP）的 SQL On Hadoop 产品。HAWQ 支持标准 SQL 接口，同时其对

事务的 ACID 特性也全部支持，不仅如此，HAWQ 还具有一款非常成熟的并行优化器（Orca）。Orca[24]是一款具有较好独立性的查询优化器，除了可以使用在 HAWQ 之上，还可以很方便地迁移到其他数据库产品之上。

在国内，也有许多研究工作正在开展当中，也诞生了不少优秀的数据库产品。阿里巴巴公司推出的支付宝产品是一款国内第三方支付平台，致力于提供“简单、安全、快速”的支付解决方案[25]。支付宝在内取得了巨大的成功，离不开其背后分布式数据库系统 OceanBase[26]的支持，以下简称 OB。目前 OB 内部版本已经达到 1.0 版本，开源版本为 0.4 版本，托管于 GitHub。本文的原型系统 Cedar 在 OB 开源版本的基础上进行了很多优化工作，比如分布式索引，日志强一致，存储过程，高可用的三集群架构，非主键多行更新，表锁等。同时，阿里巴巴在 2016 年还开源了一款基于 MySQL 研发的数据库产品：AliSQL，目前由阿里云团队维护，主要为阿里集团遇到的电商秒杀、物联网大数据压缩、金融数据安全等场景提供个性化的解决方案；除了阿里之外，腾讯公司微信团队也开源了其高可用分布式数据库 PhxSQL，PhxSQL 是一个通过 Paxos 保证强一致和高可用的 MySQL 集群，目前应用在微信后台的账号系统、企业微信、及 QQ 邮箱；除了这些大型互联网公司，国内有许多创业公司也在做这方面的研究实践工作：PingCAP 致力于实现一款类 Spanner 的开源数据库产品 TiDB；巨杉软件公司于 2012 年开源他们自研的一款文档类 NoSQL 数据库。

数据库是上层应用可靠性，可用性的基石。随着数据量，访问量地不断增加，分布式数据库越来越成为研究的重点内容。分布式数据库主要包括查询和事务两个重要的模块，查询中主要包括 SQL 支持，索引，存储过程，查询优化以及查询执行等内容。本文主要从查询优化的角度，结合多种技术，切实提升数据库的 Query 的执行速度。

2.2 LSM 架构的特点

2.2.1 LSM 存储架构

LSM 存储模型是 O'Neil, Patrick 等人于 1996 年提出的，其主要思想是延迟更新和批量写入，它将“日志文件写入后只可访问不可更改”的思想运用到数据管理之上：首先将写入的数据保存在内存中，删除数据也以更新的形式写入内存，然后在以某种策略将内存的数据融合到磁盘上。这样做的好处是将大量的磁盘随机写入转换成为顺序写入，可以节约大量的写入时间。自 BigTable 论文发表之后，LSM 模型受到很多关注，目前在 LevelDB，HBase，Cassandra，InfluxDB 等产品中均得到了应用。

读写分离是分布式数据库特有的一种体系结构，具体说来，是指在集群环境下，将服务器角色划分为读和写两种。读节点提供读取数据的功能，一般而言，是指那些存储静态数据的服务器；写节点对外提供数据写入功能，可以将写节点是理解为一个较独立的内存数据库，在内存中存储增量数据。增量数据在达到写节点内存上限或者某个预先设置的阈值之后，通过合并模块，将数据合并到读节点之上。

图 2.1 是一张典型的读写分离分布式系统的架构图。ReadNode（以下简称 RN）是读节点，在 RN 磁盘中存储的是数据的静态部分，这部分数据不允许再修改，因此被称为静态数据；WriteNode（WN）是写节点，在其内存当中存储每天数据库更新的数据，这部分数据通常称为增量数据。

可以看到整个架构结合了 LSM 存储模型和读写分离两大特点：将 LSM 的内存部分作为一个单独的写节点，将 LSM 的磁盘部分作为一个单独的读节点。在集群当中，RN 是可扩展的，即集群中 RN 是可以动态增加或减少的，每个读节点存储整个数据库中的一部分数据，并通过 Master 模块对这些数据的分布信息进行管理。通常，分布式数据库都会使用副本冗余的机制来保证高可用特性，因

此同一块数据存在许多不同分片，这些不同分片存储在不同的 RN 上。将查询请求进行拆分，对每块分片的不同副本同时读取，可以加快扫描数据的速度，这正是本文 ParallerlScan 算子的工作原理。RN 上数据的组织方式通常是采用 Sorted String Table（SSTable）[3]的形式，SSTable 中的数据按照主键排序后存放到连续的数据块中，对外提供一个可持久化的映射关系。SSTable 包括多个 Data Block，Block 之间按照主键排序存储，接着存放的是 Block Index，通过 Block Index 可以快速确定某个范围的数据在哪一个 Block 上。一旦 SSTable 写入硬盘之后，就是不可变的。更新也是通过追加写的方式进行。

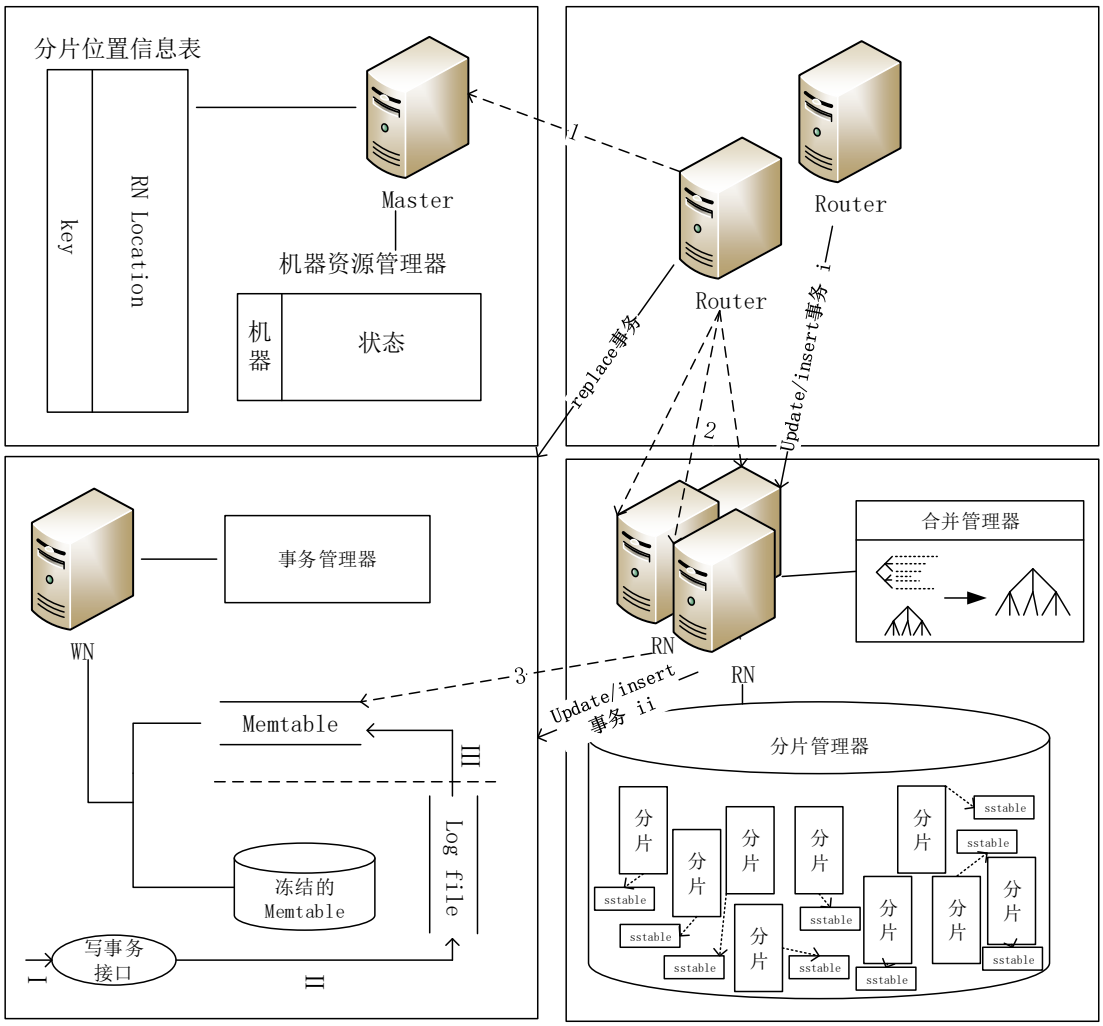

图 2.1 读写分离分布式数据库架构图

在写节点上，数据是存储在内存当中，不同的数据库实现对这些数据可能有不同的数据组织方式。通常，在 WN 上，数据都是以一种称为 MemTable 的数据结构进行存储。MemTable 在其大小达到阈值的时候会被 Dump 到磁盘。Dump 到磁盘的被称为冻结的 MemTable，是只读的，不允许写入操作。

Router 是整个集群的路由转发器，负责将客户端发送过来的请求转发到对应的服务节点上。对读事务，根据 Master 的元数据信息直接请求对应 RN 节点。对于写事务，如果是 Replace 事务，直接转发至 WN 节点，如果是 Insert/Update 事务，在获取到静态数据之后发送给 WN 节点。WN 节点内部写事务接口会先记录操作日志，然后将修改应用到 MemTable 上。

因为 WN 的内存是存在上限的，所以，增量数据会在合适的时间被合并到 RN 之上，并与 RN 上的数据进行融合。这部分是合并模块的功能，将在后面给予说明。

2.2.2 数据存取流程

数据的读取流程如图 2.1 中虚线所示，客户端将查询请求发送给 Router，Router 在获取到查询的范围之后，向集群的 Master 请求这些数据的位置信息。Router 的数量也是可以水平进行扩展的，通过这种方式可以解决高并发访问的问题。集群中每台 Router 都是无状态的，它们的内存中会保留数据库当中所有表的模式信息（Schema），当执行了 DDL 操作时，Master 会将更新之后的 Schema 分发给每台 Router。有了这些信息之后，Router 就可以对 Query 进行解析，获取此次 Query 的扫描参数，包括扫描哪些数据表，扫描数据表的哪些列，扫描什么范围的数据。Router 会将扫描参数发送给集群的 Master。由 Master 查询其内部的元数据信息（MetaData），获取到查询表所在的 RN 信息，并将这些信息返回给 Router。然后 Router 可以将相应的请求分发到对应的节点，由对应的节点执行相应的存取逻辑。

为了返回完整的数据，通常 Router 还会根据 RN 返回的信息，重新构建读取

参数，去读 WN 的 MemTable，将数据动态增加的部分获取到。在 Router 的内部还有将静态数据和动态数据合并的逻辑。

数据库之中的写事务通常包括更新，插入，删除以及替换，不同的写事务的具体实现根据语义的不同会有所区别。但大致流程如下：Router 在接收到客户端发送过来的请求之后，进行词法，语法以及语义分析，发现这是一条写事务。如果是 Replace 事务，直接发送给 WN，并将 replace 的结果写入到 MemTable 当中；如果是 Update 或者 Insert 事务，需要先去 RN 读取数据的静态部分，再将静态部分数据和写事务的数据一起发送给 WN。当行不存在或者被删除的时候，Update 什么也不做，Insert 执行插入操作；当行存在时，Update 执行更新操作，Insert 事务返回行已存在错误。Delete 操作在行存在情况下会往 MemTable 当中插入一个<Row, delete>标签，在合并的时候，这行数据才会真正意义上从数据库中被删除。

本文的研究内容是如何加快分布式数据库当中 Join 的执行速度。从数据的读取流程中可以看出相对于传统的集中式数据库而言，分布式数据库中的 Join 的数据表格往往是跨多个网络节点的，因此也多了很多的网络开销。所以优化分布式 Join 的一个思路就是最小化不同网络节点之间的数据传输量。

2.3 常见的连接算法概述

无论是在传统的关系型数据库还是在 OLAP 系统，抑或是在离线的批处理系统中，Join 算法都是最至关重要的。传统的 Join 算法包括 Nested Loop Join，MergeJoin 以及 Hash Join。这三种算法是最常见的 Join 算法，都有各自的适用场景。

假设由两张数据表 R，S 要做自然连接，R 表比 S 表小。

Nested Loop Join 算法是指从 R 中依次取出元组 r，以 r 的连接 key 值去扫描表 S，如果 S 中 key 值和 r 中的 key 值满足连接条件，那么将结果置于结果集当中。通常选择小表作为驱动表，也就是外层循环的表。Nested Loop Join 算法不

用对数据事先进行排序，因此能得到较好的响应时间。

MergeJoin 算法是数据库中最常用 Join 算法，利用了归并排序的思想。将参与 Join 的表按照 join 列的数据视为一个非常长的数组，首先按照 join 列进行排序。之后对这些排完序的元组进行合并，当 join 列的值满足 join 条件的时候，将结果输出到结果集当中。其算法的时间复杂度较低，除去排序算法，对两张表只需要遍历一遍即可。

Hash Join 算法也是常用的 Join 算法，拥有多种不同的变种，可以加快 Join 的流程。最简单的 Hash Join 可以分为三个阶段：Build 阶段，Probe 阶段和 Join 阶段。Build 阶段将小表 R 按照 key 生成一个内存 Hash 表；Probe 阶段顺序扫描大表 S，扫描过程中查询 Hash 表，以确定是否满足等值条件；最后 Join 阶段将结果存入结果集当中。Hash Join 算法在 Join 的表缺少可用索引的时候，整体性能快于前两者。查询内存中的 hash 表避免了循环过程中的随机读。

而在分布式环境中对于 Join 的处理通常有三种策略：一是将所有的数据拉取到同一计算节点，然后使用类似于集中式数据库的 Join 算法来获取 Join 结果；二是在不同的数据节点上并行地做局部的 Join，最后将每个节点的执行结果做一个汇总，其思想类似于 Map Reduce；三是将其中的一张小表广播分发到大表所在分区节点，然后并发地在每个节点上做 Join，最后汇总结果。第一种做法要求每张表的数据量都比较小，使得网络上的开销尽可能的少；第二种做法要求相同范围数据的分片在数据库中有相同的分布策略；而第三种做法适用于小表极小，广播代价才会比较低。

2.4 数据库中的统计信息

通过上述对 Join 算法的概述，可以知道 Join 算法与参与 Join 的数据表的数据规模，数据分布，有无索引以及 Join 列的基数等因素都有很大的关系。这些信息都来自于数据库的统计信息模块。传统的 MySQL，Oracle 以及 PostgreSQL 等产品对数据库的各种状态都有相对完善的记录体系[27，28，29]，而为解决大数

据问题的 NoSQL，NewSQL 等系统由于发展时间还比较短，对于统计信息收集工作的支持都不是很好。本文研究了传统数据库中统计信息的实现方式，并结合 Cedar 系统的特点提出了分布式统计信息的收集以及使用方式。

由表 2.1 可以看出，在构建数据库的统计信息的时候需要考虑的因素主要有三项：统计内容，统计时机以及统计信息的存储方式。对于统计内容的每一项，一般对应于数据结构中的一项。可以看到统计内容一般有四个维度：表维度，列维度，索引维度以及系统维度，在表维度当中通常包括表所包含的行数，聚集索引占的页的数量，每条记录的平均长度等等信息；在列维度中通常需要统计的信息包括唯一值的个数，列上的最小值，最大值，选择率因子以及空值的个数；索引的统计信息通常包括索引的层数，叶子节点的个数以及唯一值的个数等等；系统维度的统计信息只有 Oracle 实现了，主要的统计指标有 IO 以及 CPU 的性能和利用率。

表 2.1 各数据库产品中统计信息

产品	统计内容	统计时机	存储方式
MySQL	表维度 索引维度	定时轮询或者某些条件被触发时	持久化存储或者易失性存储
Oracle	表维度 列维度 索引维度 系统维度	自动收集 手工收集	数据字典 直方图
PostgreSQL	表维度 列维度 索引维度	DB 用户执行 Autovacuum	系统表 pg_class 和 pg_statistic 中

存储方式方面有两种选择：持久化或者易失性。持久化存储一般将统计信息

存储在数据库的系统表中，当数据库重新启动之后仍然可以从系统表中获取对应的数据。而易失性存储是将统计信息存储在内存当中，这样在执行查询优化的时候可以直接从内存当中获取到数据，减少了一次访盘。

一般而言，统计信息都不是十分精确的，自动收集的模式下可以设置超过多少数据被更新之后触发统计信息的重新计算，手动模式下只有当用户执行到某些特定的操作的时候才会触发更新，比如执行 Analyze Table 命令，执行 Show table status，Show index 等命令时。而在 PG 中对于大表的统计信息收集采用采样收集策略。统计信息一般用于选取较优的查询计划，因此对于精确性要求不是十分高，所以在实现统计信息的收集时可以适当放宽准确度要求。

2.5 数据副本的可读性

副本机制是数据库存储领域用来保证系统高可用性的常用手段。一般而言，使用越多的副本，系统的可用性就越高，但是数据的备份不仅仅只能考虑副本的数量，还需要保证事务的 ACID 特性。使用副本可能带来如下问题：同一个时间从不同副本读取数据能否保持一致？副本故障是否会引起系统的不可服务？副本同步是否会影响到事务的原子操作？等等。通常有多种方式来保证多副本之间数据的一致：主从异步复制，主从同步复制，主从半同步复制，多数派读写等。

1. 主从异步复制的典型代表就是 MySQL 的 binlog 复制，当主副本完成写日志操作之后即返回客户端事务成功执行，之后的某段时间再通过日志的方式将内容同步给其他副本。这样读取副本就无法保证强一致；
2. 主从同步复制恰恰相反，只有将事务同步到所有副本之后，才可以返回客户端成功。这样导致的直接后果就是，响应时间变长，可用性降低；
3. 半同步复制是一种折衷方案，只有满足 N 个副本同步完成之后才返回事务成功执行；
4. 多数派读写只需确保集群中的半数副本同步成功便返回成功，因此在读取的时候只要半数副本返回的值一样，便可认为读取成功。

在读写分离的架构下，数据的插入操作都会写入到写节点当中，并通过合并流程合并到静态数据当中。而此架构下的副本都是静态副本，写事务不会直接影响到副本。因此，每个副本在两次合并之间都能够保证强一致性。这就使得我们能够同时读取各个副本，将一个查询并发起来以提升效率。

2.6 本章小结

本章首先回顾了分布式数据库目前的国内外研究现状，指出了分布式数据库具有的研究价值，研究意义等。与此同时，指出了本文研究的主题，分布式数据库的 Join 优化对于提升 Query 性能的意义。紧接着，分析了基于 LSM 存储引擎的读写分离式数据库产品的一般架构并分析了其读写事务的执行流程，结合读事务的执行，分析了优化 Join 算法的难点，思路和意义。然后，对比了传统的数据库中的 Join 算法，归纳了分布式 Join 算法的一般做法。另外，也指出了统计信息在 Join 优化，查询优化方面的重要意义以及实现统计信息的入手点。最后，结合分布式数据库的特殊架构，阐述了副本可读性的理论依据。

第三章 相关工作及选题

3.1 概述

本章主要介绍本文工作的原型系统 Cedar 以及一些与本文相关的前期工作，包括Index以及半连接算法的实现等，并在最后给出本文实现自优化算子的缘由。

3.2 Cedar 数据库

Cedar 是一款可扩展，高可用，低成本的分布式数据库产品，是华东师范大学分布式系统组基于阿里巴巴开源版本 OceanBase 进行研发的。目前，OceanBase 已经广泛使用在阿里集团的金融领域，比如支付、交易、清算等。阿里集团广为人知的支付产品：支付宝背后的数据库产品就是 OceanBase。2016 年 2 月 1 日，Cedar 项目组完成了 0.1 版本的开发测试工作并将项目托管于 GitHub。0.1 版本新增的功能包括：高可用的三集群架构，游标，存储过程，二级索引，非主键多行更新等等。2016 年 9 月 26 日发布了 0.2 版本，新增的功能包括表锁，快照隔离级别，布隆过滤器连接算法等。

本文选取 Cedar 作为论文的系统原型主要基于以下考量：

1. Cedar 是一款优秀的分布式数据库系统，采用了 LSM 存储加上读写分离的体系架构，具有出色的扩展能力，拥有较为完整的查询处理框架，对于 SQL 语法支持也较为完善。但是 Cedar 内部可以使用的算子数目仍然偏少，查询优化工作也不是很充足。
2. Cedar 是基于 OB 做开发的，而 OB 是经过实际业务场景考验的，使用 Cedar 系统可以借鉴 OB 的丰富的应用场景。利用这些场景，在 Cedar 上的研究成

果可以被证明是有效可用的。

3. 由于 Cedar 的体系架构是比较具有代表性的，因此本文实现的各种算子以及自优化的 Join 算法可以在其他类似的系统中得到实现。

本小节将简要概述 Cedar 的体系架构，并介绍 Cedar 的读取数据的具体过程。在介绍过程中，本文将描述在测试 Cedar 过程中遇到一些问题，并给出问题定义。

LSM 引擎与读写分离体系的分布式系统中一般存在 Master，RN，WN 以及 Router 四种角色。在 Cedar 中与之对应的分别是 RootServer，ChunkServer，UpdateServer 以及 MergeServer[26]。

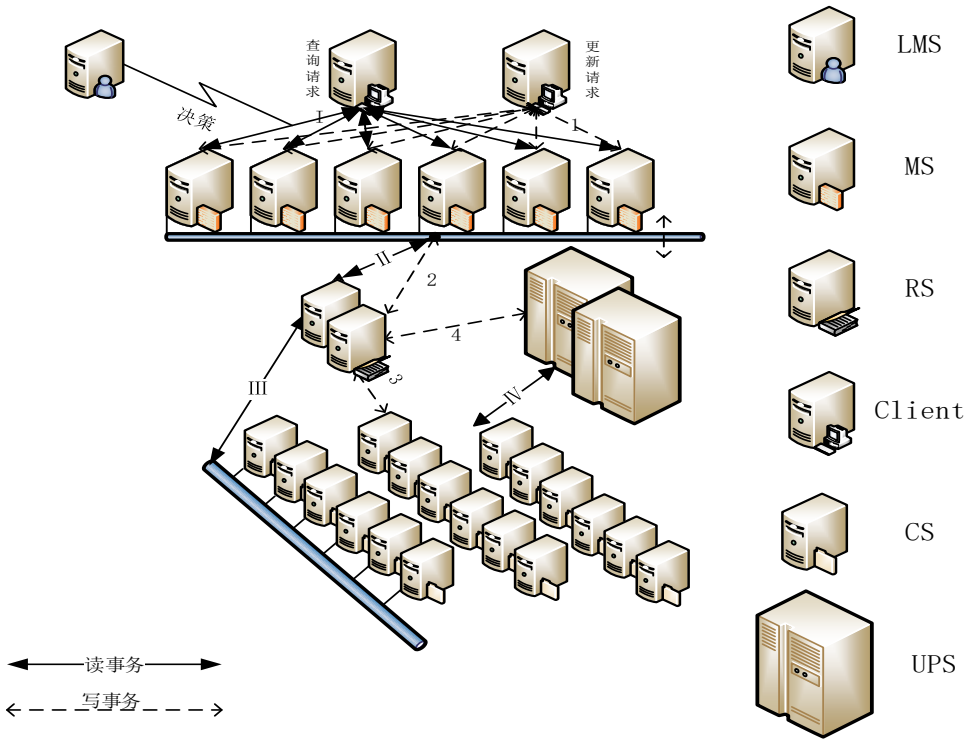

图 3.1 Cedar 架构图

如图 3.1 所示：

- RootServer(RS)是集群的主控服务器，是集群的管理者，管理集群其他

Server，管理子表等，保存数据的模式信息，分布信息等元数据。采取双机热备保证可用性。

- ChunkServer(CS)是数据存储服务器，提供分布式数据存储服务。每台 CS 上都有查询处理模块。
- UpdateServer（UPS）是更新服务器，提供一个内存事务引擎，可以认为是一个写节点和事务管理的合集。内存中的数据以 B 树形式组织。由于 UPS 在目前的架构下还存在单点写入问题，负载会比较高，一般 UPS 机器的配置要求会相对较高一些。单点 UPS 的好处是可以避免分布式事务。
- MergeServer（MS）是合并服务器，负责 SQL 解析，查询计划的生成以及执行，最终将结果返回给客户端。
- Listener MergeServer（LMS）是决策客户端请求转发到哪台 MS 上。在集群中，MS 都是对等的实体，LMS 将请求转发到负载较轻的 MS 上，起到负载均衡的作用。

数据表在 Cedar 中按照主键排序并划分成为数据量大致相等的分片(Tablet)，均匀存储在集群的每个 CS 当中。采用 LMS 架构的系统都有合并子系统，在 Cedar 中，有定期合并和手工合并两种方式。定期合并过程中 CS 需要将本地静态数据与冻结内存表的增量更新数据执行一次多路归并，融合后生成新的静态数据并存放到新生成的 SSTable 中，并向 RS 完成新 SSTable 的注册工作。

3.3 Cedar 读事务流程

在图 3.1 中，实线部分表示的是读事务执行流程，虚线部分表示的是写事务的执行流程。当 MS 接收到来自客户端的网络包之后，会在其内部对包进行解析，取出 SQL 语句。与其他数据库类似，SQL 语句在 MS 内部会依次经过 Parser 模块，完成词法语法解析，生成语法树；然后经过 Planner 模块，进行语义分析，生成逻辑树。目前，在 Cedar 中，并没有查询优化器模块，而是对逻辑计划树进行简单的调整，转换成为物理计划树。最后由 MS 将对应的请求发送给 CS 或者

UPS 去执行。如果 SQL 请求涉及多张表，在每台 CS 将读取的部分数据返回给 MS 之后，由 MS 执行结果合并。

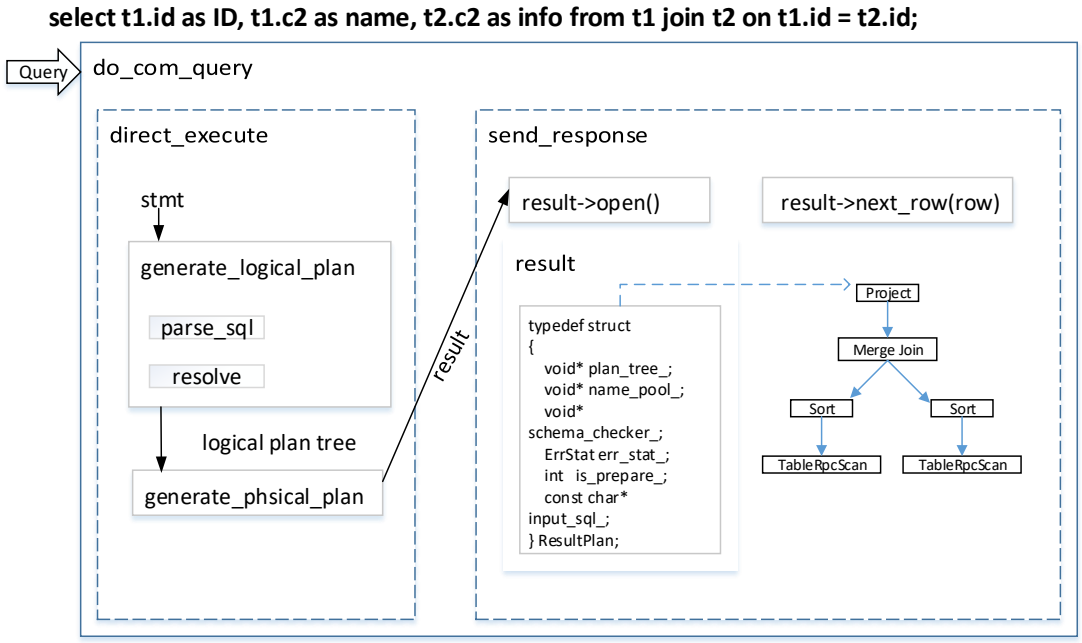

图 3.2 MS 读事务流程图

图 3.2 展示了一条 SQL 在 MS 中的处理流程，实线箭头表示流程走向：Cedar 处理 Query 的接口是 do_com_Query，这个接口包含 direct_execute 和 send_response 两个模块。direct_execute 实际上相当于 parser 模块，对 stmt（string 类型，保存的是 SQL 文本）进行词法语法解析，生成逻辑计划树。逻辑计划树作为 generate_phsical_plan 接口的输入，被转换成为物理计划树，物理计划树存储在 result 的 plan_tree 中。Result 中存储有关此查询树的一些基本信息，图中给出了 result 的数据结构：name_pool 指向词法解析阶段申请的一块内存，逻辑计划实际被保存在这块内存中；schema_checker 用于检查表模式的合法性；err_stat_用于保存 MS 处理的状态信息，若能够正常解析，则保存 success 状态，否则保存相应的错误码。之后，Result 被作为参数传递到 send_response 模块。send_response 模块实现的是物理算子 Open，Next，Close 的迭代器。执行的过程中，Open 根据

传入的物理计划树对算子进行初始化，Next_Row 实现按行返回满足条件的行，Close 阶段（未在图中画出）关闭每一个算子。

物理计划树的节点是数据库中的算子，不同算子可以完成不同的功能，常见的算子有 Filter，Project，Sort，Join 以及 Scan 等。Filter 是过滤算子，完成按照指定的条件对数据进行筛选的功能，对应于 SQL 中的 Where，Having 等操作；Project 是投影算子，完成选择输出列的功能，对应于 SQL 中的 Select 操作；Sort 是排序算子，完成对数据的排序，常对应 SQL 中的 Order by；Join 算子包含很多子类，每个子类对应一种 Join 算法，其对应 SQL 中的 Join 操作，在 Cedar 中默认的 Join 算法是 MergeJoin 算法；Scan 算子完成扫描数据的功能，内部包含 MS 与 CS 之间的交互的接口，常对应于 SQL 中的 From 操作。

3.4 Cedar 二级索引的实现

在 OB 开源版本中并不支持在非主键列上建立索引，当用户查询请求不包含数据表主键时，Scan 算子会请求数据表各个分片所在的 CS。如果数据表的规模非常大，而用户需要的数据只属于某个很小的范围，那么这种非主键查询的效率非常低。几乎在所有的 NoSQL 系统当中，都不支持非主键查询的操作。然而，现实是很多业务都有着查询非主键的需求。

为了解决这个矛盾，Cedar 在 0.1 版本，推出了二级索引解决方案。二级索引实现时对于算子的支持不是特别友好，本文在以前工作的基础上，设计实现了 IndexScan 算子，用于支持非主键查询。现给出 Cedar0.1 中的索引解决方案概述。

3.4.1 分布式索引算子的设计

分布式系统一般拥有很好的可扩展性，通过动态增加机器的数目可以获得更多的存储资源。在设计分布式索引的时候，利用了数据冗余的思想。为了便于描述，本文使用“索引表”表示 Cedar 中通过 Create index 语句创建的表，使用“数

据表”表示 Cedar 中通过 Create table 语句创建的表，使用“原表”表示与索引表对应的数据表。

在 Cedar 中，索引表的组织形式采取和数据表完全一样的方式。因此，可以使用 Select 语句直接查询索引表中的数据，但是系统不允许直接对索引表的数据进行删除或者更改。索引表构建流程大致如下：

1. 模式构建

用户通过使用 Create Index 语句指定在某张数据表的某列上建立索引，此列称为索引列。之后，系统会根据原表的 Schema 在内部完成对应索引表 Schema 的构建。此时的索引表处于初始化状态，不可提供服务，索引表中不包含任何数据。

2. 数据构建

索引表与原表之间拥有亲缘关系，需要保证索引表中的数据与原表中的数据的一致性。为了使索引表能够对外提供服务，必须根据原表的数据对索引表的数据进行构建。在定期合并完成后，可以保证所有的更新数据都从 UPS 合并到 CS 上。我们修改了 Insert 语句执行流程，保证定期合并期间插入原表的数据也会被插入到索引表当中。定期合并完成会触发索引表数据构建流程。

数据构建流程分为三步：

第一步，RS 通知原表每个主分片所在 CS 开启局部构建流程。在局部构建流程中，CS 会将数据从原表中提取出来，按照索引列排序，重新组织，写入到临时的 SSTable 中，并向 RS 汇报这些 SSTable 的范围和大致分布信息。

第二步，RS 根据所有 CS 汇报的信息，对索引列按序均匀划分，切分出来若干范围信息。这些范围信息表征了索引表的分片信息。接着，RS 会为每一个范围选取一个对应的 CS。CS 接收到了范围信息之后，从自身或者其他 CS 中请求属于这些范围的数据，重新写入索引表的 SSTable 中。

第三步，检查索引表的校验和与对应原表的校验和，确认无误后，由 RS 将索引表的状态由 Init 改为 Available，并将此信息同步至集群中其他服务器。如果出现错误，经过重试机制之后，将索引改为 Error 状态。

3. 提供服务

数据构建完成之后，处于 Available 状态的索引表可以对外提供服务。

3.4.2 索引一致性

索引表在提供服务之后与原表之间保持强一致。

Cedar 中所有的写事务原子操作中都新增了处理索引表的特殊流程：在更新原表的同时，更新索引表。通过这些特殊流程可以保证索引表自创建时刻起，其在 UPS 中数据与原表保持一致。从上一小节描述可以知道，索引在完成数据构建之后，CS 中的静态数据能够跟原表保持一致。在定期合并流程中，Cedar 是仍然允许更新操作的，修改后的写事务流程能够保证这些更新的操作会被同样操作到索引表上。这样动态数据和静态数据在任何时刻都是一致的。

为了保证一致性，实现中牺牲了写事务的部分性能。

3.5 Cedar 半连接算法概述

3.5.1 半连接优化算法的原理

在本文第二章介绍了数据库系统中一些 Join 算法，这些都是数据库中最基本的 Join 算法。然而，基于 OB 的 Cedar 对 Join 功能的支持不是很完善。在 Cedar 的初始版本中，只实现了 MergeJoin 算法。当参与 Join 的两张表规模比较小的场景下，MergeJoin 的执行效果比较好。因为此算法会将数据都拉取到 MS 上，当数据表规模比较大的情况下，MergeJoin 算法近乎失效。

半连接优化算法的思想在[30]中被提出，可以用于分布式数据库的 Join 算法的化简。其基本思想是利用半连接首先消除那些悬挂元组，然后再将半连接的结果发送的另一张表所在节点进行 Join 操作。这样做能够加快 Join 速度的前提条件是利用半连接能够消除大多悬挂元组，使得网络上传输的数据量变得很少。网

络上数据量越少，半连接优化算法效果越好。

本文采用了这种思想用于优化 Cedar 中小表与另一张大表 Join 的场景，实验证明这种优化在特定场景下是十分有效的。

3.5.2 半连接优化算法的实现

假定有两张表 R，S 参与 Join（R ⋈ S），R 表规模较小，S 表规模很大，需要很多网络传输时间才能全部传输到 MS 上。这种情况下，MergeJoin 算法不再适用。

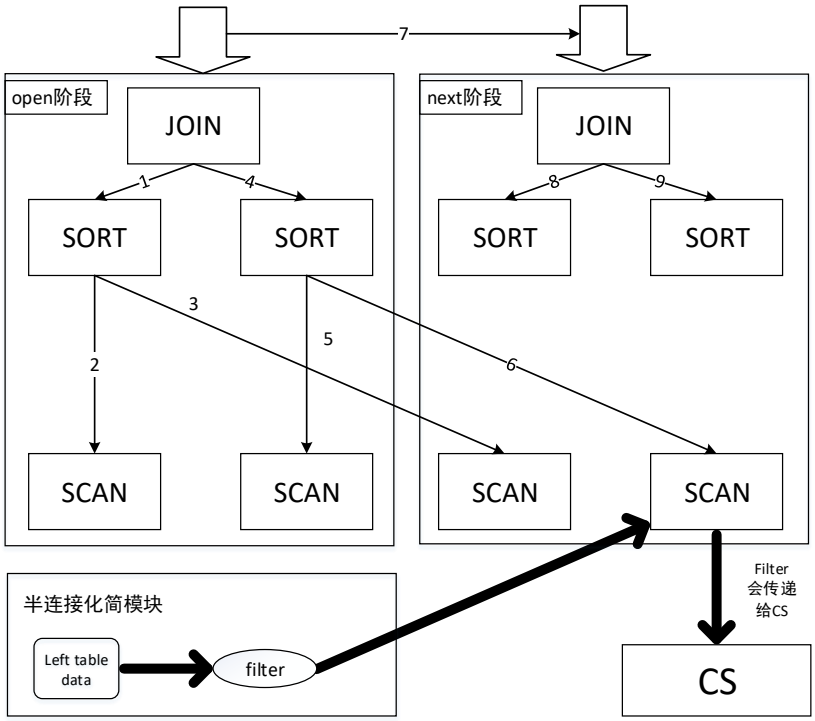

图 3.3 半连接优化算法思想

图 3.3 阐述了 Cedar 原 MergeJoin 的执行流程，较细的箭头的数字代表原先 Cedar 的执行先后顺序。由于 Sort 会等待数据全部到齐才可执行排序，所以在 Sort 的 Open 阶段会先调用子算子的 Open，紧接着会循环调用子算子的 Next，直到所有数据被拿到或者排序用的内存被填满。从图中可以看到，在左边 Sort 的

Open 执行完成后，左表的数据其实已经可以访问了。Cedar 修改了 MergeJoin 原有的执行逻辑。在 S 表的 Scan 算子执行 Open 操作的时侯，通过调用 R 的 Next_Row 操作访问左表的数据，之后将 R 上的 Join 列数据构造成过滤条件发送给 S 表所在 CS。这部分就是半连接优化模块所做的工作。其他流程均与 MergeJoin 相似。

S 表根据 R 表发送的过滤条件对数据进行过滤，之后再将数据返回给 MS。MS 继续调用 MergeJoin 的算法对来自两张表的数据进行 Join 操作。

3.5.3 多线程半连接

半连接优化的思想应用的场景在 3.4.1 小节也提到过，是 Join 的两张表有一张表比较小，并且在利用小表对大表的过滤也比较好的情况下才比较有效。多线程半连接优化方法是为了解决大表间的 Join 性能问题而提出的，其做法是利用并行的思想。与本文在第二章提到的广播小表的方法类似，多线程半连接优化方案的做法是：将左表的数据首先取回到 MS，之后对其按照索引列排序后进行分片，将每个小的分片交给一个线程。每个线程所完成的功能就是将它所得到的分片当作一个小表，然后执行半连接优化算法。所有的线程都完成半连接算法之后，在由主线程完成数据的合并工作。

目前，解决大表与大表 Join 问题的比较通用的办法，就是利用多线程技术，MapReduce 的思想。这种做法并未减少网络上数据的传输量，但是相对于 MergeJoin 来说，它提高了任务执行的并行度，这样整个任务的处理时间就等于执行速度最慢的那个线程所花费的时间加上做左表切分和数据合并的时间。

3.6 选题

上文中提到的各项工作，包括分布式索引，半连接优化算法等，均能改进一些场景下的查询性能。但是为了使用它们，用户不得不显式地用 Hint[31]的方式在 SQL 语句中写出来，使用起来很麻烦。

本文在开发索引，维护半连接优化算法的工作基础之上，设计并实现了ParallelScan 算子，提出了 Cedar 的统计信息的收集方法。在统计信息的基础上，将以前各方面工作整合在一起，设计并实现了一种自优化连接算子，在此算子内部，可以根据数据库的统计信息来动态选择子算子，自我进行优化。

3.7 本章小结

本章主要介绍了本文开发工作的原型系统—Cedar，并陈述了选择 Cedar 作为原型系统的原因。首先介绍了 Cedar 的架构特点，接着阐述了在 Cedar 中的读事务的流程。因为本文主要关注的是查询相关工作，因此对于写事务流程并未过多提及，写事务具体过程可以参考论文[32]。

接着，介绍了本文的一些背景工作，包括 Cedar 中分布式索引的实现和在 Cedar 的 Join 算法执行过程中半连接优化思想的运用。

最后，给出了问题描述，并向读者说明了本文实现自优化连接算子的原因。

第四章 统计信息的收集和使用

4.1 分布式架构下的合并模块

数据库系统中统计信息的计算往往呈现周期性：短时间段中，表格的统计量往往不会有太大的变化；对统计信息的大幅更新往往会占用大量系统资源，影响系统可用性；在使用统计信息时对其精确度的要求往往并不是非常高。本文中统计信息的收集方法依赖与 Cedar 中的每日合并模块，详见 4.3 节。本节先给出分布式架构下合并算法概述以及 Cedar 中每日合并的大致流程。4.3 给出了在每日合并模块基础上的统计信息收集方案。

在 LSM 存储引擎下，数据的基线部分和增量部分是分开存储的。目前，Cedar 的事务处理器是集成在 UPS 上。为了避免分布式事务，UPS 采取的是单点架构设计。单机的内存是有限的，因此在增量数据达到一定额度之后，需要转储到 CS 上。在系统中，合并模块负责完成转储算法。

论文[3]中描述通用合并算法的具体过程，如图 4.1，C0 和 C1 Tree 是 LSM 引擎的组件，通常以树形数据结构组织。在 C0 和 C1 中，相同层次的节点在物理上是相邻存储的。合并过程中，对于树中每一行，是按照内存快依次进行滚动合并的。合并过程不覆盖原来的旧的 C1 节点，而是写入到新节点当中，以保证发生故障时的可恢复性。C0 中被成功合并的节点将会从内存中删除。具体的合并过程通常采用归并排序，然而，具体采用什么样的策略，可以结合使用场景来决定。

在 Cedar 中，UPS 为了对外提供服务，在合并之前，会将内存中的数据结构进行冻结，转储到磁盘上，并在内存中重新申请一块空间构造 MemTable。这种做法可以保证在每日合并期间更新事务仍然能够正常被执行。如图 4.2 所示，左

上部分的是 UPS 中的 MemTable 的组织形式，它采用的是 B 树加上行操作链表的方式，图中所示的是冻结后的 MemTable。左下部分是 CS 中的 SSTable 组织形式，SSTable 的行定位机制可以采用如图所示的树形结构表示。MemTable 和 SSTable 合并之后会生成新的 SSTable。

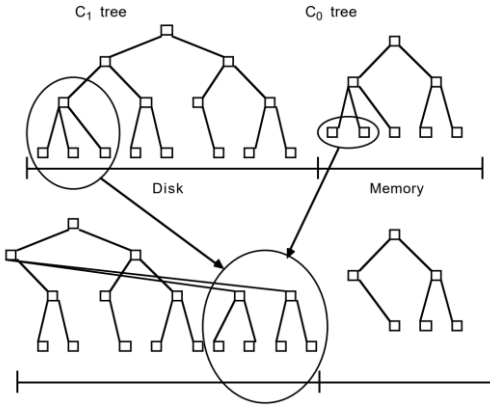

图 4.1 LSM 引擎一般合并过程[3]

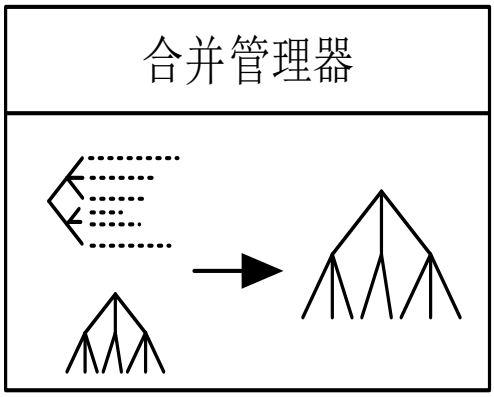

图 4.2 Cedar 合并管理器

4.2 统计信息的收集

4.2.1 统计信息的内容

根据统计信息收集过程中需要完成的指标，可以将其分为表级统计信息，列级统计信息以及系统统计信息。

目前本文的统计信息模块并未涉及系统级统计信息。表 4.1 和 4.2 记录了本文实现的表级统计信息和列级统计信息的详细收集内容。

如表 4.1 所示，在表级统计信息方面，本文主要会记录下每张表的行数，每行的平均长度，该表每个分片的大小以及大致的数据分布情况。

表 4.1 表级统计信息

table_id	表的 id
Rows_count	行数
Row_size	行平均长度
tablet_size	分片大小(压缩后)
HistogramInfo	数据分布情况

如表 4.2 所示，对于列级统计信息而言，本文会记录该表指定列的数据类型，不重复值个数，最大最小值，以及指定值出现的次数。

表 4.2 列级统计信息

column_id	属性 ID
table_id	表 ID
column_type	数据类型
distinct_num	不重复值数
max_value	最大值
min_value	最小值
<value, counter>	指定值出现的次数

4.2.2 统计信息的收集流程

数据表的分片是统计信息收集的基本单位，类似于合并模块，为了加速统计信息的收集，本文使用了 MapReduce 的思想，使得不同分片之间的统计信息收集能够并行起来。在 MS 实例能够提供服务之前，用于收集信息的线程池就已经被初始化完成，每个线程都会阻塞等待合并统计信息的信号。在编码时，为了降

低统计信息模块和每日合并模块的耦合度，我们设计在每日合并完成之后才开始统计信息的收集工作。

每日合并过程中会将有增量数据的数据表的表 id 加入到 statistic_table 数组当中。在每日合并模块完成之后，若发现 statistic_table 中有待收集统计信息的表时，唤醒阻塞的工作线程。每个线程的工作流程如下图所示：

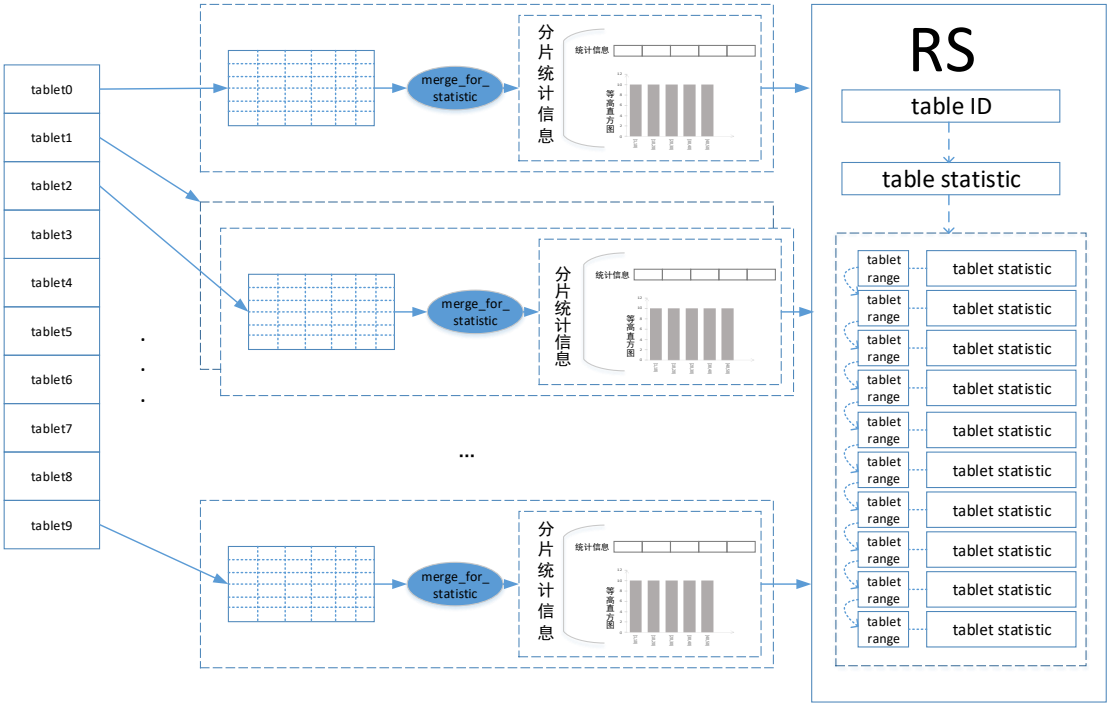

图 4.3 MapReDuce 方法计算分布式表统计信息

如图 4.3 所示，每个线程负责一块具体分片的统计信息的收集，相当于 Mapper，分片所属的表的表 id 即为 Key，Value 就是收集存储统计信息的数据结构。这些 Mapper 线程都会将信息通过 RPC 的方式汇报至 RS，RS 中存在一个类似 Reducer 的线程，会对 Mapper 中的数据进行合并，依此来构建统计信息。

统计信息模块重用了合并模块中使用的 Merge 算子。Merge 算子的执行过程中会对此分片的每一行进行遍历。在每行遍历过程中，分别使用 Rows_count，tablet_size 等变量记录相应的统计信息，原合并流程会查询这行对应 UPS 的增量

数据，对每个 Cell 进行更改，构造新行，写入新的 SSTable 中[33]。与每日合并模块不同的是，统计信息构造模块不再考虑 UPS 上的数据。在对 Cell 进行修改的流程中，可以对指定的列进行统计信息的统计工作。

4.2.3 特殊情况的考量

一种特别的情况是当天的更新事务产生了大量的增量数据，这些增量数据使得数据库中的一张表的数据的总量、数据分布发生了质的变化。如前所述，由于统计信息的收集工作安排在每日合并之后，这就使得当天的统计信息发生失真。

本文实现时也考虑到了这种情况。当系统中出现很多慢查询是在同一张表上时，很有可能是由统计信息失效导致的，此时需要 DBA 人工介入，手动发起一次统计信息的更新操作。

4.3 统计信息的存储方式

在前文介绍经典数据库中中统计信息的相关内容时，提到了统计信息的存储方式主要有两种：持久化存储或者易失性存储。持久化存储就如同 PG 一样，将统计信息存入的磁盘文件(系统表)中，使用时需要从文件中读取，Hot Query 的相关信息存放在内存当中，以优化查询；另一种方式是在内存当中以某种数据结构，往往是直方图的形式存储统计信息。

目前，本文仅实现了在易失性存储的统计信息部分，存储在 RS 的内存当中。下面以表级统计信息为例，介绍统计信息的相应的数据结构的设计。

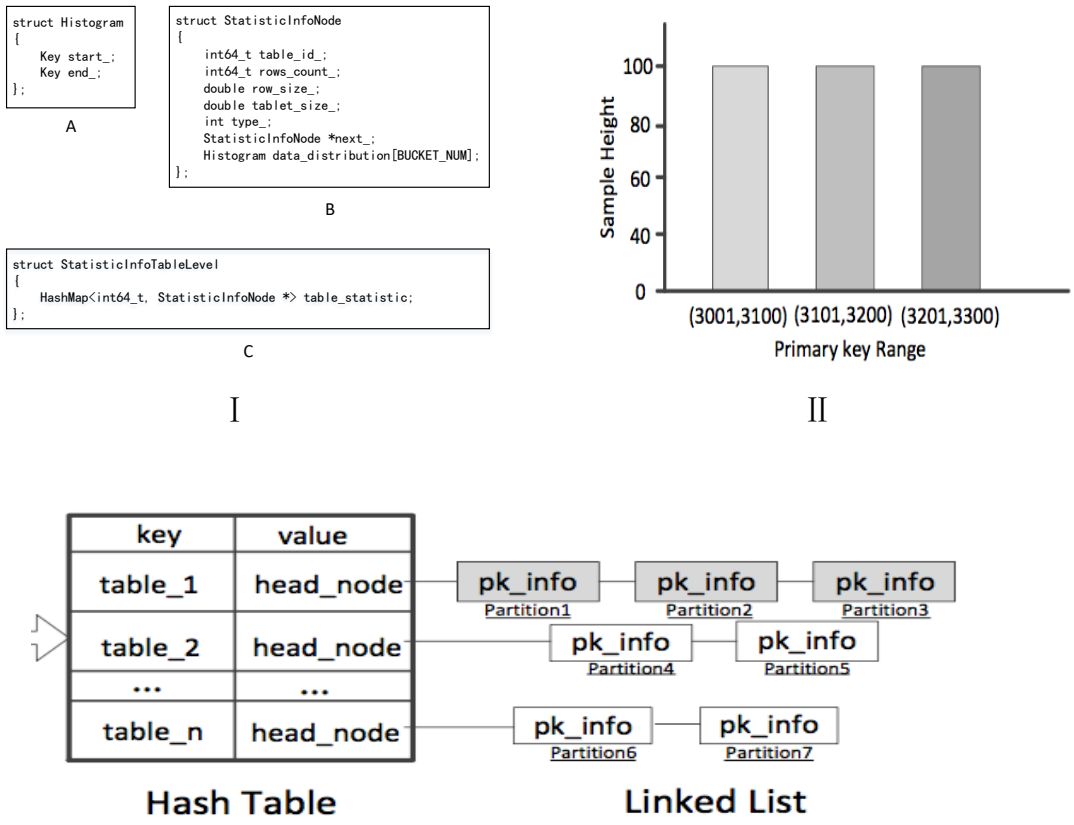

图 4.4 统计信息存储设计

图4.4中，Ⅰ是统计信息的数据结构，A是直方图的设计，直方图记录该Bucket的起始和终止键值，B是统计信息的节点，对于每一个单独的分片而言，都会为其申请一个统计信息节点，其中 type_变量指定了该分片的直方图的具体种类：等宽直方图或等高直方图。对于系统中的所有表的统计信息以 Map 形式进行组织，Map 的键为表 id，值为其统计信息链表的入口节点。如图 4.4Ⅲ所示，根据表 id 可以获取到每张表的数据分布情况。图 4.4Ⅱ所示的是等高直方图，其收集方法是在遍历表的每一行时，每隔 100 行，记录键值。这样可以保证记录的两个键值之间的数据量是一致的。

4.4 统计信息的使用

4.4.1 统计信息用于 ParallelScan 算子

主键的统计信息用处之一就是为 ParallelScan 算子提供相应分片的切分点，算子的相应实现对应本文第 5 章。使用统计信息可以保证对每个分片的切分会更加均匀，这更符合 Round-robin 算法对每个 CS 进行选择的策略。

如图 4.5 所示，当 MS 开始对一个分片进行更细粒度切分的时候，会向 RS 请求这个分片的统计信息。根据统计信息的数据结构的设计，首先，RS 会查询统计信息所存的 Map，由表 id 获取到统计信息链表，查找链表可以找到相应的分片。

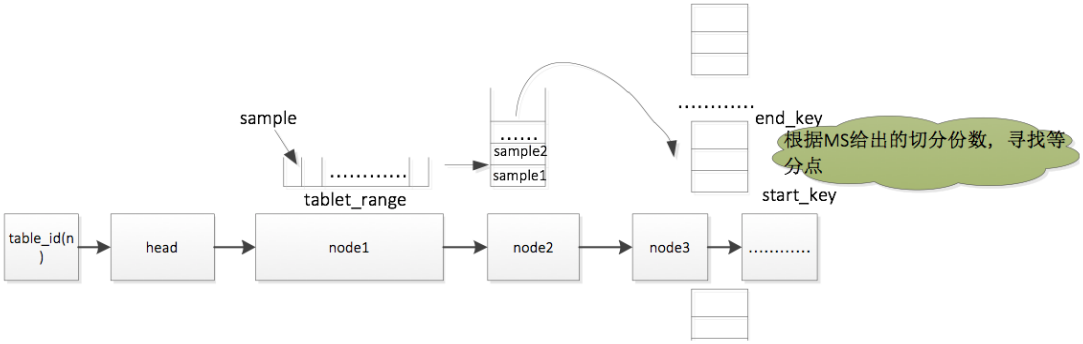

图 4.5 计算切分点

对于某些查询而言，其查询范围并不能覆盖整个分片，因此我们需要大致定位查询所覆盖的分片有哪些，在根据覆盖分片的总行数和 MS 需要切分的份数决定返回给 MS 的切分点信息。

4.4.2 统计信息用于 IndexScan 算子

在本文 6.2 节，提到 IndexScan 算子在某些情况下会产生大量回表请求，性

能反而会比普通的 Scan 算子慢。为了避免这种情况下，Scan 算子仍然被优化成为 IndexScan 算子，需要使用统计信息加以判断。

可以通过列统计信息的收集模式，将会使用到的列和指定值的统计信息预先进行收集。在实际使用的是时候根据此列此值出现的次数来预估回表的大致次数，便可以计算网络通信开销的总代价。如果此值比使用 Scan 扫描整张表的代价大，那么实际就选用 Scan 算子而不进行优化。

下面给出一个不适合使用 IndexScan 进行优化的场景。假设存在如下 Query：

Query1：**select** * **from** A **where** A.index_column = x;

A 表总共有 100 万行，每行数据的大小假设为 1MB，在 A 表的 index_column 上已经创建了索引表，查询满足 index_column 值为 x 的行，索引表每行 0.5MB。假设统计信息给出非主键值 x 出现的次数为 80 万，即超过一半的数据都是 x。

假设网络带宽为 100M/s;，那么传输一行数据的网络开销大致为 0.01s。Scan 算子需要全量传输数据，即 100 万行都从 CS 取回到 MS，网络开销约为 2.78h。

IndexScan 的代价是多少？IndexScan 会根据从索引表取回的每一行，获得原表主键信息后，继续从原表取回完整数据，容易计算出 IndexScan 的查询的网络代价为 3.34h，当然这部分时间仅仅是网络的代价，IndexScan 算子中 MS 构造原表查询信息的代价还未计算。

4.4.3 统计信息的更多用途

除了使用在 ParallelScan 和 IndexScan 算子当中，统计信息在查询优化器的其他方面也起到了独一无二的作用，特别是在估算中间结果集大小，做基于代价模型的查询优化的时候。

4.5 本章小结

统计信息在数据库的查询中起到了至关重要的作用，是基于代价模型的查询

优化器的基石。分布式环境下考虑统计信息的构建时往往需要考虑更多的因素。本文结合了 MapReduce 的设计思想，根据 Cedar 架构的特点，设计和实现了统计信息收集模块，结合底层统计信息的结构设计，分析了目前统计信息的两个使用场景。

第五章 并行扫描算子实现

扫描算子是数据库中最基础也是最常用的算子之一，扫描算子的执行效率会影响数据库中的所有 Query。在图 1.1 中可以看到：当数据量比较少时，取数据的时间比网络上传输数据所花费的时间要多很多。本文在对 Scan 算子的流程分析的基础之上，结合了 Cedar 分布式架构的特点，利用并行读每个分片副本的方法来加速 Scan 算子取数据的时间，封装实现了并行扫描算子（ParallelScan 算子）。实验证明，ParallelScan 算子在小数据量的情形之下可以带来 90%左右的性能提升，能够覆盖绝大多数业务场景。

5.1 Scan 算子的执行流程分析

在分布式数据库中，物理计划的执行就是物理计划树的树形结构从根节点开始迭代执行的过程，通常树形结构中的每个算子都对应于一个特定的关系代数运算符，每个算子都由其物理上的前驱节点驱动。当完成整个树形结构地遍历之后，数据从存储服务器被传输到查询服务器，然后返回给客户端。每个算子都会提供最基本的三个接口：Open，Next 和 Close，以完成上述迭代过程。

对于 Query 来说，scan 算子通常处于执行计划的最底层，其功能就是完成对指定 table，指定范围内的数据进行扫描。在 Cedar 中，不同分片的数据可以并发进行扫描。主要过程如下：

1. 从 RS 上获取集群中分片的元数据信息，将要扫描的数据的范围（Range）按照分片信息进行拆分，得到若干个子范围(Sub Range)，每个 Sub Range 对应一个 CS 上的一个分片。
2. 唤醒工作线程，每个工作线程获取一个 Sub Range，然后将这个 Sub Range

封装成为一个 Request，紧接着将 Request 发送给这个 Sub Range 对应的 CS。发送完成后，阻塞线程，等待 CS 返回结果。

3. 由于是并行的去每台 CS 上去扫描，不同 CS 的负载情况不同会导致 MS 接收到的数据包并非一定按照请求发送的顺序回来。因此 MS 内部实现了一个排队逻辑，当满足一定顺序，比如该 Range 的第一个 Sub Range 数据返回后，才开始将数据返回给上层操作符。

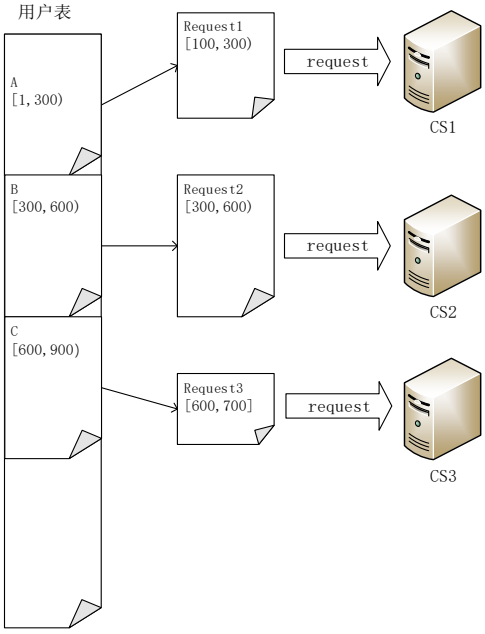

图 5.1 Scan 算子执行流程

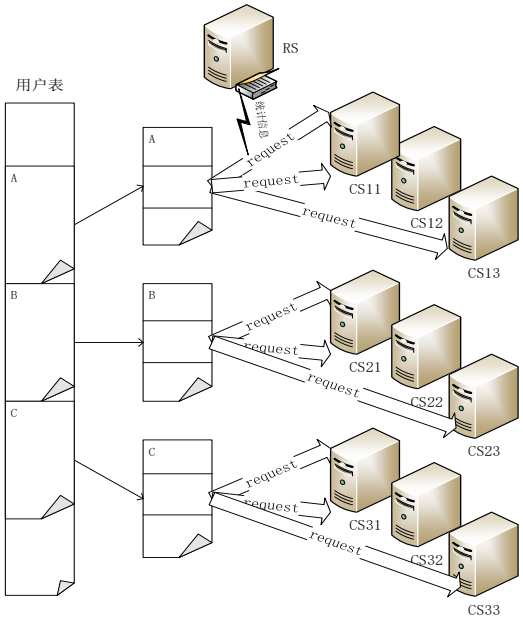

图 5.2 ParallelScan 算子执行流程

如图 5.1 所示，假设有一张用户表，表 ID 为 3001，其主键是 UserID，整型自增。某次扫描的数据范围是 UserID 属于[100,700]，MS 会将表 ID 和这个范围设置给 Scan 算子，Scan 算子在 Open 阶段会向 RS 请求分片的位置信息，假设是[1,300)在 CS1，[300,600)在 CS2，[600,900）在 CS3，[900,1200)在 CS4…那么 MS 将会封装 3 个 Request：[100,300),[300,600)以及[600,700]，分别发送给 CS1，CS2 以及 CS3。这个 3 个 Request 在 MS 处是同时发送出去的，由于 CS1，CS2,CS3 相对于 MS 的物理距离以及处理速度均有差异，很可能出现 Request2，即[300,600），

首先返回到 MS。MS 中的排队逻辑会判断出此次接收到数据并非 Range[100,700]的第一个 SubRange，因此会继续等待。当 Request1 返回数据后，MS 才会开始向上层操作符返回数据。另外一方面，由于 Server 之间传输的数据包大小限制为 2MB，因此 CS 一次并不能返回[100,300)所有的数据。假设用户表每 10 行大小恰好为 2MB，那么 CS 第一次返回的数据应该是[100,110)。MS 在处理完这个数据包后，继续向对应的 CS 请求[110，300）范围的数据。这个过程会维持在一个 Session 当中，源源不断将 SubRange 中所有的数据都获取到之后才会断开。

MS 整个取数据过程存在的一个问题是：当 CS 将数据准备好并返回给 MS 的时间要长于 MS 内部处理数据的速度时，MS 会因为不能及时得到数据而处于阻塞态。也就是说，MS 不断将数据处理好并返回给上层算子的速度要远远快于 CS 把数据准备好并传送给 MS 的速度，这表明 CS 取数据过程限制了整个 Scan 算子的执行速度。

5.2 ParallelScan 算子的设计

在分布式数据库中，通常会采取多副本来保证数据的可用性。由第二章的讨论可知，在 LSM 存储模型的数据库中，副本之间的数据保持强一致。因此可以将原本落在一个分片上的查询进行拆分，使其落在这个分片的各个副本上，再由 MS 做一个简单的汇总。

如图 5.2 所示，假设分片 A，B，C 均有三个副本存储在集群中不同 CS 上，那么 ParallelScan 实现的难点是如何得到切分点，将 A,B,C 切成三份。最简单的是按照分片的范围进行均分，如[300,600)可以均匀切成 3 份：[300，400)，[400，500)，[500，600)。在本例中，这么做是没有问题的。但是考虑以下三种情况：

1. 查询条件可能不是主键
2. 主键不是自增列
3. 主键类型不便于按范围均分

对于第一种情况，在 Cedar 中会转换到主键进行查询；对于第二种情况，很

难保证对分片进行均匀地切分，即可能出现情况是[300,400)这个范围只包含几条数据，而[400，500)区间可能包含上万条数据。不均匀的切分会导致后面为每个切分选择副本的过程变得复杂；对于第三种情况，当主键的数据类型不是 Int 类型，而是 Varchar 类型或其他，这样会浪费许多 CPU 资源去计算每个切分点。

正是由于以上原因，本文才引入了统计信息模块。在表维度的统计量中会有这张表的大致分布情况，简而言之，就是会记录切分点，使得每两个切分点之间的数据量是一致的。有了统计信息，在对分片进一步切分的时候，结果会更加准确，同时免去了大量地 CPU 计算。但是这样方式的代价是：统计信息的收集需要额外的计算资源。另外，需要注意的是，必须等待统计信息生效后才能使用 ParallelScan 算子。在 ParallelScan 内部有判断统计信息是否生效的逻辑，如果没有统计信息，ParallelScan 的不会对分片进行下一步的切分。

5.3 ParallelScan 算子的实现

算法 5.1 描述了 ParallelScan 算子的工作逻辑。因为在原来 Scan 算子的逻辑中就已经对查询 Range 进行了切分，ParallelScan 算子会对每个 Sub Range 再次进行切分，这会导致系统当中的 Request 的数目会急剧膨胀，当 Request 得不到及时处理时，会积压起来，对系统的整体性能产生副作用。因此 Line2 判断当前并行度小于系统最大并行度的时候才可以进行分片的切分，以保护系统。

算法的 Line3-13 描述了如何根据统计信息对查询涉及到的每个分片进行再次切分。算法的第 5 行是根据从 RS 返回的信息判断统计信息是否生效：具体而言，如果返回的统计信息为空，那么统计信息是未生效的；如果统计信息非空，那么将对比统计信息和该分片的范围信息，判断其是否一致。算法第 8 行提到给每个 Sub_Sub_Range 分配副本的方法是轮询式，这种方式的优点是逻辑简单，便于实现，缺点是未能真正考虑每台 CS 上实际的负载情况，可能会导致某台 CS 在某个时刻负载特别高，反而影响到整体的查询性能。在以后的工作中，本文会添加更多的选择 CS 的算法，来避免这些问题。

算法第 16 行会阻塞主线程，这样做的主要目的是在系统负载较高的情况下对系统进行保护。代码第 18 行处理了出现异常的情形，当切分分发过程中出现错误或者处理超时，ParallelScan 返回失败。

Alogrithm5.1 ParallelScan 处理逻辑

```
输入： 处理后的子范围数组（Sub_Range_List)，每个子范围只属于某个分片
输出： 每个子范围切分后的结果
1  从查询范围中获取查询表的 id;
2  while (当前并行度小于系统最大并行度时) :
3  |   for Sub_Range In Sub_Range_List :
4  |   |     根据表 id 和 Sub_Range,从 RS 获取该分片统计信息;
5  |   |     根据 RS 返回结果判断统计信息是否生效;
6  |   |     if (统计信息生效) :
7  |   |     |     根据统计信息对 Sub_Range 进行切分,得到 Sub_Sub_Range;
8  |   |     |     采用 round_robin 算法给 Sub_Sub_Range 分配一个副本;
9  |   |     |     为每个 Sub_Sub_Range 封装一个 Sub_request;
10 |   |     |     将 Sub_request 发送给对应的副本 CS
11 |   |     else
12 |   |     |     此分片不进行切分;
13 |   |     将此 Sub_Range 标记为已处理
14 |   curr_parallel_count++;
15 |   if (当前并行度达到了系统上限) :
16 |   |     等待当前并行度小于系统最大并行度;
17 |   |     从上次处理的断点继续处理;
18 |   if (超时或者某个过程出现错误) :
19 |   |     设置错误码和错误信息;
20 |   |     break;
```

```
21 return Sub_Sub_Range
```

5.4 ParallelScan 算子的切分粒度讨论

并行算法一般都是利用多处理器多线程的机器特性，同时处理任务，来达到缩短总任务执行时间的目的。ParallelScan 算子的切分粒度实际上指的是对每个分片进行再切分的份数的多少，切分粒度直接决定处理这些任务线程的数目。如果使用的是线程池技术，那么切分粒度决定了是否会有任务的积压。

表 5.1 切分份数对 ParallelScan 算子的影响

测试 SQL		**Q1**	**Q2**
不进行切分	结果集大小(行)	300099	202196
	执行时间(s)	6.48	7.32
切分粒度为 3	结果集大小(行)	300099	202196
	执行时间(s)	5.73	4.7
	提升比	11.57%	35.79%
切分粒度为 6	结果集大小(行)	300099	202196
	执行时间(s)	4.63	4.65
	提升比	28.55%	36.48%

并行任务调度问题已经被证明是 NP-hard[34]。切分粒度影响了并行的粒度，因此暂时没有一个好的方法给出一个准确的数字，来使得 ParallelScan 算法达到最优。本文实现过程中，向用户提供设置切分粒度的接口，支持用户在执行 ParallelScan 算子前，通过更改系统参数设置此次执行的切分粒度。如果用户没有指定切分粒度，将由系统自动将其设置为当前表每个分片平均拥有的副本数目。

本文对不同切分粒度做了一个简单地对比，如下所示：

Q1: select * from fd_user;

Q2: select * from fd_user **where** user_id<2000141344 **and** user_id>103031;

表 5.1 分别列举了对 Q1、Q2 不进行切分，切 3 份及切 6 份的查询执行结果。在实验中，每个分片的副本数是 3。可以看出通过增加切分粒度，可以一定程度上加快查询的速度。但是 Q2 相对于 Q1 的提升不是很明显，原因在于 MS 上多做了一个 filter 操作，MS 处理任务的速度就会降低，查询的速度被限制在 MS 的处理速度上，而并非阻塞等待数据上。

5.5 ParallelScan 算子并行性能分析

相对于 Scan 算子而言，ParallelScan 算子拥有更高的并行度。按照上一节的分析，在 ParallelScan 算子中切分粒度默认是副本的数目。

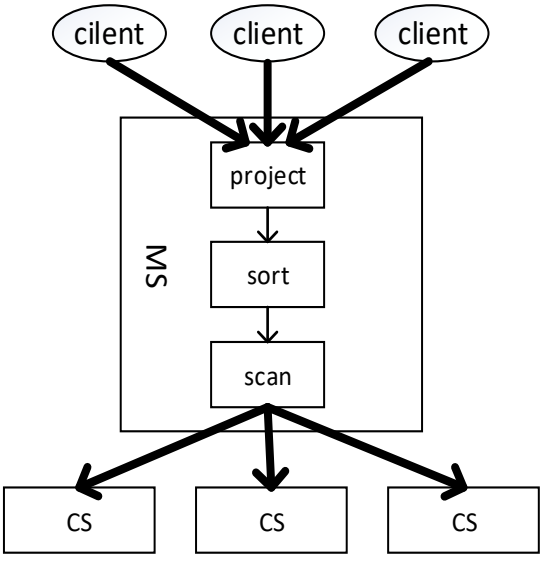

图 5.3 整体执行流程

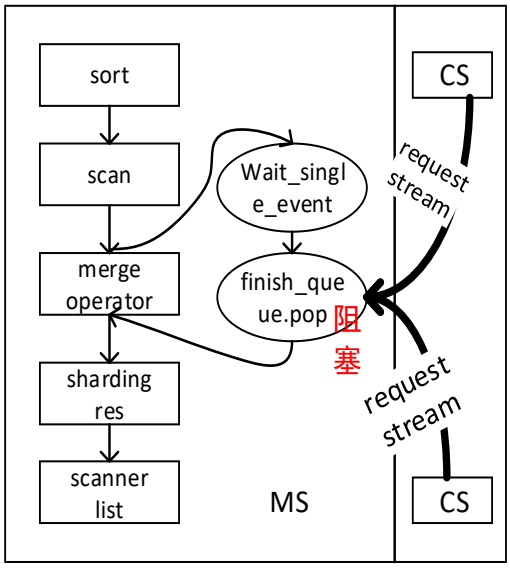

图 5.4 Scan 内部执行流程

图 5.3 是 Cedar 中最简单的 Query 的物理计划树。粗线代表两个模块之间有

网络数据的交互。图中，Sort 操作符会在 Open 阶段等待所有行到来，之后再对结果进行排序，所以该 Query 总的执行时间可以分为两个部分：第一部分是 MS 中 Sort 算子的子树迭代的时间，即从 Scan 算子返回数据的时间和；第二部分是 MS 与 CS 交互的时间，这部分时间中包括了 MS 等待 CS 数据准备好并发送过来的时间。Sort 的 Get_Next_Row 阶段会将排好序的数据返回 Project，再通过 Libeasy[35]返回给客户端。整个 Query 的时间包括 Project 处理返回的时间，MS 迭代物理计划树的时间以及 MS 和 CS 之间数据交互的时间。如图 5.4 所示，Scan 算子内部会与 CS 进行交互。主要分为两部分，一是等待 CS 数据到来(包括 CS 取数据和网络传输的时间)；二是更底层算子处理 Row，层层迭代返回给 Sort 的时间。

从图 5.4 可以看到在 Scan 算子的内部有一个 Scanner_List_，这是一个 Scanner 的链表，Scanner 是 Cedar 中数据在内存中存储的数据结构。如果 Scanner_List 为空，即此时 Finish_Queue_为空，那么 MS 中执行查询的主线程将会阻塞在 Finish_Queue_的出队操作上。当 Request 从 CS 返回时，被放入到 Finish_Queue_当中。Finish_Queue_不为空时将会唤醒工作线程把放入队列的 request 中的 data 取出来存入到 Scanner 中。Merge_Operator 是负责将到来数据按照主键合并排序。每一个 Sharding_Res_对应与一个分片，前文中提到了 CS 返回数据是以 2MB 包的形式返回的，这些 2MB 包返回后会被放入对应的 Sharding_Res_中。

因此在对 ParallelScan 进行性能分析时需要区分两种情况：总的 Query 执行时间提升比例和 Scan 算子消耗时间提升比例。本节的实验部分，通过统计 MS 各个线程阻塞在 Finish_Queue 上的总时间来表征 Scan 算子的消耗时间。本章开头部分提到的 90%是指第二种情况。

5.6 实验

本节设置了一系列对比实验，对 ParallelScan 算子的实际效果进行探究。

5.6.1 实验环境

5.6.1.1 机器配置

本节实验采取的机器性能指标如表 5.2 所示：

表 5.2 服务器配置

角色	CPU	内存	磁盘	网络
CS/MS	6 核 12 线程（Intel (R) Xeon (R)） CPU E5-2620 V2 @ 2.10GHz）	64GB	3TB SSD	千兆网
RS/UPS	6 核 12 线程（Intel (R) Xeon (R)） CPU E5-2620 V3 @ 2.10GHz）	165GB	1.5TB SSD	千兆网

5.6.1.2 集群配置

集群采用了 3 台 CS，3 台 MS，1 台 RS，主备两台 UPS 的架构，如图 5.3 所示：

表 5.3 实验集群配置

RS	UPS	CS	MS
182.119.80.72	182.119.80.62 182.119.80.63	182.119.80.65 182.119.80.66 182.119.80.67	182.119.80.65 182.119.80.66 182.119.80.67

5.6.1.3 测试表

本实验中采取的表结构一致如下：

表 5.4 表结构

字段名	id	name	col2	col3	col4	col5	col6
类型	int32	varchar(1020)	double	int32	int32	int	int32
主键	pk	0	0	0	0	0	0

建表语句如下所示：

create table table_name(id int, name varchar(1020), col2 double, col3 int, col4 int, col5 bigint, col6 int, **primary key**(id));

根据需要，将 table_name 替换成为不同的表名。

5.6.1.4 数据集

主键均匀分布，从 1 开始自增，数据集大小是指操作系统中查看到的导入数据库前的文件的大小。导数工具使用项目组研发 import 工具。如表 5.5 所示，分为 smallfile、testfile 以及 bigfile 三种类型。

5.6.1.5 负载

100%读，SQL 如下：

Q1：select * from smallfileX;

Q2：select * from testfileX;

Q3：select * from bigfileX;

Q4：select * from bigfileX where id>Y;

需将 X 换为对应的数字，Y 为此表的最后一个分片的起始 key，即（Y，MAX]中的 Y。

表 5.5 测试数据集

表名	总行数(万行)	大小	表名	总行数(万行)	大小
smallfile0	1	11M	testfile0	100	1004M
smallfile1	5	51M	testfile1	200	2.0G
smallfile2	10	101M	testfile2	300	3.0G
smallfile3	15	151M	testfile3	400	4.0G
smallfile4	20	201M	testfile4	500	5.0G
smallfile5	25	251M			
smallfile6	30	302M	bigfile0	1000	9.9G
smallfile7	35	352M	bigfile1	2000	20G
smallfile8	40	402M	bigfile2	3000	30G
smallfile9	45	452M	bigfile3	4000	40G
smallfile10	50	502M	bigfile4	5000	50G

5.6.1.6 测试结果表说明

- 关于测试结果中图例的说明：

1. 基础版本总执行时间：未打开优化开关时测试的 SQL 的总执行时间
2. 优化版本总执行时间：打开优化开关时测试的 SQL 的总执行时间
3. 基础版本等待数据包时长：未打开优化开关时测试中 Finish Queue 总的阻塞时长
4. 优化版本等待数据包时长：打开优化开关时测试中 Finish Queue 总的阻塞时长

   通俗而言，优化开关控制的就是是否使用 ParallelScan 算子。

- 测试结果中坐标轴意义说明：

1. 主要横坐标：数据集编号
2. 主要纵坐标：Query 执行时间，单位秒
3. 次要纵坐标：优化版本较基础版本性能提升百分比

- 关于测试结果说明：

1. 为了尽可能排除偶然因素的影响，测试过程中所有 Query 均执行多次，记录除去最大最小值的平均值。

5.6.2 单表性能测试

测试 ParallelScan 算子对一张表的全表扫描的性能提升。

5.6.2.1 smallfile 测试

测试对象：表 smallfile0~10，使用 Q1 进行查询，分别统计总体的执行时间和 Scan 算子的消耗时间，结果如下：

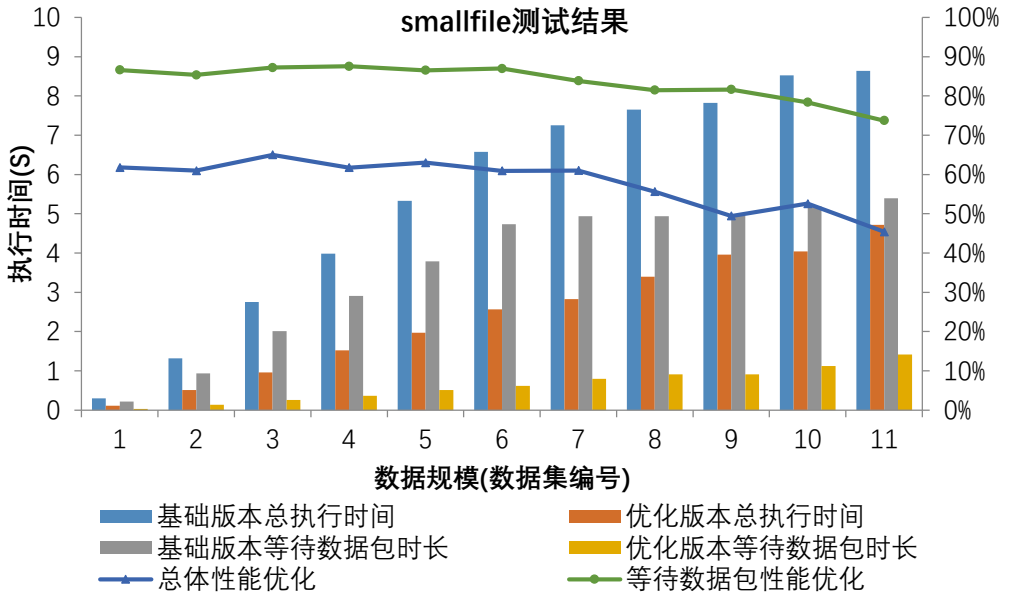

图 5.5 smallfile 测试结果

结果分析：在图 5.5 中，通过比较基础版本总执行时间和优化版本总执行时间，可以得出结论：在 smallfile 的测试中，rpc 切分之后可以将查询的总体性能提升 60%左右；通过比较基础版本等待数据包时长和优化版本等待数据包时长，可以得出结论：在 smallfile 测试中，查询中着重优化的数据扫描的性能提升 80%左右。

5.6.2.2 testfile 测试

测试对象：表 testfile0~4，使用 Q2 进行查询，分别统计总体的执行时间和 Scan 算子的消耗时间，结果如下：

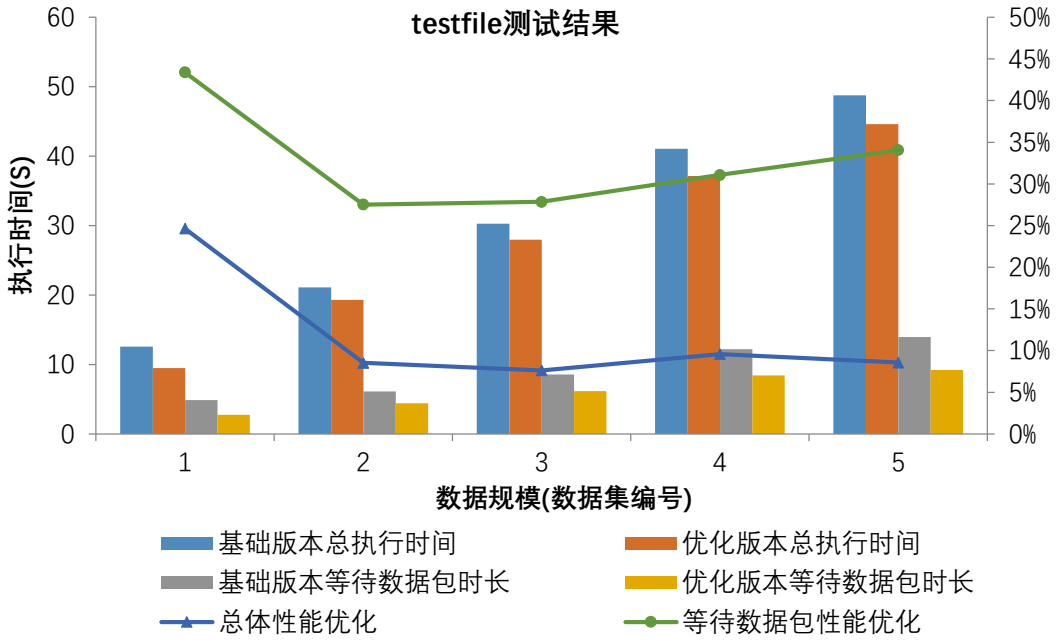

图 5.6 testfile 测试结果

结果分析：testfile 的测试当中数据规模的增长量是 100 万行。从上图可以看出在 testfile 测试当中，提升百分比跟数据规模没有必然联系，基本能够保持稳

定在 30%左右。在下一节的测试中将会分析较 smallfile 测试提升效果下降的原因。

5.6.2.3 bigfile 测试

测试对象：表 bigfile0~4，使用 Q3 和 Q4 进行查询，分别统计总体的执行时间和 Scan 算子的消耗时间，结果如下：

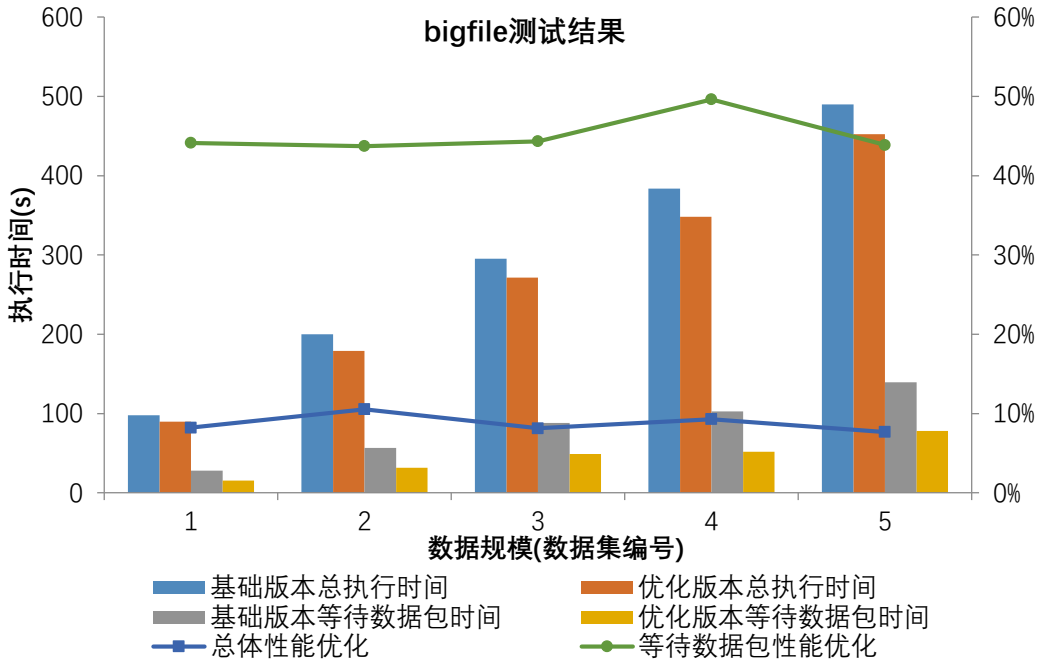

图 5.7 bigfile 测试结果

结果分析：表格 bigfile 系列中的每张数据表包含了大量的分片，例如 bigfile4 包含 199 个分片。对于基础版本而言，其并行度已经可以达到 199，此时，优化版本在针对 199 个分片进行再切分，将并行度从 199 提升至 597(3 倍)，此时相对于表格 smallfile 系列并未带来更大的性能提升。原因是，原来的并行度已经很高了，实验设置的 CS 系统资源有限，继续提高并行度，效果并不会再提升。为此，增加了下面的实验。

生产上一般很少出现大表全表扫描的情况，我们对 SQL 添加 where 条件，使其只查询最后一个分片范围上的数据。以下是测试结果：

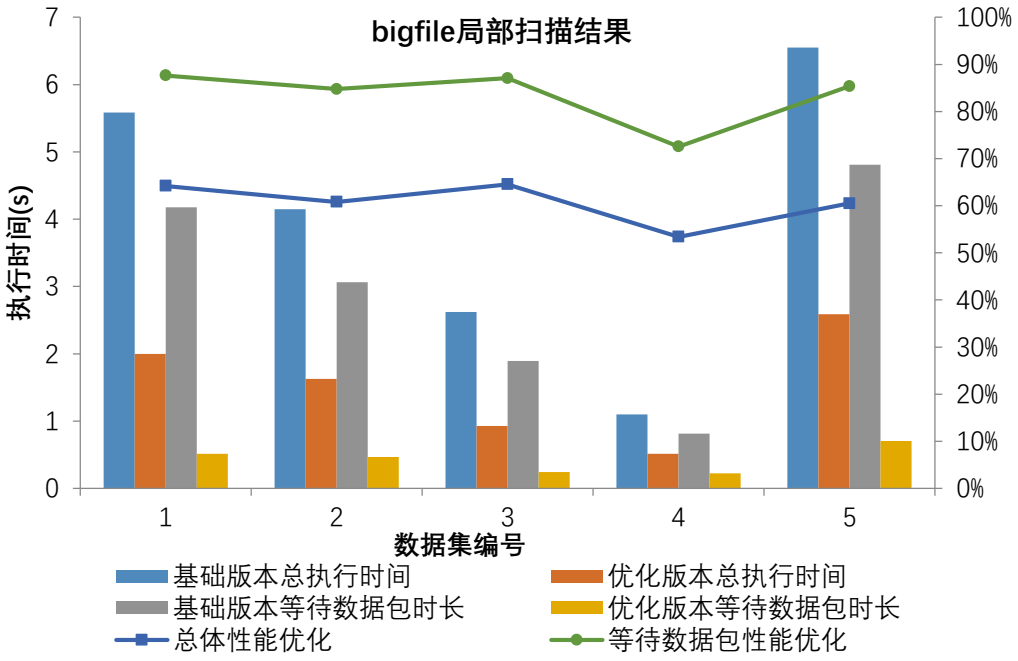

图 5.8 bigfile 扫描结果

bigfile 的查询测试结果与 smallfile 相似。无论是 MS 等待数据包的时长还是 Query 执行的总时长提升都比较明显，其中，等待数据包的性能提升为原来的 9 倍左右，总的查询性能提升为原来的 2.5 倍左右。

5.6.3 扩展性测试

本小节测试 ParallelScan 算子的可扩展性，实验过程中选取含有较多分片的 bigfile4 表作为测试对象，对比不同 CS 数目下的性能。

图 5.9 是大数量情况下的总体性能测试结果。从曲线可以看出来，较大数据量情况下（bigfile4 包含 199 个分片），查询性能是跟 CS 的数目是有关系的，随着 CS 数目的增多，查询时间有下降的趋势，也就是说查询性能变好。实验中发现在由 3 台 CS 变为 4 台 CS 的时候，bigfile4 的分片分布信息发生了变化（本文

采用动态增加 CS 的方式进行实验）：原来，所有分片的第一副本都在同一台 CS 上；当增加到 4 台 CS 的时候，某些分片的第一副本发生了迁移。

在基础版本中，对于 bigfile4 的 199 个子查询请求（按分片进行划分）是可以并行的，但是都会发送到一台 CS 上，CS 上的处理线程有限，并不能实现 199 个子查询完全并行。当副本发生迁移的时候，子请求会发送到另外的 CS 上，多台 CS 会增加 199 个子查询的并行性。所以查询性能在图中 3 台变为 4 台处会有一次提升。

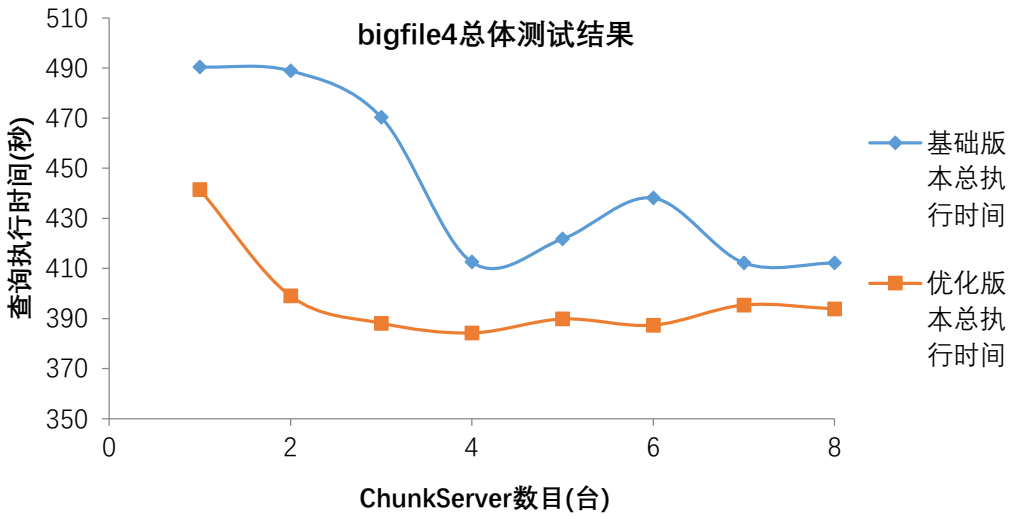

图 5.9 bigfile4 总体测试

随着 CS 数目的继续增多(从 4 台 CS 开始)，无论是基础版本还是优化版本，查询的性能逐渐趋于稳定。因为随着 CS 数目的增加，副本数目并不会继续增加。系统可配置的副本数目可选项为 1,2,3，默认值为 3。当集群中 CS 的数目少于配置的副本数时，实际副本数取值为 CS 的数目，也就是说，当集群中只有一台 CS 时，副本数为 1；有两台 CS 时，副本数为 2；3 台及 3 台以上 CS 时，副本数为 3。

实验过程中随着 CS 数目的增加，由于副本数目不再变化，副本分布信息会有一些小的变动，但此时查询基本稳定下来，包括查询哪些 CS，查询的并行度

等等。因此查询的性能随着 CS 数目的增多只有一些小程度的变化，基本稳定下来。

但是稳定下来的之后优化版本较基础版本的总执行时间提升仅有 17-50 秒左右，相对一个 400 多秒的查询，优化程度有限。

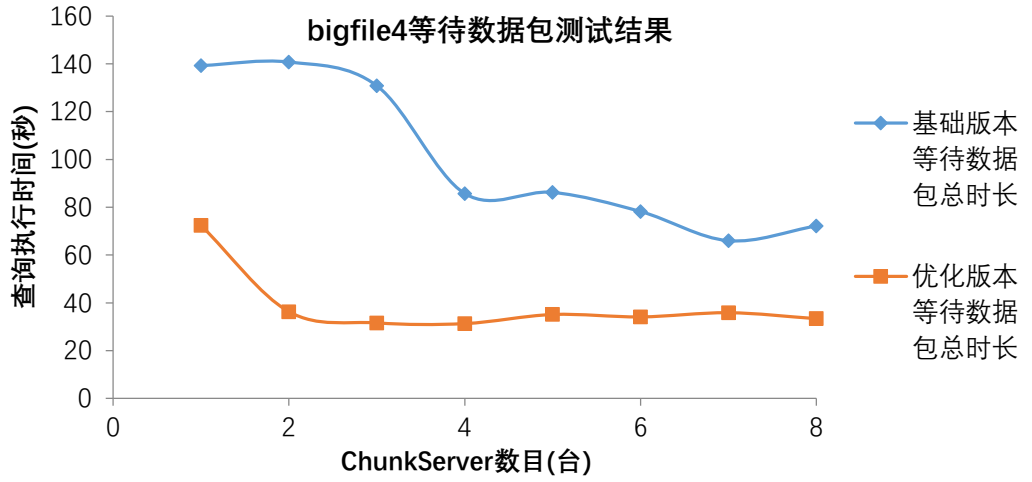

图 5.10 bigfile4 等待数据包测试

图 5.10 描绘了 bigfile4 测试过程中 Finish Queue 被阻塞的总时长。在 4 台 CS 之前，两条曲线随着 CS 数目的增多都呈现明显的下降趋势。稳定之后基础版本的等待总时长约为 78 秒，优化版本的等待总时长约为 35 秒，提升了 43 秒。

5.6.4 联表性能测试

本小节测试 ParallelScan 算子对于数据库中的联表操作(JOIN)的性能影响。本测试选择一条执行时间较长的含有 5 个 join 的 SQL，并将其拆分成为 2,3,4,5 和 6 张表 join 的情形进行比较测试。测试过程统计 Join 执行过程中各 Scan 算子的总的时间成本，如表 5.6 所示。总体性能是指整条 Query 执行的总时间，单位是秒。等待队列性能是指每张表的等待数据所耗费的时间，顺序为其 join 的先后顺序。单位为秒。

由测试数据分析，总体而言，ParallelScan 算子对于 MergeJoin 是会带来一定程度的性能提升的。其中，2 张表格的 join 测试中，由于第二张表带有等值的过滤条件，因此不能对查询范围进行切分，并未使用 ParallelScan 算子，所以 wait 部分时间均为 0.002 秒；而进行切分的第一张的 wait 时间由原来的 0.178 秒提升到 0.021 秒，提升了 0.157 秒。

表 5.6 Join 性能测试结果

参与 join 表数目	总体性能		等待队列性能	
	基础版本	优化版本	基础版 wait 时间	优化版 wait 时间
2	0.292	0.145	0.178+0.002	0.021+0.002
3	0.319	0.156	0.176+0.002+0.010	0.022+0.002+0.002
4	1.548	0.825	0.174+0.002+0.012+0.653	0.022+0.002+0.003+0.085
5	4.347	3.514	0.173+0.002+0.010+0.617+ 0.098	0.021+0.002+0.002+0.074+ 0.021
6	4.511	3.744	0.174+0.002+0.009+0.653+ 0.088+ 0.140	0.024+0.002+0.002+0.080+ 0.018+ 0.026

Join 表从 4 变为 5 的过程当中，总体执行时间从 1.5 秒提升至 4.3 秒，与此同时，wait 时间却只增加了 0.098 秒。尽管，使用 ParallelScan 算子能将这部分时间缩短为 0.02 秒。但是总体带来的提升却不大。这也是图 5.11 中，性能提升曲线急速下降的原因。这部分的时间可能需要另行优化，不在 ParallelScan 优化方案的讨论范围之内。

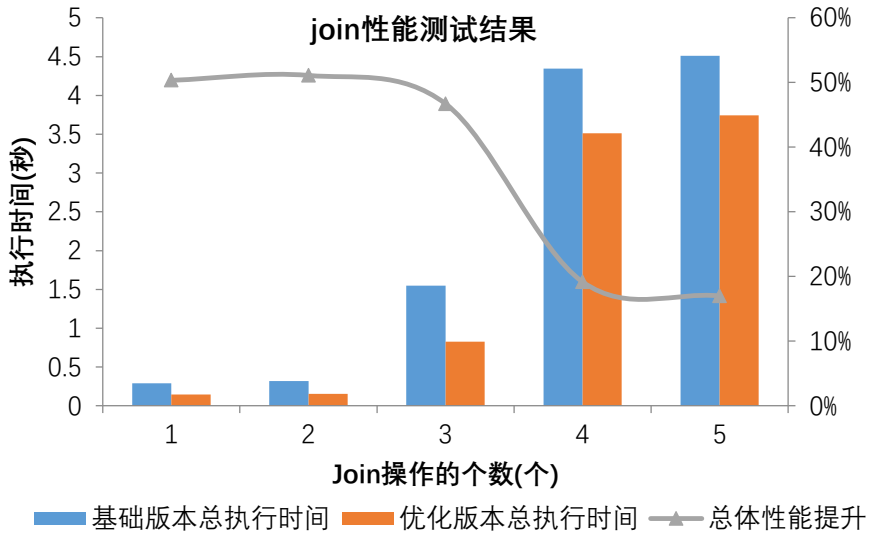

图 5.11 Join 性能测试结果

5.7 本章小结

本章描述了 ParallelScan 算子的设计实现。在对于 Cedar 原有的 Scan 算子进行分析的基础上，结合对 Cedar 系统架构的分析，提出了分片各副本之间保持强一致的观点，并根据此观点设计了 ParallelScan 算子。算子利用并行读取副本的策略，可以成倍提升 Scan 算子的效率。在性能分析小节，本文分析了 Query 执行的各部分时间的花费结构，虽然 Scan 算子的效率得到了提高，但是 Query 其余部分的时间仍保持不变，因此 Query 总体执行效率的提升并不能达到理想中情况。对于如何确定切分粒度，本文暂时也没有很好的结论，因此实现过程中，我们预留了设置切分度的接口。在未来的工作中，我们将结合实践效果和理论分析，来完善切分粒度值的设置方式。

第六章 自优化连接算子的选择策略

6.1 算子概述

论文[36]指出了 SQL 语言的核心 select-projection-join（SPJ）虽然是十分有用的，但是 SQL 几乎包纳了所有的可以想到的特征，这就使得弄明白 SQL 的各个语义细节变得异常困难。在这个背景下，SQL 调优就如同噩梦一般。正是因为如此，自优化成为数据库系统的发展目标之一。

自优化 Join 算子就是一种能够实现自我优化，使用一定策略，选择出最优化的算子搭配。免去了使用者优化 SQL 的代价。

在前文叙述 Cedar 读事务流程中，曾提到了 Cedar 是缺少查询优化模块的。想要得到性能较优的 SQL，必须要用户显式地修改 SQL 语句，包括调整 Join 表的顺序，使用 Hint，增加 Where 条件等等。这就使得用户必须拥有较高的水平，深厚的 SQL 知识，才能够写出性能较好的 SQL。自优化连接算子希望结合已经存在的 Join 算法，在统计信息的基础上，实现自我调优，输出一个最佳的查询计划。

目前，自优化连接算子中结合了基于规则和基于代价两种方式，对内部算子进行决策：

1. 缺乏主键时才考虑使用索引，并考虑索引不能覆盖查询时的代价
2. 根据集群的整体负载以及 MS 消费数据的速度来确定是否使用 ParallelScan 算子
3. 结合统计信息模块输出的表的相关信息决策使用哪种 Join 算法
4. 结合 join 左表的相关信息动态决策具体 filter 算子

一般在连接算子内部，时间花费最高的就是每一张表的 Scan 操作和 Join 算

法。因此在自调优的过程中，本文关注的两个重点即为 Scan 算子和 Join 算法。6.2~6.4 小节中，本文将具体讨论算子的选择策略。6.5 节实施了一些实验来验证每部分选择策略的正确性以及带来的性能提升。

6.2 IndexScan 和 Scan 算子的选择策略

在 Cedar 中，索引和原表的存储结构是一致的，为了优化查询，索引表除了包含原表主键和索引列之外，还会添加一些冗余列。如果查询中涉及到的列全部在索引表中，那么就称索引能够覆盖查询。当索引表不能覆盖查询时，那些不能访问到的数据就必须再次访问原表取得。

假设存在表 A = {a0, a1, a2, a3, a4, a5}，主键为 a0。我们在 a1 上建立索引表 B 并冗余 a2 列，那么根据 Cedar 设计 B = {a0，a1，a2}，并有如下查询 Q1：**select** a3 **from** A **where** a2 = 100，即查询 A 表上满足 a2 列等于 100 所有 a3 的值。一般情况下，Q1 使用 IndexScan 算子进行查询会大大提升查询速度。但是考虑一种情况，A 表中所有 a2 的值都是 100。对于列基数过小时，我们不建议在此列上建立索引，但是如果出现这种情况，不进行判断而直接选择索引算子就会导致对于从索引表每次查询出一行，得到原表主键之后，都会调用一次回访原表的流程，去取出 a3 的值。也就是说此时使用 IndexScan 算子将会造成 2 倍于 Scan 算子的访表代价。

利用统计信息计算该列上满足查询条件的值的选择率，可以避免出现这种问题。

在 Cedar 中对于是否使用索引表，遵循以下规则：

1. 当查询条件包含原表全部主键或者主键前缀时，不使用索引表。所谓主键前缀是指当主键包含多列时，查询条件能够按序覆盖主键的前 N 列（$1 \leq N \leq$ 主键总列数）。
2. 优先使用不需要回访原表的那些索引表。
3. 索引回表代价不能大于直接访问原表的代价。

抉择 IndexScan 还是 Scan 时，本文提出了规则和代价两方面的参考因素。一般情况下，通过查询索引的方式取得原表的效率都会高于对原表进行全表扫描。

在 Self-Tuning 算子中会对算子中涉及的每张表判断是否有可用的索引。对于满足索引使用规则的查询，均使用 IndexScan 算子替换 Scan 算子。

6.3 ParallelScan 的选择策略

MS 请求数据，CS 取数据是一个生产者和消费者模型，CS 返回数据，MS 消费数据也是一个生产者消费者模型。因此在评价 Scan 算子的执行效果时要考虑这 4 个方面的因素。

ParallelScan 算子原理是将请求数据的范围进行切分，同时发送到这个分片不同副本所在的 CS 上。如此做法，实际上是利用了 CS 并行处理。因此，在是否选择使用 ParallelScan 算子时，需要考虑两方面的因素：

1. CS 是否有足够多的取数据的线程来执行 MS 发送过去的子请求。
2. MS 消费数据的速度是否会成为瓶颈。当各个 CS 源源不断地将数据从 CS 返回到 MS 上时，MS 需要按照执行计划树一层层地将数据返回给调用者，当 MS 无法及时将数据返回时，数据会在 Scan 算子中积压，这种情况，CS 速度再快都是无益的。

因此在决策是否使用 ParallelScan 算子时，需要结合 CS 的负载和 MS 的消费速度两方面因素。前面讨论统计信息时提到有些系统会对系统负载进行统计，但是在 LSM 存储架构下，统计信息是在合并模块进行的，并且本文实现时为了减轻系统的压力，只对那些数据更新量达到一定程度的表进行统计信息的更新。这就会造成系统统计量无法及时更新的问题。

实际上，在预估 ParallelScan 算子的执行效果时，MS 将会访问一次 CS，询问此 CS 的可用线程数，CPU 利用率以及磁盘 IO 及网络 IO 的负载情况等信息。这样的做法会增加多次额外的网络访问。在我们提出更好的统计信息更新策略之后，CS 的负载可以直接从统计信息中获取。

目前，本文评价 MS 的消费数据速度的方法是综合查询列的属性，以及 MS 生成执行计划树 Scan 之上算子的数目两方面因素。之所以要考虑查询列的属性，是因为类似与字符串比较这种操作需要消耗更多 CPU 资源，处理速度也比较慢。因此，本文尽量避免在长字符串类型上使用 ParallelScan 算子。另一方面，之所以考虑 Scan 上层算子的多少，是因为每个算子都会对数据进行一定处理，处理会影响 MS 的消费速度，算子越多，MS 处理数据的速度越慢。但是由于每个算子功能的不同或者采用的算法不同，所以不能一致对待。为此，我们调研了 Cedar 中与查询相关的主要算子，并为其赋予了相应的权值。如表 6.1 所示，本文从两个角度：算法复杂度和是否有磁盘 IO 来确定算子的权值。

表 6.1 算子权值

算子名称	平均算法复杂度	是否访问磁盘	权值
Filter	O(N)	否	1
Limit	O(m)	否	1
Project	O(N)	否	1
Sort	O(NlogN)	否	2
		是	3
聚集函数	O(N)	否	1
Groupby	O(NlogN)	否	2
		是	3
Distinct	O(NlogN)	否	2
		是	3
MergeJoin	O(M+N)	否	1
Union/Intersect/Except	O(M+N)	否	1

6.4 InFilter 与 BtwFilter 的实现及选择策略

如图 3.3 所示，在半连接优化算法中提到了将左表的数据构造成为 filter，用于对右表进行过滤。在 Cedar 中，所有 filter 算子都是通过表达式形式进行构造的。

算法 6.1 描述了构造 filter 的过程。算法中，首先确定了根据表的列值的信息动态决策了使用算子的种类：如第 3 行所示。如果列的 distinct 值比较小，此时会选择使用 InFilter，InFilter 利用 in 表达式进行过滤。In 表达式也是 In 子查询底层的实现方式。当 In 的数据数目较少时，效率较高。当 distinct 的值很大的时候，使用 InFilter 算法效率并不高。此时利用 between…and…表达式构造 BtwFilter，btw 算子能够取回区间的所有 block，可以避免 In 表达式为了加载不同数据块的内存而产生的换入换出代价。遍历左表的所有的数据，去重后存入表达式对象中，然后根据 type 构建 InFilter 或者 BtwFilter 算子，并设置读取参数，发送给 CS。CS 会利用这些 filter 对数据进行过滤，可以减少了网络上数据的传输量。

Algorithm6.1 构造 filter 算法

输入：左表全部数据 left_table_Rows

功能：根据左表数据构造过滤算子传送给 CS

```
1  acquire_left_tid_cid(tid, cid) //获取表 id，Join 列 id
2  ExprItem expression; //定义表达式项
3  type = dynamic_choose_filter_type(left_table_Rows, tid, cid);
4  if(type == "in") :
5  |    expression.set_type(In);
6  else if(type == "btw") :
7  |    expression.set_type(btw);
8  else :
9  |    //更多的构造 filter 表达式的策略
```

```
10 for (auto Row : left_table_Rows) : //对于左表每行
11 |    cell = query_cell(tid, cid); //获取 Join 列该行的值
12 |    res = do_distinct(expression.data, cell.data); //去重
13 |    if(res) :
14 |    |    expression.add_constent(cell.data);
15 switch(type):
16 case "in" :
17 |    in(expression); break; //生成 InFilter 对象
18 case "btw":
19 |    btw(expression); break; //生成 BtwFilter 对象
20 default: break;
21 read_para_.add_filter(type=="in" ? in : btw);
22 rpc_from_MS_to_CS(read_para_);
```

6.5 实验

本文在 Cedar 系统上实现了自优化的连接算子，该算子以统计信息为基础，结合 IndexScan，ParallelScan 以及半连接优化算法等，可以对用户输入的包含 Join 操作的查询进行内部优化，执行效率更高。

6.5.1 实验环境

本节描述实验的集群全部搭建在 4 台服务器上，每台服务器的硬件配置如下表所示：

表 6.2 机器配置

配置项	配置说明
CPU	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
	24 核
	Cache：15680K
磁盘	3.6T
内存	169.431G
网卡	千兆以太网
操作系统	CentOS release 6.5(Final) 2.6.32-431.el6.x86_64

在 Cedar 中，RS 和 UPS 均采用的是双机热备技术来实现容错高可用，通常部署在一起，而 MS 和 CS 都是支持高可扩展的，通常部署在一起。如表 6.3 所示，实验将 RS 和 UPS 配置在同一台服务器上，另外三台均配置为 MS 和 CS，数据库的副本数目设置为 3。在集群规模比较小的情形下，机器故障的处理和恢复都比较简单，因此实验过程中并没有搭建 RS/UPS 的双机热备。

表 6.3 集群配置

RS/UPS	10.11.1.193
CS/MS	10.11.1.191，10.11.1.192，10.11.1.194

6.5.2 测试工具及数据集

本节使用雅虎的标准测试工具 YCSB(Yahoo! Cloud Serving Benchmark)[37]进行数据的生成，导入以及测试工作。

YCSB 的测试分为 load 和 run 两个阶段，内部接口为 doInsert 和 doTransaction，

分别完成数据的生成装入和负载的执行。YCSB 提供五个核心负载集 A-E，测试的内容为数据库的读写性能，主要用到的是主键查询，可以很方便地测试 NoSQL 系统以及一些关系型数据库。为了完成本节的测试内容，我们以 YCSB 为原型进行了一些扩展工作，主要包括以下两个方面：

1. 数据生成方面：增加了额外索引列 INDEX_FIELD，可配置索引列不同值的个数。
2. 负载方面：增加了非主键列的等值查询操作；增加了指定两张表的 Join 操作，Join 列为两张表达主键，为了简化实验，本文直接使用 Usertable 进行自连接。

表 6.4 YCSB 中用户表

列名称	是否是主键	数据类型	长度(Byte)
YCSB_KEY	是	Varchar	24
FIELD0…FIELD9	否	Varchar	100

YCSB 的所有测试都运行在表 Usertable 之上，其 schema 如表 6.5 所示。由表 6.5，YCSB 中一行的长度为 1KB，每行长度相等，因此，行数决定了表格的大小。在配置文件中设置 recordcount，可指定生成表格的大小。为了加速导入，可在每台服务器上并行 load，需要额外指定 insertstart 和 insertcount 参数以说明导入的起始值及导入行数。

6.5.3 实验结果及分析

6.5.3.1 IndexScan 性能测试

对于 Cedar 系统，所有的写入操作均需写入到 UPS 内存中。如前文所述，查询流程中，CS 将会访问 UPS 获取增量数据，这个过程对于 Scan、ParallelScan 以

及 IndexScan 消费的时间都是相同的。因此，在测试 IndexScan 性能时，本文选择了 YCSB 的核心工作负载 C，该负载为 100%读操作，数据量为 1000 万行，Query 执行总次数设置为 10 万次，readallfields 设置为 false，读取表格随机一列。测试开始前，进行一次手动合并，将数据转储到磁盘。

图 6.1 测试的是最适合建立索引的场景：生成索引列数据时，设置索引列的不重复值为 1000 万，从而保证了索引列与主键列为一对一的关系。实验过程中，在未创建索引表前，利用 YCSB 测试非主键查询时，时间超出了 YCSB 的超时时长限制，未能得出吞吐量结果。为此，我们通过对比 IndexScan 算子的性能与主键列的查询性能，从侧面反映 IndexScan 的性能。

由图 6.1 可知，在使用 IndexScan 之后，查询的性能能够达到主键查询的速度，相对与未能得出结果的 Scan 而言，有质的提升。因此只要能够在合适的列上建立索引，对于改善非主键列的查询非常有意义。

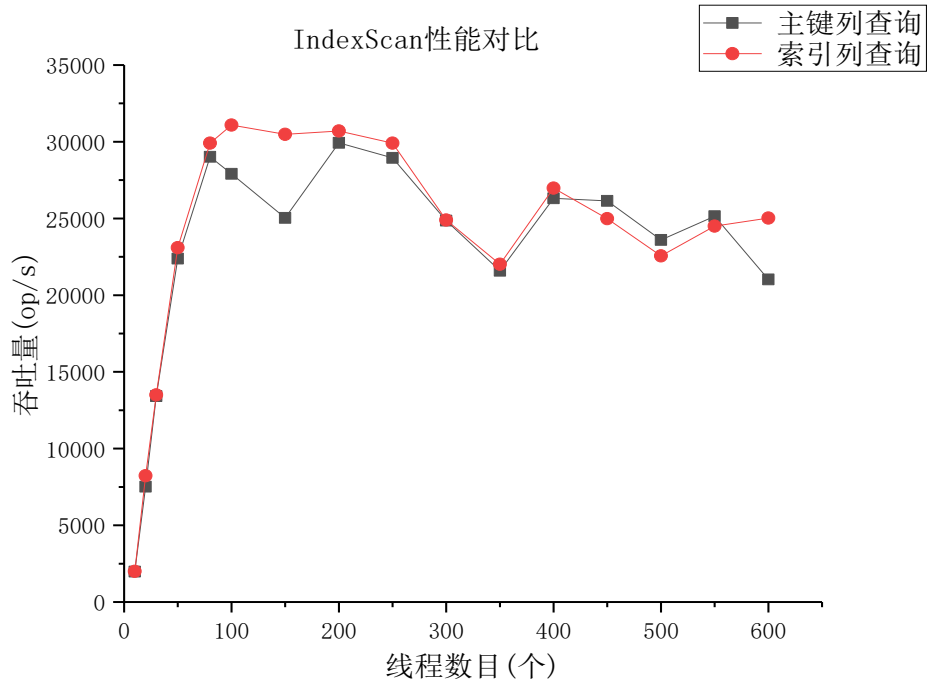

图 6.1 IndexScan 算子与主键查询性能对比

6.5.3.2 InFilter 和 BtwFilter 性能对比

InFilter 底层采用 In 表达式构造过滤条件，当 In 表达式中含有的数据量较少时性能较优；BtwFilter 底层采用 Between…And…表达式构造过滤范围，可用于一次过滤较大量的数据，在数据量大的时候性能较优。

本文采用 Usertable 与 Usertable 自连接的方式来测试 InFilter 以及 BtwFilter 含有不同数据量对于算子性能影响，Usertable 的大小为 100 万行。SQL 中，为左表增加 where 条件，来获取不同大小的数据集，右表不含任何过滤条件。Filter 中构造的每条数据都会选出右表中的一条数据。

在图 6.2 中对自变量进行了对数化，每个整数刻度分别对应 1 千，1 万，10 万以及 100 万行数据。

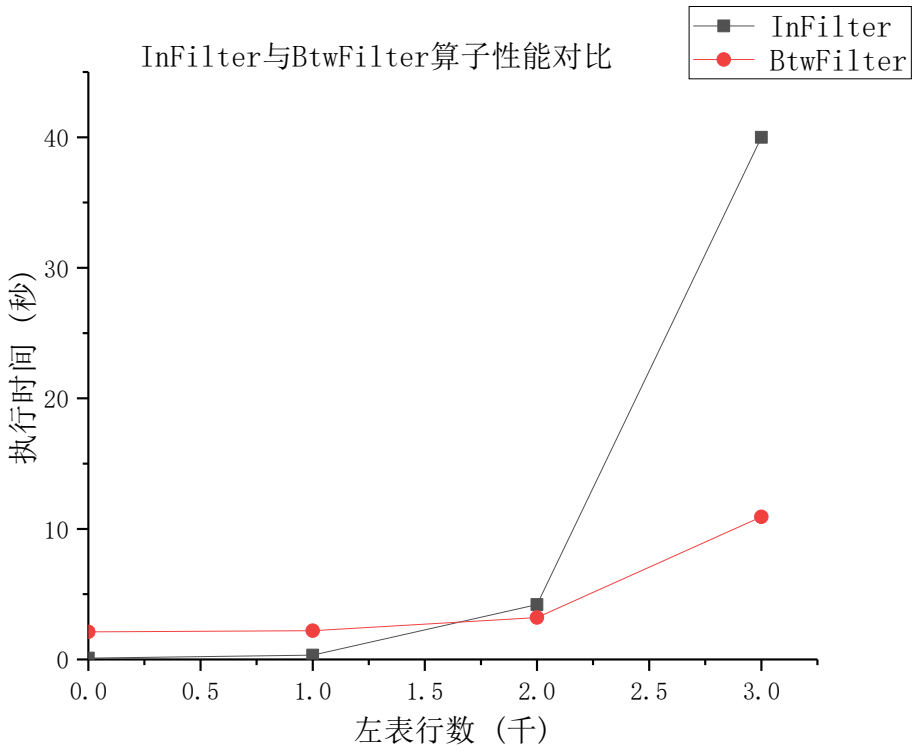

图 6.2 InFilter 与 BtwFilter 算子性能对比

由图 6.1 可见，随着数据量的增长，无论哪种算子的执行时间都会增长，在左表行数低于 1 万行时，InFilter 的执行效率比 BtwFilter 的效率要高，但是当随着左表行数继续增长时，InFilter 的执行效率会严重下降。这跟 Cedar 中 In 表达式的实现有关系：In 表达式对于其中的数据都会按每一项进行过滤，其执行时间复杂读为 O(n*m)，n 为 In 表达式中的数据量，m 为右表的数据量，因为 m 已经很大，所以 InFilter 在 Join 的左表较小的时候有较好的性能。而 Btw 会一次将这个范围内的所有数据块同时加载到内存中，在数据量较大时较 In 有更好的性能。

6.6 本章小结

本章介绍一种自优化 Join 算子的实现，自优化的手段既有来自于规则的，也有来自代价的，分别介绍了 IndexScan 和 Scan 的选择策略，叙述了 ParallelScan 可能的问题在于加倍消费了 CS 上资源，提高 Query 吞吐量的同时缺降低了系统的并发度。在系统负载较高时，仍然对查询使用 ParallelScan 将会增大系统的压力，严重时会导致一些 Query 无法被响应。因此在选择使用 ParallelScan 算子时一定预先查询 CS 的系统负载。Filter 是一种过滤器，能够运用一定的匹配规则，对数据进行过滤。在半连接 Join 算法中，本文封装了 InFilter 和 BtwFilter，结合不同的数据分布特点，可以使用不同 Filter 方法。

第七章 总结与展望

无论是最开始的网状数据库还是传统的关系型数据库抑或是异军突起的NoSQL系统又或者是回归SQL的NewSQL系统，数据库自诞生以来，就不断地完善自我，不断地向前发展。“应用驱动创新”，数据库的发展离不开业务的支持，也正式因为要不断解决人类生活生产上遇到的问题，数据库技术才成为了计算机史上最最亮眼的一颗新星。

如何高效率地进行查询一直以来都是数据库系统追求的目标之一。先辈们从软硬件的各个角度对提升查询效率进行了思考。本文总结了数据库中 Join 方面的查询优化工作，并结合Cedar系统，在提升查询效率方面作了一些微薄的工作。

现对本文作一个总结。

- 本文介绍了一款LSM存储引擎下的读写分离分布式数据库系统Cedar，并介绍了Cedar上的分布式索引模块和半连接优化算法。
- 结合在生产环境中遇到的CS取数据相对于网络传输时间过长的问题，仔细分析了Cedar原Scan算子的流程，利用副本可读性，实现了ParallelScan算子，缩短了从CS取数据的总时间。
- 设计并实现了基于每日合并的统计信息收集算法。
- 查询优化器相关工作离不开系统内部算子的支持，但是Cedar诞生初期，算子资源极度缺乏，是挡在通往查询优化器路上的一块大石头。本文设计实现了IndexScan算子，InFilter，BtwFilter算子等，为生成最优的查询计划提供了更多的可选性。
- 本文在实现了许多其他可供选择的算子基础之上，提出了一种自优化连接算子。此算子的设计是将数据库中查询优化理论运用到算子层面，解决了以前使用Cedar时必须显式SQL语句中指定Hint的问题。

- 设计实施了丰富的实验数据对各个算子的实际效果进行测评。

本文提到的算子的设计思想可以很容易应用到其他类似的读写分离式分布式数据库产品当中。

由于时间上的限制，本文在设计和实现过程不可避免的会有一些不完善的方面，如：本文实现的统计信息现阶段还没有固化到磁盘当中，当系统宕机，或者RS 重启，都会影响统计信息；基于半连接优化思想，本文将 Join 的小表构造成为 Filter 形式发送到右表，发送形式只有 In 和 Btw 形式，这无法覆盖所有的查询场景，Filter 的形式亟待优化；目前自优化连接算子只能优化其内部的执行计划树，我们仍然缺少一个系统的查询优化器。

实现查询优化的前提条件：一是，拥有统计信息，能够及时感应到数据的变化，并且拥有一定的容错性；二是，数据库当中针对不同场景实现了不同的算子，算子够丰富，查询时有更多选择性。Cedar 中这两方面都不够完善，在未来的工作中，我们会逐步完善 Cedar 系统。

参考文献

[1] Sears R, Ramakrishnan R. bLSM: a general purpose log structured merge tree[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012: 217-228.

[2] Cedar. ECNU DaSE. https://GitHub.com/daseECNU/Cedar. [Online; accessed 10-Oct-2017].

[3] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[4] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.

[5] Lakshman A, Malik P. Cassandra: a decentralized structured storage system [J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

[6] George L. HBase: the definitive guide: random access to your planet-size data[M]. " O'Reilly Media, Inc.", 2011.

[7] DeBrabant J, Arulraj J, Pavlo A, et al. A prolegomenon on OLTP database systems for non-volatile memory[J]. ADMS@ VLDB, 2014.

[8] Arulraj J, Pavlo A. How to Build a Non-Volatile Memory Database Management System[C]//Proceedings of the 2017 ACM International Conference on Management of Data. ACM, 2017: 1753-1758.

[9] Keeton K. The machine: An architecture for memory-centric computing[C]// Workshop on Runtime and Operating Systems for Supercomputers (ROSS).

2015.

[10] Bakkum P, Chakradhar S. Efficient data management for GPU databases[J]. High Performance Computing on Graphics Processing Units, 2012.

[11] Schadt E E, Linderman M D, Sorenson J, et al. Computational solutions to large-scale data management and analysis[J]. Nature reviews. Genetics, 2010, 11(9): 647.

[12] Wang K, Zhang K, Yuan Y, et al. Concurrent analytical Query processing with GPUs[J]. Proceedings of the VLDB Endowment, 2014, 7(11): 1011-1022.

[13] Soliman M A, Antova L, Raghavan V, et al. Orca: a modular Query optimizer architecture for big data[C]//Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014: 337-348.

[14] 樊秋实, 周敏奇, 周傲英. 基线与增量数据分离架构下的分布式连接算法[J]. 计算机学报, 2016, 39(10):2102-2113.

[15] 李智慧. 大型网站技术架构:核心原理与案例分析[M]. 电子工业出版社, 2013.

[16] 皮兴杰. 基于 Spark 的电网大数据统计中等值连接问题的优化及其应用[D]. 重庆大学, 2016.

[17] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]//ACM SIGOPS operating systems review. ACM, 2003, 37(5): 29-43.

[18] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[19] 宋纪成. 海量 RDF 数据存储与查询技术的研究与实现[D]. 北京工业大学, 2013.

[20] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Ab

adi, "H-Store: a High-Performance, Distributed Main Memory Transaction P rocessing System," Proc. VLDB Endow., vol. 1, iss. 2, pp. 1496-1499, 200 8.

[21] Shute J, Vingralek R, Samwel B, et al. F1: a distributed SQL database that scales[J]. Proceedings of the Vldb Endowment, 2013, 6(11):1068-1079.

[22] Corbett J C, Dean J, Epstein M, et al. Spanner: Google's globally dist ributed database[J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 8.

[23] Chang L, Wang Z, Ma T, et al. HAWQ: a massively parallel processin g SQL engine In hadoop[C]//Proceedings of the 2014 ACM SIGMOD inter national conference on Management of data. ACM, 2014: 1223-1234.

[24] Soliman M A, Antova L, Raghavan V, et al. Orca: a modular Query o ptimizer architecture for big data[C]//Proceedings of the 2014 ACM SIGM OD international conference on Management of data. ACM, 2014: 337-348.

[25] 么红月. 支付宝用户使用行为及满意度分析——以东三省为例[J]. 商场现代化, 2017(13).

[26] Yang Z K. The architecture of OceanBase relational database system[J]. 华东师范大学学报(自然科学版), 2014.

[27] Chakkappen S, Cruanes T, Dageville B, et al. Efficient and scalable sta tistics gathering for large databases In Oracle 11g[C]//Proceedings of the 20 08 ACM SIGMOD international conference on Management of data. ACM, 2008: 1053-1064.

[28] Schwartz B, Zaitsev P, Tkachenko V. High performance MySQL: optim ization, backups, and replication[M]. " O'Reilly Media, Inc.", 2012.

[29] PostgreSQL D. The Statistics Collector en PostgreSQL v9. 1[J]. https:// www.postgresql.org/docs/9.1/static/monitoring-stats.html

[30] Garcia-Molina H, Ullman J D, Widom J, et al. 数据库系统实现[J]. 2001.

[31] 樊秋实. 面向 OceanBase 的分布式大表连接与优化[D]. 华东师范大学, 2016.

[32] Huang G, Zhuang M Q. Scalable distributed storage of OceanBase[J]. 华东师范大学学报(自然科学版), 2014.

[33] 杨传辉. 大规模分布式存储系统:原理解析与架构实战[J]. 中国科技信息, 2013(19):140-140.

[34] Krikellas K, Cintra M, Viglas S. Scheduling threads for intra-Query parallelism on multicore processors[J]. EDBT, University of Edinburgh, Edinburgh, Scotland, 2010.

[35] Libeasy. http://www.cnblogs.com/foxmailed/archive/2013/02/17/2908180.html. [Online; accessed 10-Oct-2017].

[36] Chaudhuri S, Weikum G. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System[C]//VLDB. 2000: 1-10.

[37] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010: 143-154.

致　谢

时光如流水，本科毕业典礼上的情景还历历在目，而如今，我即将研究生毕业。回首在师大的这两年半时间，我收获的东西真是太多太多，包括学习，工作以及为人处事等各个方面。当然，要感谢的人也特别多。2015 年 7 月份，我来到华东师范大学。面对这个陌生的环境，是周傲英老师的谆谆教导指明了我在研究生阶段学习的总体方向，是高明老师和孔超学长带着我熟悉了新环境，并帮助我安顿下来。无论是在地理馆 314，还是在交通银行实习的日子，大家一起奋斗，一起总结工作，一起进步的日子总是如此难忘。值此，我要向那些关心，支持和帮助过我的老师，朋友和家人表示深深的感谢。

首先，我要感谢的是我的导师张召老师。当我接手到二级索引实现这项任务的时候，是张召老师统筹规划，指导我各项工作。在我遇到问题，疑惑的时候，和我一起想办法，帮助我解决问题；在我写论文的时候，为我审阅论文，提出修改意见。同时，我也要感谢蔡鹏老师和胡卉芪老师，在交通银行实习的日子中，无论是工作，还是生活，两位老师都给了我非常大的帮助，让我学会了团队协作，了解了工作是什么样子。

也要谢谢实验室中的其他老师们，谢谢你们对我的关心。谢谢钱卫宁老师发聋振聩的谈话，让我在实验性能不好的低谷期，重拾信心，继续坚持下去；谢谢张蓉老师在我心情低落时对我的安慰，谢谢您对我生活上的关心；谢谢周烜老师帮助我安排答辩的相关事宜。

另外，我还要感谢我身边的小伙伴们。感谢翁海星，樊秋实，张晨东和庞天泽学长，感谢他们带领我完成各种任务，帮助我理解代码，找问题，感谢他们在我找工作时提出的各种宝贵经验。感谢张燕飞，茅潇潇，肖冰，王嘉豪，胡爽，张春熙，谢谢你们对我学习生活上的帮助，包容我的缺点，陪伴我一路走来，谢

谢你们。谢谢金天阳，魏星，黄建伟以及王冬慧，感谢你们为系统组的开发工作做出的努力。

最后，我要特别感谢我的父母，没有你们，也就不可能有今天的我。谢谢你们对我的抚养之情，谢谢你们为我付出的努力，谢谢你们为我铺好我前进的道路，谢谢你们一直陪伴在我身边，做我最坚强的后盾，谢谢你们理解支持我的每一项决定。千言万语都无法表达我对你们的感恩之情，谢谢你们。

隆飞

二零一七年十一月十三日

发表论文和科研情况

■ 已发表或录用的论文

[1] 论文：隆 飞, 翁海星, 高 明, 张 召. 基于 LSM Tree 的分布式索引实现[J]. 华东师范大学学报(自然科学版), 2016, 2016(5): 36-44. LONG Fei, WENG Hai-xing, GAO Ming, ZHANG Zhao. Distributed secondary index based on LSM Tree. Journal of East China Normal University(Natural Sc, 2016, 2016(5): 36-44.

■ 参与的科研课题

[1]国家自然科学基金，集群环境下基于内存的高性能数据管理与分析，2014-2018，参加人

[2]国家高技术研究发展计划（863 计划）课题，基于内存计算的数据库管理系统研究与开发，2015-2017，参加人