2016届研究生硕士学位论文

分类号：_____ 学校代号: 10269

密　级：_____ 学　　号: 51131500039

華東師範大學

East China Normal University

硕士学位论文

MASTER'S DISSERTATION

论文题目：面向读写分离、批量更新存储机制的高效分布式索引

院　　系:	计算机科学与软件工程学院
专　　业:	软件工程
研 究 方 向:	数据库与数据挖掘
指 导 教 师:	宫学庆 教授
学位申请人:	翁海星

2016 年4 月6 日

Dissertation for master's degree in 2016

School Code: 10269

Student ID: 51131500039

East China Normal University

Title: A HIGH PERFORMANCE DISTRIBUTED INDEX DESIGNED FOR READ/WRITE ISOLATED &BATCH-UPDATE STORAGE

Department:	School of Computer Science and Software Engineering
Major:	Software Engineering
Research direction:	Database and Data Mining
Supervisor:	Prof. GONG Xueqing
Candidate:	WENG Haixing

April, 2016

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、批量更新存储机制的高效分布式索引》，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:_____ **日期:** 年 月 日

华东师范大学学位论文著作权使用声明

《面向读写分离、批量更新存储机制的高效分布式索引》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_____

年 月 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）。

翁海星 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
周傲英	教授	华东师范大学	主席
钱卫宁	教授	华东师范大学	
高明	副教授	华东师范大学	
蔡鹏	副教授	华东师范大学	
张召	副教授	华东师范大学	

摘　　要

基于日志结构合并树（Log-Structured Merge-Tree，LSM-tree）[1]思想实现的分布式数据库，采用读写分离、批量更新的存储架构管理数据。它能够有效地解决高负载的集中写入问题，并实现海量数据的可扩展存储。但这类架构的数据库在数据访问支持上还存在不足，其中不支持高性能的非主键查询是主要问题之一。

索引是传统的提高非主键查询性能的技术。当下，对于读写分离架构的数据库，创建二级索引主要挑战是，数据更新增量全部存储在内存中并动态更新，而磁盘又存储了大量的基线数据。分布式环境下如何实现统一而高效地基于这两部分数据的索引构建，并能够支持数据的动态更新，保证索引与数据的一致性，是本文所要面对的问题。本文针对这样的问题，提出一种面向读写分离存储架构的索引方法，实现对海量数据的高效索引构建，同时使索引适应于系统的负载均衡和可扩展性。本文工作的主要贡献如下：

1. 解决了读写分离架构下对海量数据高效索引构建的问题。设计了适用于分布式系统的底层存储结构，保证在动态更新状态下的索引正确性；设计了索引延迟生效机制，实现不阻塞事务的索引构建，提高数据库的可用性；设计了基于采样的两阶段排序和多线程并行调度算法，加快基线数据的索引创建。

2. 提出了读写分离架构下索引高效维护和查询优化策略。设计了该架构下索引的维护和查询算法，并使用移除索引恢复日志的方法优化索引的维护，以及通过冗余列避免回表查询的额外开销优化索引的查询处理。

3. 验证了在典型架构的数据库中本文所有方法的有效性和正确性。本文的索引方法在开源数据库OceanBase[2]实现，并开展充足的实验。用大量模拟实验评测各种性能指标。并且，该索引技术已经在某银行的UAT[3]环境中用真实业务场景测试。两种评测都充分证明了本文方法的有效性和正确性。

关键词:读写分离数据管理，分布式存储，分布式索引，二级索引，查询优化

Abstract

The distributed Data Base Management System(DBMS) based on Log-Structured Merge-Tree (LSM-tree), use read/write isolated and batch-update storage architecture to manage data. It can effectively solve the problems of high-workload, centralized write and has the capability of the scalable storage of massive data . But such DBMS of this architecture cannot meet the technical requirements for data access, such as high-performance non-primary key query.

Index is one of the traditional technologies adopted to improve the performance of the non-primary key query. At present, for the DBMS of read/write isolated architecture where massive baseline data are stored in the disks while incremental data are entirely maintained in the memory and updated dynamically, constructing secondary index is rather difficult. The challenges for us are how to efficiently construct index based on the two parts, satisfy the demands for dynamic update and guarantee the consistency of index and data. This paper mainly proposes the index algorithm for read/write isolated architecture which is adapted to load balancing and scalability of the system. The main contributions of this paper are:

1. Propose methods to support secondary index construction for massive data under the read/write isolated architecture. The structure of index with the underlying storage of distributed system is designed. It guarantees the correctness of index with dynamic update. Then, delay-effective mechanism is presented to avoid blocking transaction's process of constructing index, in this way, DBMS's availability is improved. And two-phase sort and multi-thread parallel dispatch algorithms are presented to speed up index construction of the baseline data.

2. Provide efficient index maintenance and query optimization strategies under the read/write isolated architecture. The algorithms of maintenance and query are designed. Index is maintained by the means of removing recovery log. And index query is optimized, by using redundant columns to avoid additional overhead of

querying original table.

3. Demonstrate the validity and the correctness of the index algorithms by implementing our methods on Oceanbase which is the typical database using read/write isolated architecture. The adequate experiments are carried out. In addition, this index technique has been executed in the one of domestic bank's UAT environment. By two evaluations, validity and correctness of the algorithms are both proved.

Key Words: *read/write isolated data management, distributed storage, distributed index, secondary index, query optimization*

目录

插图

表格

第一章　绪 论

1.1　研究背景

1.1.1　数据库系统的发展

总结数据库系统几十年的发展历史，可以发现其架构受到数据量的增长，以及计算机硬件技术的革新等的驱动而衍变。

Edgar Codd在1970年的论文中首次提出了关系模型的概念[4]，奠定了关系数据库的理论基础。关系模型具有严谨的数学基础，易于使用和理解，并且提供了结构化的查询语言（SQL），这使得非专业的开发人员也能熟练地使用数据库。因此，关系模型很快地取代了网状模型[5]和层次模型[6]，而以此为基础的关系数据库管理系统（Relational Database Management System, RDBMS）也得到了快速的发展。从1978年Oracle第一个版本的发布开始[7]，直到二十一世纪初，几乎所有的主流数据库都使用了关系模型[8]。

但是从2005年开始，许多新型架构的数据库相继出现，数据的存储和管理工作不再只由传统的集中式关系数据库承担。新型架构数据库的出现主要受到以下几个方面的影响：

1. **数据规模的高速增长。**信息社会步入高速发展时期，现代社会的数据量正在以持续、快速、海量的规模增长。根据国际数据公司（IDC）在2012年公布的白皮书的研究结果（如图1.1[9]），预计到2020 年，全球产生的数据总量将高达40ZB （泽字节，Zettabyte，1ZB=1024EB），相比于2010年增长了50倍，相当于全球所有人一天时间里每秒产生1.7MB的数据[9]。大数据的出现推动

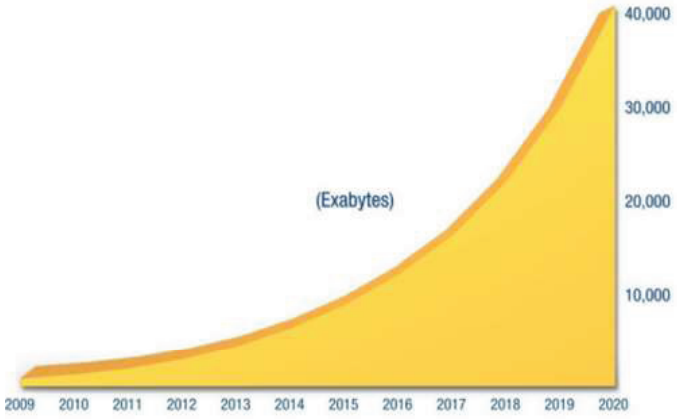

图 1.1: 2009年以来数据增长规模（IDC）

了集中式的数据存储向分布式转变，数据库的可扩展性(Scalability)也成为了研究的热点。

2. **多元化的数据分析需求。** 主要体现在两个方面：数据结构的多元化和应用场景的多元化。数据结构的多元化是指因为信息技术的进步和数据源的不断变化，对于流媒体、空间数据和XML 结构的数据的存储和分析变得愈加重要，诸如面向对象的数据管理系统（OODBMS）[10]和XML数据库[11]得到了重视与发展；而应用场景的多元指不同技术领域和企业的应用类型不尽相同，如银行金融企业的核心交易系统多使用部署在高性能服务器的商业数据库，而互联网企业的日志分析系统则多选择开源或者自主研发的分布式数据库。

3. **计算机硬件和技术的进步。**近年来，CPU的计算能力和存储容量在不断提升，多核技术的发展解决了过高CPU集成度带来的散热和能耗问题。内存价格不断下降，使得高性能内存的服务器成本降低，闪存等新型非易失性存储也发展迅速。计算机硬件的进步，使得高性能的内存数据库等技术应运而生。另一方面，无共享(Shared-nothing)架构[12]的计算模式盛行，不存在单机架构中的存储瓶颈问题，同时大幅提高了计算分析能力。人们逐渐倾向将数据管理系统部署在这样的架构之上。在无共享架构集群里，每个节点通过

网络相互连接，并独立运行操作系统，管理各自的计算资源，通过增加节点即可高效且方便地提升集群的读写性能和存储容量。基于无共享集群架构的BigTable[13]、HBase[14]都在工业实践中取得了瞩目的成就，被学术界广泛认可。

正因为如此，数据库系统架构的衍变具有持续、多元的特点。值得注意的是，传统的集中式关系数据库无法适用于所有场景，而新型的数据库架构中，也没有所谓的全面、通用的最佳方案。Mike Stonebraker在2005 年主张：根据实际负载，现代应用将不同类型的数据库进行混合优化，能够提供更好的数据服务。这个观点驳倒了之前一个数据库能够解决所有应用问题的“One Size Fits All”思想[15]。他的观点在现在看来是足够正确的。以Web应用为例，过去的Web服务总是基于LAMP（Linux，Apache，MySQL，PHP/Python）架构，但由于网络流媒体的发展和XML结构数据的广泛使用，产生了大量非结构化数据，而MySQL或其他结构化的新型数据库无法处理这类数据。一些应用被迁移到MongoDB[16]；以数据分析应用为例，数据分析经常会对大数据进行批量处理，需要高效的数据加载和聚合操作。这种场景下，列存数据库相比于行存数据库就更为适合一些。一种数据库的架构类型，通常有其独特的性质和最适合的应用场景。

在以上所述的背景下，读写分离、批量更新架构的分布式数据库应运而生。这类数据库具有良好的可扩展性，可以满足海量数据的分布式存储需求。同时充分利用了服务器内存资源以及无共享架构的优势，适应于高负载的集中写入应用场景，逐渐成为许多大数据应用的解决方案。

1.1.2 读写分离、批量更新架构的分布式数据库

现实中经常存在如下的业务应用场景：有高负载的集中写入需求，即数据的更新总是集中在某个时间段，写入数据有明显的峰值，且对写性能要求较高；同时，拥有海量的数据规模，一段时间内写入的数据与之前的数据相比，所占比重并不大。淘宝网站每日的电商交易，有明显的白天非常集中，晚上和凌晨相对很

少的数据写入特点，且对于在线业务几十亿、上百亿的数据规模，几千万到几亿的更新量是相对较少的。银行金融企业的历史库系统，只会在晚上将当天的数据抽取并导入到数据库中，一天的数据总量只在全年中占很小的比例。

对于此类应用，传统的集中式数据库显现出明显的不足。一方面，集中式单点数据管理数据，面对海量数据规模时存在着数据库扩容成本过高，集群规模增加困难以及管理开销大的问题；另一方面，由于传统的集中式数据库多采用B+树架构存储数据，随着数据规模的扩大，其随机写的性能会变差，无法满足应用需求。

如第1.1.1节所述，读写分离、批量更新架构的分布式数据库适用于具有海量数据规模的应用。数据分片、冗余副本、读写分离和批量更新是这类系统的四项基本要素，也是其能够满足应用需求的重要原因。下面本文对这四项基本要素进行简要介绍：

1. 数据分片。系统对数据按照主键划分，并分布存储在集群的各个节点，这是数据库系统可扩展的基础。

2. 冗余副本。系统通常使用成本较低的PC服务器，与大型高端服务器相比，节点故障率较高，多副本机制能够提高其可用性。

3. 读写分离。即在一段时间内，所有的数据更新都以追加日志的方式写入内存结构，数据通过读写分离被分为内存中的增量数据与磁盘中的基准数据两个部分。这实现了数据库随机写到顺序写的转化，提高了写入性能。

4. 批量更新。内存中的增量将与磁盘上的数据归并，进行批量的更新处理，充分利用了内存资源，保证当中的数据结构不会过大而影响性能。

因此，读写分离、批量更新架构的分布式数据库开始于工业实践中流行，在互联网企业的典型应用中逐渐广泛使用。一些国有企业出于自主可控的技术需求、国内“去IOE 化”的发展趋势[17]、以及降低大型机和商业数据库高昂的购入维护成本等原因，也开始将一些非核心的业务系统迁移到这类数据库上。

然而，这类数据库存在的问题也很明显。其中一项就是对数据访问的支持不够全面，灵活性不够，在一定程度上限制了可用性。数据分片是分布式数据库的重要特征，系统根据主键按照一定规则进行划分，将各个数据片段冗余存储在集群中的计算节点，并在主键上构建索引，因此能够提供高效的主键查询。但是对非主键的查询，分布式数据库因为无法确定满足条件的数据所在的节点以及数据分片信息，只能进行全表扫描。虽然分布式系统可以在多个计算节点使用并发技术扫描所有分片，但由此造成的时延依然是难以忍受的。因此，提高多维度查询性能，丰富查询功能，成为分布式数据库系统的研究热点之一。

1.1.3 读写分离架构下索引构建的挑战

索引是提高非主键查询性能的有效方法。在查询处理期间，优先查找索引的方式替代全表扫描，大幅度降低了响应延迟时间。传统的集中式数据库经过不断的发展和完善，已经拥有完整成熟的索引创建方法和维护机制。但由于读写分离、批量更新数据存储机制的特性，构建海量数据的高效索引具有挑战性，主要表现在：

1. 如何高效地实现对海量基线数据和增量数据进行索引构建，同时能够处理实时的增量更新，保证索引的正确性和系统的可用性。

2. 获得集群系统中增量数据和基线数据的数据分布信息是比较困难的，影响到索引在集群中的分布决策。

3. 构建出的分布式索引，不仅要满足高效的维护和查询处理需求，也要适应于分布式系统的负载均衡、高可扩展特性。

在新型架构的数据库上展开对索引技术的研究，已成为大数据管理领域的关注热点。分布式索引技术对于提高分布式数据管理系统的查询性能，拓展其应用领域，更好的实现对海量数据的存储和管理具有重要的研究价值和实际意义。本文主要针对读写分离、批量更新架构的数据库系统，设计与实现了一种高效、通用的索引方法，并能够应用到真实环境下业务系统的迁移工作。

1.2 本文工作

本文通过对分布式存储系统以及分布式索引技术的理论研究与实践探索，旨在设计适用于读写分离存储架构的二级索引LSM-Index（Log-Structured Merge-index），该方法在提升非主键数据访问性能的同时，降低索引的维护成本开销，也使索引能够满足可扩展性、负载均衡等分布式特性。本文在读写分离的分布式架构下作出了以下贡献:

1. **提出基于批量更新的高效索引构建方法。**主要包括：索引与底层存储的集成，令索引的内存组件维护创建索引期间发生的更新，保证了索引的正确性，且使索引和数据一样适应于系统的负载均衡和可扩展性；索引的延迟生效机制，基于批量更新对基准数据构建索引，只有这部分的索引构建成功后，才可与索引的内存组件共同用于查询处理，实现了不阻塞事务创建索引，提高了数据库可用性；基于采样的两阶段排序和并行任务调度，高效地完成全量基准数据的索引构建，并使索引数据在集群的均匀分布。

2. **制定读写分离架构下索引高效维护和查询优化的策略。**本文研究和分析了不同组织形式下，分布式索引的更新维护和跨节点查询的开销，设计了读写分离架构下LSM-Index 的维护和查询处理算法。并通过移除索引恢复日志的方法优化索引的维护处理，使用冗余列避免回表代价的方式在实际查询中进行了优化。

3. **验证了LSM-Index在典型架构数据库OceanBase上的有效性。**OceanBase在工业界是被广泛承认的高效读写分离数据库。本文在OceanBase上完成了LSM-Index 的实现，具有实际意义和工业价值，通过一系列基准测试以及真实企业的UAT环境测试，评测了索引的查询效率，以及对系统整体性能的影响，论证了索引方法的有效性。

1.3 本文结构

依据本文的主要研究工作，本文结构安排如下：

第二章介绍本文研究内容的相关工作。首先概述了读写分离、批量更新的分布式数据库系统，分析和论述其在某些应用查询方面的局限性。接下来讲解了分布式索引方法的技术现状，并指出现有研究工作中仍然存在的挑战。

第三章定义了读写分离架构下，创建分布式索引需要解决的问题。

第四章详细论述和分析了LSM-Index的设计与实现。首先概述本文的索引方法，对相关概念做了定义，并论述其核心架构与组织形式。然后，对LSM-Index的构建作了详细叙述，介绍了实现高效创建海量数据的索引的新批量更新算法。

第五章设计了LSM-Index在读写分离存储机制下的查询、维护的方法，介绍了移除索引恢复日志的维护优化和基于冗余列的查询优化方法。最后，论述了LSM-Index 的故障恢复场景。

第六章是实验部分。本文选择在开源的可扩展数据管理系统OceanBase上实现LSM-Index索引方法。首先介绍了实验环境以及实验参数的设置，然后描述了要进行的基准测试，最后论述并分析实验结果，并结合企业真实UAT环境的测试结果，论证索引方法的有效性。

第七章是对全文的总结，以及对未来研究工作的展望，并说明了分布式索引在今后研究还将面临的挑战。

第二章　相关工作

几十年来，数据库的架构类型受到海量增长的数据规模、多变的数据类型和业务需求、计算机硬件和技术的高速发展等多方面的因素驱动，以持续、多元的形式衍变。不同的架构类型的数据库有其自身的特性和适用场景，采用LSM-tree思想的读写分离、批量更新架构的分布式数据库，能够高效支持海量、集中的数据写入负载，同时拥有良好的可扩展性和高可用性。然而，这类数据库在非主键的查询应用上的性能很差，且因其读写分离的架构性质，在该系统上构建分布式索引存在着不同于传统数据库索引技术的挑战。本章节将介绍读写分离架构的分布式数据库及其分布式索引的技术现状。

2.1　读写分离架构的分布式数据库

2.1.1　分布式数据库技术概述

随着信息技术的革新和发展，传统的集中式数据库架构在许多应用场景下显现出局限性，主要表现在：

1. 集中式数据处理的性能瓶颈。由于单点服务器的处理性能始终有限，传统的集中式数据库无法承受高并发、高负载的数据写入和访问，虽然可以通过中间件实现备库单独处理读请求（Read Offloading），一定程度上降低负载，但主库仍然会出现性能瓶颈。

2. 可扩展和高可用方面的不足。为进一步提高性能，传统的集中数据库主要使用分库分表的方式扩展系统[18]，部署多个相同的数据库系统，不同应用的

数据存储在不同的库中，或者根据主键将数据划分到对应的数据库。这种方式存在着管理困难，扩展方式复杂的缺点，且无法满足高可用需求，当某个服务器出现故障，整个系统对应用程序提供的访问将受到影响。

3. 高昂的硬件成本。为了确保集中式数据管理的性能，以及降低可能发生的单点故障，企业通常会选择CPU性能较高、内存很大的服务器或大型机上部署集中式数据库，这极大增加了购入和维护这些硬件环境的成本开销。

而分布式数据库技术的出现，解决了传统的集中式数据库的这些缺陷，成为大数据处理的一种标准解决方案。这类数据库通常部署在无共享计算架构（shared-nothing architecture）上，多个独立的服务节点各自维护一部分数据，并通过高速网络通信，对外作为一个整体的数据库系统提供服务，拥有良好的可扩展性、高可用性和并行处理性能。分布式数据库种类存在许多不同的划分方式：

1. 按照数据处理应用划分，分为事务处理类型（OLTP）和分析处理类型（OLAP），前者注重实时事务处理，对并发性和事务完整性要求较高，后者对实时性要求不高，侧重于数据分析和决策支持。

2. 按照在磁盘上的存储格式划分，分为行存储与列存储数据库[19]两种。前者是常见的磁盘存储格式，适用于行级别的事务处理；后者对数据的相同列组织存储，有利于数据分析常用到的聚合、归并操作，常见的列存数据库有HBase、BigTable 等。

3. 按照集群的管理方式划分，分为主从模式（master-slave架构）和对等模式（P2P架构），主从模式通过主控节点管理、监控集群各节点状态，且负责管理数据在集群的分布信息，如HBase、VoltDB[20]；对等结构中每个节点都是平等的，通常使用一致性哈希[21]来决定数据分布和集群管理，如Canssandra[22] 和DynamoDB[23]。

4. 按照系统对事务的处理能力划分，分为NoSQL[24]、NewSQL[25]系统两种。前者无法支持满足ACID(Atomicity, Consistency, Isolation, Durability)[26]性

质的长事务，对跨节点的复杂查询存在局限性，但NoSQL数据库拥有灵活的数据组织方式，如键值对存储（Key-Value系统，如DynamoDB，CouchDB[27]等）、列存储；能够支持非结构化数据类型（如支持文档类型数据管理的MongoDB，图像数据存储的OrientDB[28] 等）。NewSQL 系统则能够保证事务的ACID属性，适用于结构化数据的事务处理，如Google Spanner[29]，VoltDB等。

5. 按照系统的存储架构划分，分为基于内存和面向外存两种架构。前者将所有的数据维护在内存中直接操作，即分布式内存数据库，如Geode[30]与VoltDB；后者对数据的存储和管理还是以磁盘等的非易失性存储为主，这类数据库存储和管理数据时，有三种数据的组织方式：基于B 树（及其变种）的存储方式，如MongoDB的MMAP存储引擎、CouchDB等；基于Hash的存储方式，如Riak[31]；以及基于LSM-tree的存储方式，如BigTable，HBase。

不同架构类型的分布式数据库，各有其特点及适用场景。与B树/B+树存储引擎相比，LSM-tree将更新进行延迟和批量处理，避免了随机写的开销，具有良好的插入性能。采用LSM-tree思想实现的分布式数据库，具有读写分离、批量更新的架构特点，并通过MVCC[32]（multiversion concurrency control，多版本并发控制）实现读可提交[33]的隔离级别，适用于频繁进行大量的集中写入的应用场景。因为互联网或其他企业的应用负载都存在着明显的写入高峰，对写性能的要求较高，所以基于LSM-tree 思想的数据库得到了广泛的应用。

2.1.2 读写分离、批量更新的存储机制

读写分离、批量更新的数据存储机制，核心思想是将一段时间内的写操作都在内存当中以追加日志的方式执行，记录成为增量数据，再通过批量更新的方式将增量数据写入磁盘，成为只读的基线数据，从而避免了随机写造成的磁盘额外开销，提升了写入性能，LSM-tree 是其实现的基本原理。

LSM-tree于1996年被提出，在Google公司采用其思想实现Big Table之后，LSM-tree得到了越来越多的研究关注。

如图2.2所示，LSM-tree由两部分组件构成，一个维护在内存中的树形结构（或者其他映射）以及数个写入到永久存储的B树（或者其他变种）结构。最简单的情况是由一个内存中的C_0树和一个磁盘上的C_1树组成。处理写入更新时，将记录项插入到维护在内存中的C_0 树，LSM-tree对所有类型的更新采用追加日志的方式写入。由于内存的成本和空间有限，当C_0 树因为更新操作而接近某个阈值大小时，就会启动一个称做滚动合并（rolling merge）的过程，将C_0 树中的部分数据（可以是全部）批量归并到磁盘中的C_1树上。处理对记录的查询请求时，将会在C_0树和C_1树一起查询，并将结果合并返回。

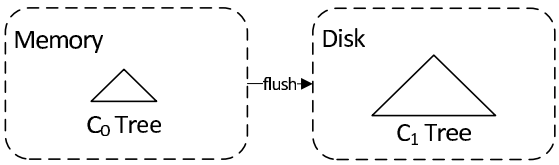

图 2.1: 两个组件的LSM-tree模型

当然，C_1树的规模增加时，会对批量更新的性能产生影响，这时就要在C_1树和C_0树之间再增加一个树，称为C_2树，并将C_0树的更新批量改写入C_2树。在数据不断增加的情况下，LSM-tree就需要多组件结构，由C_1...C_k树维护磁盘上的更新，明显地对于$k > 0$，C_k的数据版本较C_{k-1}树新，且C_k占用的空间小于等于C_{k-1}树。图2.2说明了多组件LSM-tree 的结构。

在实践中，LSM-tree在更新之前先要写用于恢复的操作日志，所以其更新的I/O代价为一次内存写和一次磁盘I/O。因为磁盘上存在多个批量更新后的数据，所以数据查询的时候代价为1次内存读取以及k次磁盘I/O，如图2.2.b。为了减少磁盘I/O代价，LSM-tree会定期将小树与大树进行压缩，如图2.2.c,压缩过后，C_2,C_3中不同版本的数据与C_1 合并成为最新的版本。

通过以上的分析可以得知，LSM-tree基于读写分离和批量更新，能避免随机

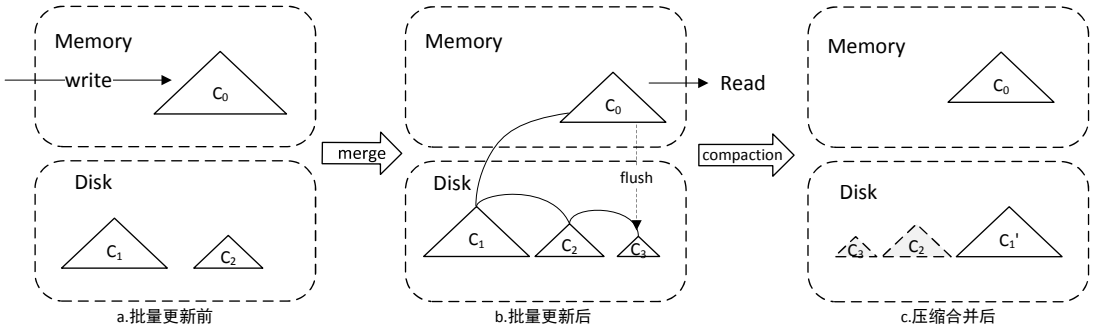

图 2.2: 多个组件的LSM-tree模型

写造成的硬盘磁盘臂开销，在持续的时间内提供对数据的高速更新（插入或删除）。然而在查询记录需要快速响应时性能不佳，因为其要在磁盘上读取多个版本的数据与内存的增量合并。所以，LSM-tree 通常适用于数据写入性能要求较高的场景，对于记录查询，可以通过滚动合并和批量更新的方式，以充分利用内存来维护近期或者常用的数据，减少对其查询而产生的基线数据的访问。

如小节1.1.2中所讨论的，因为现今的互联网或其他应用领域，许多业务的数据写入场景比较集中，存在着明显的写入峰值，如电商网站的秒杀活动、每天的库存记录等，且这类应用的查询场景中查询近期一段时间内的数据较为常见，如秒杀活动期间的交易量查询，库存记录中一天时间的出货/进货量等，因此采用这种架构的分布式数据库逐渐成为处理大数据的热门解决方案。

工业实践中，有两种方式维护内存中的增量更新（本文会在章节5.2详述），一种是增量数据在多个服务器节点中维护，与基线数据处在相同的计算节点，如HBase，BigTable等；一种是使用单独的计算节点存储增量数据，整个系统实现单点写入，这种方式带来的好处是能避免分布式事务，从而实现满足ACID性质的事务处理，如文献[34]实现的Deuteronomy系统，以及国内大型互联网企业阿里巴巴使用的OceanBase 等。

2.1.3　OceanBase 高效的分布式关系数据库

OceanBase是一种具有可扩展、高可用、高可靠以及低成本特性的关系数据库，在2015年阿里巴巴公司双11活动中支撑了14万笔/秒的支付宝在线交易，成为国内数据库的关注热点。本文选择在OceanBase 的开源版本上实现本文设计的索引方法。其原因主要有：

1. OceanBase是典型的读写分离、批量更新存储架构的数据库。具有分布式存储系统和LSM-tree思想的鲜明特点，选择其作为实现原型比较有代表性。

2. OceanBase是国内外工业界广泛承认的高效数据库系统，除了阿里巴巴公司正在使用外，某大型国有银行企业以及物流企业公司也采用了该数据库，具有广泛的应用场景，在此系统上实现具有实际意义和工业价值。

3. 本文针对采用读写分离、批量更新的存储架构，具有通用性，在OceanBase上实现之后，也能依据其原理和方法在其他系统上实现。

本小节将概述OceanBase的系统架构与实现原理。OceanBase在设计上将数据划分为两个部分维护：基准数据按照记录主键分片，冗余分布在集群系统内的不同节点，本质上是写入外部存储的静态B+树结构（称为SSTable）；将持续时间内的数据增量修改全部维护在内存当中，称为增量数据。在内存中以B+ 树和哈希散列的方式构建主键索引，内存中的数据结构称为内存表（MemTable）。因为要支持跨行跨表事务，保证数据的一致性，避免出现分布式事务，OceanBase选择单点维护增量更新，实现了集中化写事务、分布式读事务的事务处理方式。

Oceanbase数据库系统由四种类型的节点服务器组成：

- RootServer：主控节点服务器。实现Oceanbase的数据分布管理、副本管理以及集群节点状态管理等功能。

- UpdateServer：内存数据库引擎。实现事务处理，并存储维护一段时间内的数据增量。

- ChunkServer：数据存储服务器。提供分布式数据存储服务，存储基准数据。
- MergeServer：查询处理服务器。实现SQL解析，生成并执行查询计划，将所有节点的查询结果进行合并并返回客户端。

处理读请求时，OceanBase将数据以基线数据与增量数据合并后作为查询的返回结果。MergeServer解析查询的SQL语句，生成对应的查询执行计划，根据缓存的数据分布信息（向RootServer请求并缓存）将请求分发到数据所在的各个ChunkServer节点并发执行；ChunkServer在收到请求后根据查询的主键范围向MemTable请求数据的增量更新，合并后返回给MergeServer；将收到所有ChunkServer的结果合并，MergeServer就得到了最终的查询结果。

图2.3和图2.4分别描述了Oceanbase处理读请求和写请求的流程。

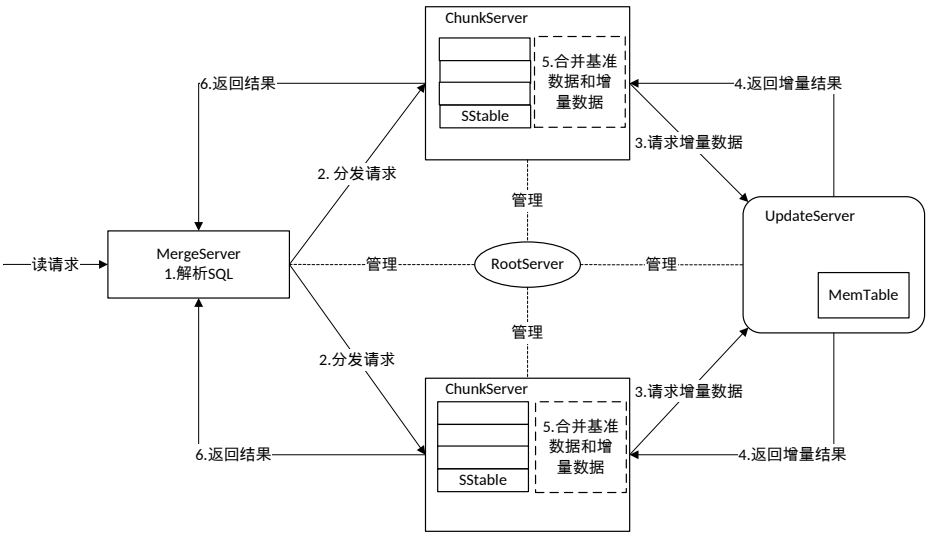

图 2.3: OceanBase读请求处理模型

处理写事务时，如图2.4.a，依旧由MergeServer解析SQL生成物理计划，并读取基线数据用于判断更新的数据行是否存在（由于REPLACE 语句有覆盖原数据的性质，这个步骤不需要），将返回的基线数据和执行计划发给UpdateServer，直接修改内存表MemTable。内存表的数据结构如图2.4.b所示，在内存中依据记录主键构建Hash索引和B+树索引，内存表的节点为主键构成的行头及该行的数据项操

作链表，行操作链表记录了以一个数据项为单位的不同版本的修改操作，分为已提交链表和未提交链表两个部分。如图2.4.c，修改MemTable时，需要锁住修改的数据行，将最新的数据增量操作追加到改行的未提交链表部分，然后往任务队列中加入一个提交事务的任务，这部分过程称为预提交，由多线程并发执行。之后，使用单线程处理提交队列，根据提交任务生成操作日志（用于故障恢复）并刷盘，成功后将未提交链表中的操作加入到已提交链表末尾，最后释放锁，完成写事务提交。

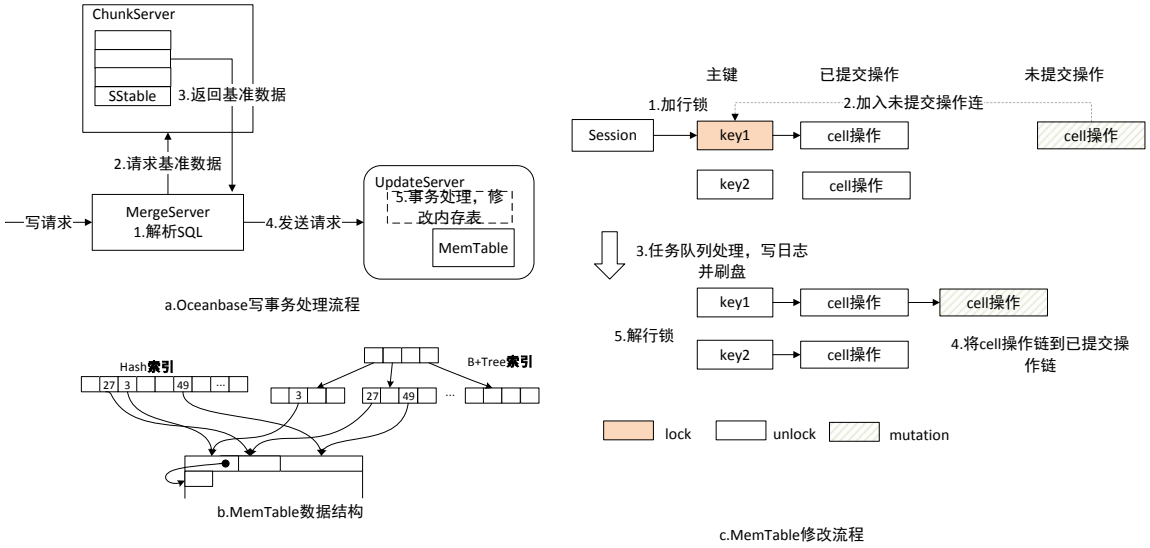

图 2.4: OceanBase写请求处理模型

UpdateServer的内存是有限的，随着数据增量的增长，需要将内存表的数据写回到ChunkServer的磁盘存储，这个过程在OceanBase系统中称为合并。如图2.5，数据合并由UpdateServer 发起，首先将停止对当前内存表的修改，将当前的内存表写入UpdateServer的外部存储，并加上版本号标记，同时使用更新的内存表处理随后的更新，这就是对内存表的冻结过程。随后，将冻结的版本号发送给RootServer，RootServer在接收到新的冻结版本号后通知所有存活的ChunkServer发起合并。各个ChunkServer 请求冻结内存表中符合本机sstable的增量数据，合并生成最新版本的sstable，并向RootServer 汇报更新之后的数据分布。

OceanBase采用读写分离、批量更新的架构，将UpdateServer上的修改增量定期写入到多台ChunkServer存储服务器当中，既防止了内存成为瓶颈，也实现了良好的扩展性。将写事务处理都集中在单点UpdateServer 上，避免了分布式事务，从而高效地保证跨行跨表事务的实现。在硬件（内存、网卡、带缓存的RAID卡等）和系统（成组提交减少I/O）等也做了优化，防止UpdateServer单点限制OceanBase集群的整体读写性能。

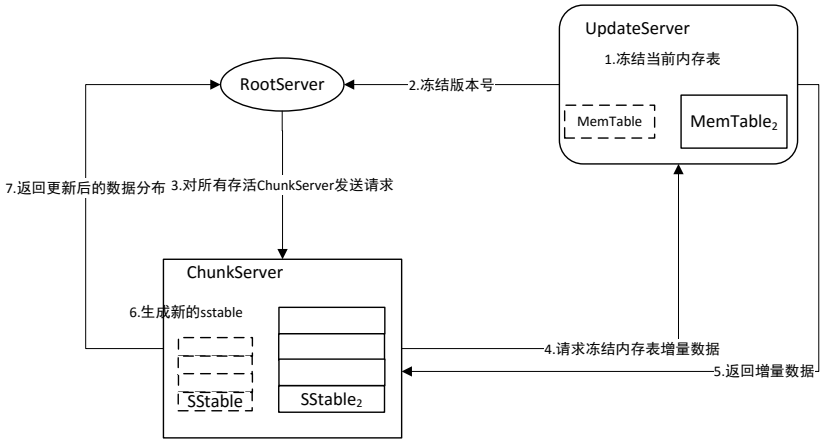

图 2.5: OceanBase批量更新流程

2.2 分布式索引技术

2.2.1 非主键查询的问题

因为读写分离、批量更新架构的分布式数据库既能支持海量集中的写入负载，也具有分布式系统的高可用和可扩展性。如章节1.1.2所指出，这类系统逐渐在工业实践中得到广泛使用，互联网企业需要借由这类系统支撑应用规模，国有企业需要通过迁移系统逐步达到技术可控和降低成本的目的。此时，企业应用场景不得不面对分布式数据库在非主键查询上的挑战。

这类数据库系统为了保证高可扩展性、高可用性以及高容错性，将数据按照主键(rowkey)进行分片，并冗余存储在集群中的多个节点，建立针对主键的索引结构，因此能够提供高效的主键属性查询。然而对于非主键属性的查询，分布式数

据库只能扫描所有的数据分片，尽管能够使用并发的方式进行，相比于主键查询仍有着很高的时间延迟。出于性能的考虑，该类数据库多数不建议进行非主键属性的查询。而实际应用中，对于非主键的查询是无法避免的，在业务中占着十分重要的地位，例如，根据实际调查，国有银行企业的某个系统中非主键查询的业务占了60%，大型物流企业公司的历史报表系统中非主键查询占了42%。

以银行企业中常见的历史库交易流水系统为例，其表格schema信息见表2.1

表 2.1: 历史库交易流水表（简易）

字段	BUSINESS_ID	TRX_DATE	ORG_NO	CUS_NO	BUSI_STATE
类型	varchar	date	int	int	varchar
含义	交易流水号	交易日期	交易机构号	客户号	交易状态
是否为主键	是	否	否	否	否

若系统要求查询某个客户或者某个交易机构当天的流水信息，由于分布式系统只能根据主键BUSINESS_ID分片和创建主键索引，对于ORG_NO 和CUS_NO 需要进行全表扫描才能返回结果，极大地影响了业务效率。

针对这个问题，通常互联网或银行企业采用两种解决方案：

1. 业务改造。将经常查询的属性提升为主键或者主键前缀，以利用高效的主键查询效率。

2. 使用中间表。利用ETL[35]工具，定期将数据中的表经过处理导入到中间表，中间表使用查询的属性作为主键，业务通过查询中间表替代全表扫描。

上述两种方案都不能彻底解决问题，存在着业务改造无法满足全部需求、ETL工具导入时间及管理成本过高的问题。因此，在非主键属性上建立二级索引的需求变得非常重要。

2.2.2　分布式索引技术现状及挑战

为了丰富分布式数据管理系统的查询功能，提高数据查询效率，国内外开展了大量关于分布式索引技术的研究，并提出了适应于各种系统和应用场景的实现

方案。现有的二级索引解决方案，主要分为客户端管理的索引方法和服务端内实现的索引方法两种。

把索引的实现全部交由应用层管理，其典型方案是通过客户端将需要查询的信息在系统中创建一个（或者多个）能够查找的映射表，将映射视作一种特殊的数据存在数据库的表中，并在客户端中实现映射表(也可以称为索引表)的维护和查询的逻辑。即每当程序写入数据时，也同时将更新同步到映射表，查询数据的时候，客户端先查找映射表得到原表记录项的主键，再用主键读取实际数据。

这种索引方法既有优点也有缺陷。其优势在于实现简单，且整个逻辑都由客户端的代码实现，应用层可以按照各式的需求来设计映射。但是这样会存在很多缺陷，对于NoSQL 数据库来说，跨行的事务不能保证原子性，在数据写入系统后且索引更新成功前的这段时间，发生了任何导致失败的错误都会使索引和数据的信息不一致；并且也会出现在高并发场景下数据和索引更新执行顺序不同导致的异常。因此，这种索引方案通常只用于非即时的离线应用，如银行金融企业历史库系统，每天定时将当日的数据全量导入系统，之后不会再有其他的更新，只用于查询和分析。另一个缺点是应用层在设计映射的同时，还要将查询数据的方法全部实现，应用程序的代码要因为需求的更改而变动，灵活性十分有限。

在分布式系统服务端内实现的二级索引方案，现今大致都分为三种类型：局部索引、全局索引和混合索引方案。

局部索引方案面向每个节点维护的数据分片，建立了查询属性上的局部排序索引，各个局部索引之间相互独立。局部索引的优势在于索引维护得到一定简化，但查询需要访问所有节点，造成不必要的资源开销（这将在章节中4.1.2论述），典型的实现方案有HIndex 和分片位图索引。

HIndex[36]是华为公司实现的典型局部索引方案，基于HBase系统实现，使用协处理器（coprocessors）在与数据分片相同的节点上维护了局部索引，在维护数据的同时修改局部索引，为保证索引和数据在同样的节点，Hindex禁止了局部索引文件可能发生的迁移、分裂，因此，其索引的实现无法满足自动的负载均衡及

可扩展性，而这些正是分布式系统的重要特性。而分片位图[37]索引将过滤条件生成为条件位图，并与所有节点维护的局部位图进行位运算，加快了节点内的查询。

全局索引方案中，索引信息根据查询属性在进行全局的排序后，才分布到各个节点，各节点管理的并不仅是本地数据分片的索引。全局索引能够避免访问无用的节点，提高了查询效率，但相对的，维护成本相对较高（在章节4.1.2分析）。文献[38]的作者提出全局索引方案CC-Index（Complemental Clustering Index）,CC-Index将需要查询的数据详细信息也存入全局索引中，减少了第二次访问的开销，但这样做会带来存储空间上的增长，于是文献[38] 的作者将数据库系统的备份数设为1以减少存储空间开销，备份减少会影响到分布式数据库的容错性，CC-Index使用互补校验表（Complemental Check Table）代替表的备份实现容错和恢复。

Diff-Index（Different-schemes Index）[39]是着力解决索引与数据一致性问题的全局索引方案。作者认为在不同互联网的应用当中索引和数据的一致性要求存在着差异，允许索引和数据异步更新的情况存在。针对不同的一致性需求，Diff-Index在HBase上实现了不同级别的索引一致性维护方法。

Sattam Alsubaiee提出的基于LSM-tree的混合异构索引[40]（multiple heterogeneous LSM-based indexes），是在其项目的系统AsterixDB[41]上实现的全局排序索引，作者用了多类型组件（LSM R-tree和LSM B+-tree）来分别记录索引的插入操作和删除操作，且使用混合组件实现了索引信息的bulk-loading。

混合索引方案结合了局部索引和全局索引各自的优点，其基本思想是每个节点首先维护局部索引，并在全局构建一个索引在集群中的分布信息，查询时首先确定正确的局部索引所在节点，再直接查找需要的记录，如CG-Index（Cloud Global Index）[42]。CG-Index通过BATON[43] 协议维护各节点的局部B+树分布信息，避免访问全部节点造成的额外开销，与Hindex类似，这种方案也无法实现索引的可扩展性和负载均衡。

如上所述，现今分布式索引的局部方案、全局方案与混合方案，各自具有不同的组织方式，但对于读写分离、批量更新架构的分布式数据库系统，这些方案都没有很好地解决如下问题：读写分离、批量更新架构的分布式系统环境下，如何针对已有的海量数据高效地创建索引。这类分布式数据库中，更新的数据增量全部维护在内存当中，而磁盘又存储了大量的基线数据，因此难于兼顾分布式基线数据的索引构建，和内存数据随时可能发生的更新。以上的方案并没有系统完善的解决方案，通常的做法是在创建表的时候预先定义了索引的结构，并让索引和数据表共同创建。Sattam Alsubaiee的方案虽然提出了一种bulk-loading的实现，但该方案是针对特定的系统AsterixDB，且使用多种数据结构用来构成索引组件，对实现索引的维护、负载均衡和故障恢复等增加了复杂度。

本文针对读写分离、批量更新的存储机制，实现了不锁表创建海量数据的索引构建、维护和查询算法，并在分析维护和查询代价的基础上做出优化，该索引能适应于原系统的可扩展性和负载均衡。在下一章节中，本文将对读写分离架构下，创建索引需要解决的问题做出定义。

2.3 本章小结

本章主要对本文的相关工作进行描述。首先概述了分布式数据库技术，介绍读写分离、批量更新架构的基本原理和典型实现（OceanBase 数据库），并指出其在非主键查询上的性能缺陷。接着介绍了分布式索引技术现状，而在现今的研究工作中，仍存在着海量数据规模下索引的高效创建困难、索引难于适应分布式系统负载均衡、高可扩展性的问题，这也正是本文的研究内容之一。

第三章　问题描述

3.1　基本定义

采用LSM-tree思想的存储架构的数据管理系统，无论是NoSQL数据库，还是可扩展的关系型数据库(如OceanBase)，通过提供高可用性以及高可扩展性，为大数据应用带来便利的同时，也存在明显的问题，例如对数据的查询方式支持得不够全面，缺少足够的灵活性，对于非主键的数据访问方式，其性能远不如主键查询。

定义3.1.1. 数据形式：假设有一张表格T，包含一个数据集合S，集合的空间大小为n，集合S中的数据定义如下：

$$(K_1, R_1), (K_2, R_2), ..., (K_n, R_n).$$

其中K_i是表格T中的每一项记录的主键属性，而R_i表示与K_i相关联的其余的属性信息，$(K_i, R_i) \in S$，$1 \leq i \leq n$。

分布式数据库为了实现可扩展和高可用，通常会将记录根据主键进行分片(sharding)，不同的分片冗余存储在不同的节点上。

定义3.1.2. 数据分片：假设有m个服务器节点，记为$C_1, C_2, ..., C_m$。数据的分片定义为，将数据集合S，依据一个映射函数：

$$F : S \to \{S_1, S_2...S_k\}$$

划分成k个子集，使得$S_1 \bigcup S_2... \bigcup S_k = S$且数据子集$S_i$被放在节点$C_j$上,其中$i,j,k$满足$1 \leq i \leq k, 1 \leq j \leq m, k = 1, 2, 3...$

映射函数F与主键K相关，常用的算法有对主键进行哈希计算，或者全局排序两种。通过数据分片和分布存储，集群中的每一个节点都存储并管理了系统的一部分数据。一般说来，分布式数据库系统都采用多副本冗余存储，当某个节点发生故障而下线时，该节点上的数据都能在其它节点找到对应副本，数据仍然可以被查询到，整个系统依然是可用的。

集群中的每个节点内部都会对数据分片构建主键上的索引，而数据的分片信息通常由一个主控节点维护（对于P2P架构的系统，这部分信息由所有节点共同维护）。对于主键上的查询，分布式数据库能够在短时间内选择合适的节点进行访问，并发地向这些节点发送对应主键（范围）的查询请求，最后将读取到的结果整合后返回客户端。而对于非主键的数据访问，系统无法确认数据记录所在的位置，只能对每个数据子集s_i进行全部扫描，虽然扫描查找可以并发执行，但在海量的数据规模下，全表扫描造成的时延是无法忍受的。通过非主键属性创建索引来避免全表扫描，是提升非主键数据访问性能的有效手段。

定义3.1.3. *索引记录：记集合I是表T数据S上的索引，通常情况下$|I| = |S|$（特殊地，对于自适应索引[44]二者不一定相同），每条索引的形式如下:*

$$(r_1, P_1), (r_2, P_2), ..., (r_n, P_n).$$

其中r_i是数据集中非主键的信息，且有$r_i \in R_i$,不同的索引结构P_i具有各异的内容，但多数都包含主键信息K_i。

因为索引和数据存在着对应关系，所以可以使用查询索引代替全表扫描，得到主键信息及数据所在的位置，从而提升数据查询的性能。传统的集中式数据库技术经过不断的发展，已经拥有很成熟的索引技术，但对于分布式数据库系统，创建和使用分布式索引面临着新的困难和挑战。本小节将针对读写分离、批量更新架构的分布式数据库，给出分布式环境对索引方法带来的问题的定义。

3.2　索引分布

由定义3.1.3，除了自适应索引（不是本文研究工作的内容）等特殊情况，有$|I| = |S|$。考虑到大数据应用下，$|S|$的值很大，索引拥有与数据相似的规模。存储和管理海量的索引信息，一种方法是优化索引结构减小空间开销，一种方法是对索引也采用分布式管理。第一种方法的效果十分有限，分布式系统通常都考虑将索引分片存储。

定义3.2.1. *索引分片：假设有m个服务器节点，记为$C_1, C_2, ..., C_m$。索引的分片定义为，将索引集合S，依据一个映射函数：*

$$F' : I \to \{I_1, I_2 ... I_k\}$$

划分成k个子集，使得$I_1 \bigcup I_2 ... \bigcup I_k = I$且索引子集$I_i$被放在节点$C_j$上,其中i,j,k满足$1 \leq i \leq k, 1 \leq j \leq m, k = 1, 2, 3...$

划分方法F'决定了索引在集群中的分布状况，也影响到整个索引的性能。对于分布式环境，若一个处理器C_j维护的索引子集过多，或者C_j 上的一个索引子集$|I_i|$过大，会造成集群在查询或者更新的时候C_j的访问量明显高于其它节点，增加了C_j的负载，不利于整个系统的负载均衡。若方法F'将I划分得过细，$C_1 ... C_m$都只维护很小的索引分片，这样在查询的时候，就需要从多个处理器请求索引数据，增加了网络开销成本。避免数据倾斜过大，是决定划分方法的重要考虑因素。

要对I进行索引划分，信息r的特征是需要考虑的因素，包括r 的数据范围，各个数据段分布等。分布式数据库系统中，因为已经对数据S进行了一次分片，当$|S|$很大时，各个节点上均有r的数据，而系统在创建索引之前却并未维护r的整体分布信息，这给索引划分带来了困难。

3.3　索引构建

定义3.3.1. *增量数据与基准数据：记集合S是表T的数据集，在LSM-tree架构系统*

中，数据集S被分为两个部分维护，分别是内存中的增量数据和外存中的基准数据。定义mem_s为S在内存中的增量数据集合，$sst_{s1}...sst_{si}$是写在外部存储的基准数据（基线数据），$i = 1, 2, 3...$

对于数据集S,在某个时间t_1开始创建在r上的索引，根据数据集的初始大小$|S|$，索引的创建工作会在一段时间后的t_2 完成，那么在t_1至t_2的时间段内，如果对表T的内存增量mem_s发生了修改，因为建索引需要对表T进行扫描，若修改发生在已扫描过的部分，这些更新就不会维护在索引里，索引和数据会因此而不一致。

以PostgreSQL[45]为例，传统的集中式数据库对此的解法有两种，一种是为了保证创建索引时与表的数据同步，执行创建索引的语句时系统将整张表T锁定。也就是说，创建索引期间写事务都会被阻塞，直到索引创建成功。另一种解法是不阻塞写事务，对表T进行两次扫描，在第二次扫描时将对数据表的更新补充到索引当中，但在第二次扫描发生时，与索引有关的事务都会阻塞住索引的创建工作。

两种解法对于分布式数据库均不适用。读写分离存储架构的分布式数据库多数没有分布式的表锁机制，分布式的表锁因为其存在多节点的同步问题，所以实现起来比较复杂；而且，对于这种架构的数据库来说，写事务是其性能优势以及主要的应用场景所在，因此锁表阻塞写事务不可行。而且在分布式环境下，表T可以达到很大规模，进行两次扫描的成本过高，且事务阻塞分布式索引的创建工作需要各个节点处理器协调，难度较大且容易出错。现在的分布式索引解决方案，一般都是在建表T的同时将设计的索引一起创建，如果需要在之后的某段时间动态创建索引，则在应用层上停止所有的写事务。

本文的研究工作是，探索并实现将索引集成于读写分离、批量更新的存储架构，可以在任意阶段高效地动态创建索引，并且不阻塞原来的任何事务。

3.4　索引维护和访问

索引维护和索引访问是实现索引功能时需要重点考虑的两部分内容。维护索引的目的是要保证索引中的信息与数据保持一致，防止查询索引得到错误的结果。在分布式数据库系统中，数据S和索引I都是分片存储在集群的不同节点上，在数据S上执行了写事务，其修改也要同步到I上。为了确保跨节点情况下能够正确且高效地实现数据和索引的同步，一方面要优化索引结构，减少更新索引的代价，另一方面，需要最小化不同节点的网络通信代价，后者的作用往往较前者显著。

类似的，节点的通信对分布式环境下索引的查询性能有较大影响，这点在高并发访问数据时尤为突出。查询的网络通信代价主要由两个部分决定，一是不同节点传输数据的次数，二是每次通信传输的数据量大小。在优化分布式索引的访问性能时，经常需要尽量减小这两个方面的开销。

3.5　负载均衡及可扩展性

自动的负载均衡机制，能够动态地最大化系统资源利用率，是可扩展的分布式数据库重要的性质之一。在读写分离架构系统中，副本冗余提供了良好的负载均衡条件，系统能够根据全局负载信息，如节点的CPU，磁盘，内存和网络利用率，对访问副本的选择以及副本迁移等进行整体调度。本文中提及的负载均衡指的都是处理索引维护和查询时的负载均衡，在原系统对数据的自动负载均衡机制正常的前提下，由分布式索引而引起负载不均的原因有如下可能：

1. 索引不均衡。各个节点上索引的分布不均衡导致了对索引的查询请求无法被均衡地处理，小节3.2中论述了其中的一种情况。另外，新上线的节点也会引起索引不均的问题:假设系统中m个服务器$C_1, C_2, ..., C_m$无法承载当前的数据规模，需要对数据库进行成倍扩展，即加入m个新节点$C_{m+1}, C_{m+2}, ..., C_{2m}$，如果分布式索引没有负载均衡策略，新节点没有维护索引，对索引的查询请求依然只在$C_1, C_2, ..., C_m$处理，也会导致整个集群负载不均衡，降低了系统

吞吐量。

2. 流量不均衡。每个节点的索引分布是均衡的，但仍然会由于索引中的某个查询属性存在热点等原因而导致访问请求的流量不均。

分布式索引的设计应当能够解决这些原因带来的负载不均，从而保证不影响整个分布式数据库系统的正常状态。

另外，分布式系统中加入新的计算节点，其集群的处理能力会得到相应提升，这就是可扩展性。分布式索引也应当具有可扩展性，索引的查询处理性能可以随着集群规模的扩大而有所提高。

3.6 本章小结

本章对读写分离架构下创建分布式索引的问题进行了系统的描述，并分析了创建分布式索引时面对的难点，包括以下几点：读写分离架构下难以获取数据的分布情况，因此划分索引、防止索引分布倾斜是困难的；基准数据和增量数据的变化均会影响索引的正确性；需要制定面向该架构的索引维护和查询处理算法；分布式索引还需要满足原系统的负载均衡及高可扩展特性。

第四章　读写分离架构下海量数据的索引构建

本文提出了通用的LSM-Index索引方法，集成于读写分离、批量更新的存储架构，能够实现海量数据的索引构建，并在提升非主键查询性能的同时，降低索引维护开销，使索引满足分布式系统负载均衡、高可扩展特性。本章将论述LSM-Index的设计原理和特点。首先介绍本索引的组织架构。接下来，本章系统地介绍了解决海量数据的索引创建问题的三点关键方法：索引与底层存储的集成，索引的延迟生效策略，以及基于采样的两阶段排序。

4.1　概述

LSM-Index是面向读写分离、批量更新存储机制的分布式索引方法，采取与数据相同的存储架构，对索引属性和主键信息进行全局排序，并将索引划分到多个节点。本小节将对这样的索引组织方式进行系统的讨论。

4.1.1　索引的存储

LSM-Index由两个组件构成，第一部分是写在非易失性存储（磁盘）上的索引文件，称为静态索引$sst-index$，其特点是该组件只读，不接受写事务的更新；第二部分是维护在内存的数据结构，称为增量索引$mem-index$，记录一段时间内对索引的修改增量。如图4.1 所示，LMS-Index完全集成于批量更新的存储机制，其$mem-index$也会将增量记录写入到磁盘的$sst-index$以减少内存上的存储开销。

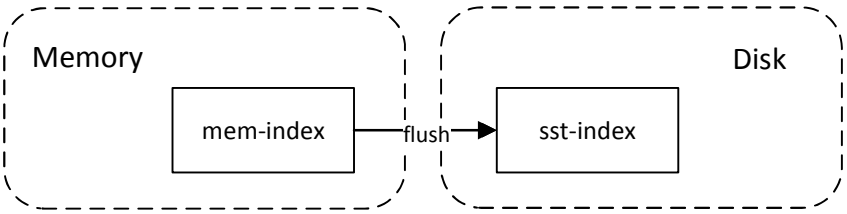

图 4.1: LSM-Index的组成

也就是说，从存储架构的层面上来看，索引和数据有着相同的结构。这类索引的组织方式带来的优势主要有：

1. 只需要在内存中维护索引增量，减少了更新索引的I/O，索引的持久化存储与数据的批量更新一同进行。当然，批量更新是读写分离架构的数据库的核心思想，添加索引需要对批量更新算法进行修改，关于这部分的内容本文将在小节4.3中描述。

2. 读写分离特性的继承，使得在对基准数据$sst_{s1}...sst_{si}, (i = 1, 2, 3...)$构建$sst - index$期间（定义3.3.1），能够将索引的更新维护在$mem - index$，维护了索引和数据的一致，有关不锁表动态创建索引的内容本文将在4.2 小节中论述。

3. 索引和数据的存储管理保持一致，从而分布式数据库能依照原来对数据的管理策略，实现对索引的负载均衡，保证索引的高可靠性。本文将在小节5.3中介绍相关内容。

采用与数据相同的存储机制管理索引，索引在维护的时候需要依照键值来排序，才能实现精确查询和范围查询。每条索引的记录项在LSM-Index 中应该是唯一的，因此本文对索引信息按照查询属性列与原表主键组合进行全局排序，以定义3.1.1和定义3.1.3为基础，下面给出LSM-Index的索引信息描述：

定义4.1.1. *LSM-Index索引记录形式：假设有一张表格T，包含一个数据集合S，每*

条记录的形式如下：

$$(k_1..k_i, r_1...r_j), i = 1, 2, 3.., j = 1, 2, 3..$$

其中列$k_1..k_i$组合成为主键K,列$r_1..r_j$组合成为非主键集合R。对其中一列r_m构建*LSM-Index*，$1 \leq m \leq j$，则*LSM-Index*的索引记录形式为：

$$(r_m, k_1..k_i), i = 1, 2, 3..., 1 \leq m \leq j$$

LSM-Index按照这个形式组织数据并排序，这样在进行精确查询和范围查询时，就可以由查询列r_m得到包含该列的记录项主键$k_1..k_i$，避免全表扫描。

4.1.2　索引的组织

在章节3.2中已经得出结论，分布式环境下，每个处理器需要维护部分索引的分片，因为索引需要排序以满足范围查询，分布式索引技术中，处理器在维护排序的索引分片上主要有两种方案，即局部索引和全局索引。

由定义3.1.2，数据集合S被划分成为$S_1, S_2...S_k$，且分布在节点$C_1, C_2, ..., C_m$中，$k = 1, 2, 3...$，$m = 1, 2, 3...$对每个节点上的数据分片，局部索引的方案建立了一个对应的局部排序，例如，对于S_1 分片，I_1 是由S_1 里的数据按照索引列排序得到的索引，I_2则是对S_2上的索引组织……依次类推，如图4.2，各个局部索引之间相互独立。

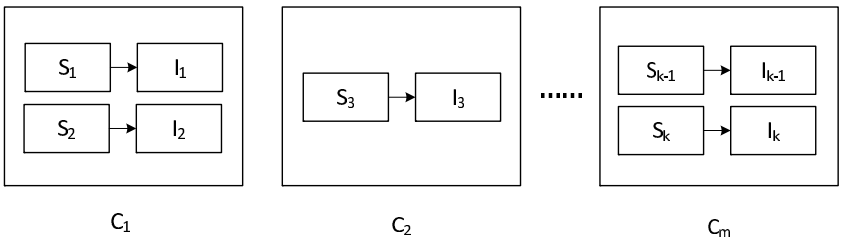

图 4.2: 局部索引方案

局部索引的优势在于，索引和数据总是维护在相同的节点，系统不需要知道索引的全局分布信息，使索引的维护得到了一定的简化，如图4.3.a，在

处理写事务时，只需要定位到数据所在的节点，将数据的更新请求发送到该节点，就可以将数据分片和对应的局部索引一同修改。数据在集群的分布信息维护在coordinator节点上。一般来说coordinator节点为集群的主控节点，如HBase的Master 节点，Oceanbase的rootserver。

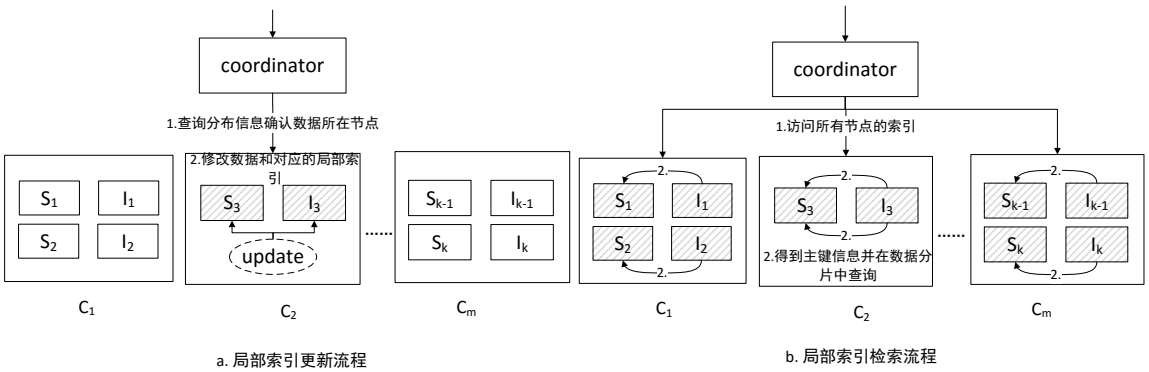

图 4.3: 局部索引更新/查询流程

然而，因为缺少索引的分布信息，局部索引在查询的时候需要访问集群中的所有节点，并将这些节点返回的结果汇总提交给客户端，如图4.3.b，有些节点很可能返回空的结果集，特别是当一个索引的区分度很低（即索引列值相同的数据行很少）时，访问了很多不必要的节点，在高并发的查询时会造成计算资源的浪费和节点负载的增加。

与之相对的，在全局索引方案中，所有的索引信息I在全局排序后，再划分为$I_1...I_k$存储在各个节点当中，如图4.4，一个节点管理的并不仅是本地数据分片的索引。同时，系统维护了索引在各节点的分布信息，如图4.5中的coordinator节点。

如图4.5所示，在响应查询请求的时候，该方案能直接定位出正确的索引分片所在的节点，查询索引并得到索引值所在数据项的主键，使用主键访问数据分片得到结果，避免访问无用的计算节点，提高了查询性能和资源利用率。此时，索引的维护代价会有所增加，因为数据分片对应的索引可能被划分到其它节点上，需要额外的网络通信用以修改索引。

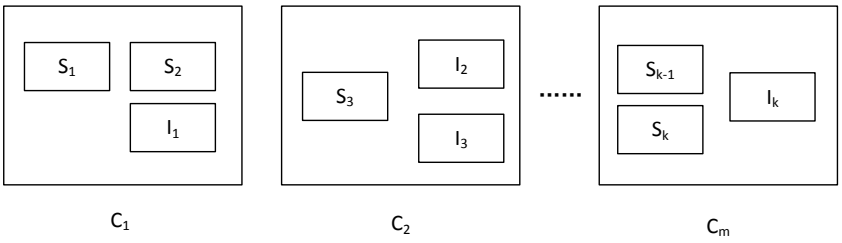

图 4.4: 全局索引方案

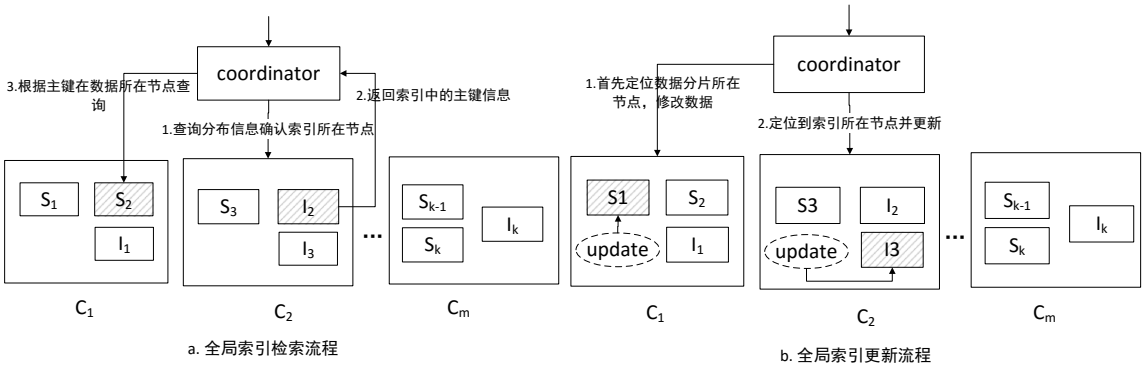

图 4.5: 全局索引查询/更新流程

LSM-Index将索引列和主键属性组合进行全局排序，并划分到不同节点，属于全局索引。索引采用全局排序是为了与数据的存储保持一致，让分布式数据库对数据的管理策略同样适用于索引，比如负载均衡、故障恢复等。

4.2　基于延迟生效策略的索引创建

在章节3.3中已经论述过，为了防止在构建索引期间发生更新而导致索引与数据不一致的问题，分布式数据库通常只支持在建表的时候同时创建索引，无法使用集中式数据库表锁等方法实现任意时刻动态创建索引，现有的研究工作也并没有很好的解决方案。本文的研究基于批量更新原理，提出了索引的延迟生效策略，使LSM-Index能够被动态创建，且不阻塞读写事务。本小节将介绍这种延迟生效策略。

4.2.1 索引周期

依据索引是否可以用于查询服务，本文研究将索引从创建开始的生命周期分为四个阶段，分别是$init$，$auditing$，$available$以及$error$。在这四个阶段中，索引的完整度、可允许的操作不尽相同，延迟生效策略在这四个阶段中逐步完成。

1. $init$，初始状态。表示索引的信息仍未构建完成，处于这个状态的索引允许更新，但不允许查询。

2. $auditing$，审核状态。此时索引的信息已经构建完成，但需要对索引与数据的一致性，以及索引的备份数进行检查，索引仍处于不允许查询的状态，但依然可以更新索引。

3. $available$，可用状态。此时索引可以正常被维护以及提供查询服务。

4. $error$，出错状态。发生错误，索引在这个阶段不允许读写。

图4.6说明了四个阶段的互相转化关系，一般地，索引状态都会随着索引信息的构建完善从$init$ 转变为$auditing$，在经过一致性以及备份数检查后成为$available$。如果索引构建失败，或者在某个阶段检查出与数据不一致时，会将索引置为出错状态。

4.2.2 延迟生效

批量更新是读写分离架构数据库的重要特征，内存中的数据增量会批量写入到永久存储，与基准数据合并成为新的数据文件。LSM-Index 将索引设置为可服务状态推迟至批量更新之后，用新的批量更新算法，在数据合并期间构建基准数据的索引，并将这段时间的索引更新维护在内存中，从而实现了不阻塞事务构建索引，并维护数据和索引之间的一致性。

定义4.2.1. 不同版本下索引的两部分组件：I是在数据集S上建立的索引。使用一个全局的版本号V，表示每次数据库中批量更新完成后数据和索引的版本状态。

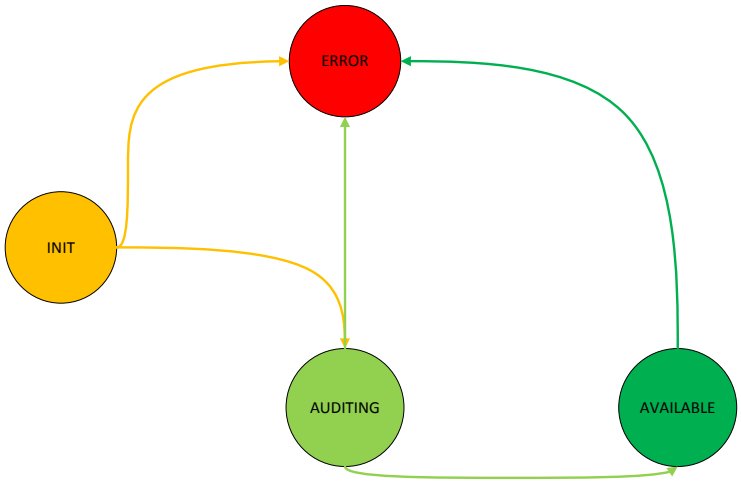

图 4.6: LSM-Index生命周期

定义$mem(S,V_i)$和$sst(S,V_i)$分别为经过V_i版本的批量更新后的增量数据与基准数据。$mem-index(I,V_i)$和$sst-index(I,V_i)$分别标识该版本下的*LSM-Index*的两个组件，$i=1,2,3...$

设定$T_0,T_1...$是时间戳，基于这些时间戳构建LSM-Index流程如图4.7

1. T_0 :数据的全局版本为V_0，对表T上的数据集S创建一个索引I，在内存中创建索引的增量数据结构$mem-index(I,V_0)$，与数据存储不同的地方在于，磁盘上的基准数据的索引文件仍未开始创建，索引的状态为$init$。

2. T_1 :批量更新过程启动，数据集S在内存中的$mem(S,V_0)$中的增量更新开始写入磁盘，与$sst(S,V_0)$合并成新的基准数据文件，而索引的增量数据$mem-index(I,V_0)$不需要做批量更新。

3. T_2 :所有的数据增量更新完毕，数据的全局版本增加至V_1，数据集S的增量数据更新为$mem(S,V_1)$，基准数据更新为$sst(S,V_1)$，索引的增量记录更新为$mem-index(I,V_1)$，相当于清空或者重构了一下数据结构。从此时起，开始构建S基准数据上的索引$sst-index(I,V_1)$，组织静态索引的过程不需要读取增量索引上的记录。若生成$sst-index(I,V_1)$期间有写事务修改

了S和对应的I，将索引的增量更新写入$mem-index(I,V_1)$，索引增量更新将在章节5.2作出论述。

4. T_3 :静态索引$sst-index(I,V_1)$构建工作完成，索引由$init$转化为$auditing$，开始检查$sst-index(I,V_1)$ 和$sst(S,V_1)$的一致性，同时创建索引表的备份。

5. T_4 :索引与数据的一致性检查无误，复制备份完成后，将索引由$auditing$状态转变为$available$，将索引用于查询服务，此时索引包含版本为V_1的$mem-index(I,V_1)$和$sst-index(I,V_1)$。

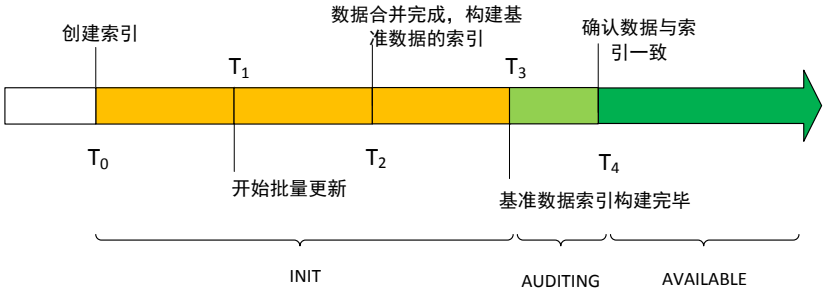

图 4.7: LSM-Index延迟生效的索引创建策略

对基准数据$sst(S,V_1)$创建对应的静态索引，本质上来说是一个如何将基准数据按照索引列有序组织成分布式索引文件的问题，在下一节中将讨论其具体实现。

综上所述，LSM-Index将索引的生效推迟到批量更新完成之后，在创建静态索引期间，对索引的修改都写入内存中的索引增量，最后生成的静态索引和增量索引组成了可用于服务的索引结构，保证了不阻塞事务时索引的正确性。

4.3　静态索引构建

小节4.2.2介绍了本文研究所使用的基于延迟生效的索引创建方法。该方法在最新版本的基准数据上构建静态索引，并将索引的生效延迟到批量更新之后，因此，需要对批量更新过程做一些改动，这些更改只是为了实现索引的创建，不会对数据的批量更新产生影响。本节将描述构建静态索引的方法。

4.3.1　基本思想

根据定义4.1.1，构建静态索引问题的本质，是将表T存储在各计算节点的基准数据按照索引列与主键的组合$(r_m, k_1, k_2...k_i)$ 全局排序组织成分布式索引文件，LSM-Index采用map-reduce[46]思想调度执行一次分布式全局排序任务，分为三个调度子任务：

1. 局部排序。每个计算节点对各自维护的数据分片局部排序。

2. 全局排序。多个计算节点进行数据交互(shuffle)，获取需要的数据排序。

3. 索引备份复制。复制生成更多副本，选择节点分布。

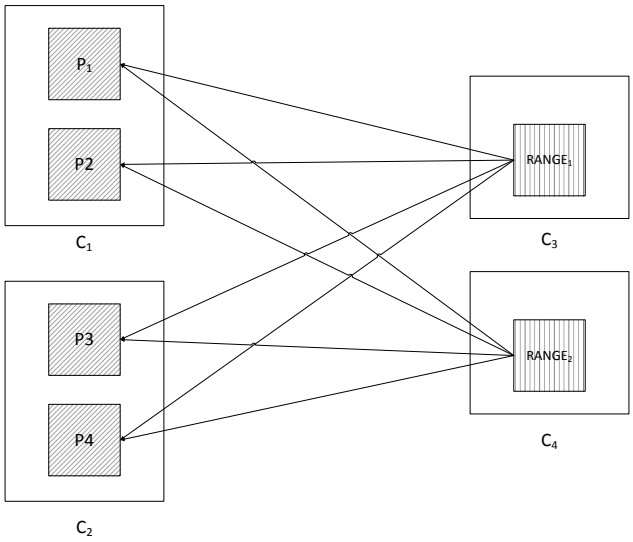

图 4.8: 全局排序的shuffle交互

如图4.8所示，假设计算节点C_3和C_4分别需要构建主键范围为$RANGE_1$和$RANGE_2$的索引分片，而C_1和C_2对数据集S的分片进行了局部排序得到了索引文件（称为局部索引）$P_1 \sim P_4$，那么C_3和C_4将与C_1和C_2分别通信，获取满足$RANGE_1$和$RANGE_2$的索引记录，并在各自节点再进行一次全局排序，得到最终的静态索引文件。

4.3.2 算法介绍

依据定义4.1.1，假设在V_0版本对表T上的r_m列构建索引I，需要对其创建静态索引文件$sst-index_1(I,V_1), sst-index_2(I,V_1)...sst-index_{k'}(I,V_1)$，$k'=1,2,3...$。整个过程的算法伪代码如算法1所示。

Algorithm 1 静态索引构建算法(Part 1)

输入: 表T数据集S的分片集合
输出: 索引I的分片集合

```
1: 启动批量更新过程
2: 对于计算节点{C_1, C_2...C_m}，每个计算节点并发执行如下调度任务
3: for all 增量数据Δ in memory do
4:     if Δ ⊆ DataSet Q then
5:         merge Δ with sst(Q, V_0) and flush Δ → Disk
6:         update sst(Q, V_0) to sst(Q, V_1)
7:     else if Δ ⊆ IndexSet
8:         none
9:     end if
10: end for
11: report to coordinator
12: 节点调度子任务结束
13: update global version V_0 to V_1
```

伪代码2 $\sim$ 11行描述的是每个计算节点执行增量数据合并到磁盘，写成新版本的基准数据的过程，与原批量更新不同的地方是，在内存中只有数据的增量数据是需要写入磁盘的（伪代码4 $\sim$ 6行）。每个节点的增量更新写入结束后会回复反馈消息给coordinator节点（伪代码11行）。最后将全局的版本由V_0 升至V_1。

coordinator节点在确认全局的批量更新完成之后，调度计算节点开始进行局部排序。伪代码14 $\sim$ 23行描述了此任务调度，即迭代本节点中数据集S的基准数据，将每一行数据由$(k_1,k_2..k_i,r_1,r_2..r_j)$的形式映射成由索引列与表$T$主键组合的索引记录$(r_m,k_1,k_2..k_i)$并进行局部排序（伪代码15 $\sim$ 19行），将排序后的结果写入磁盘，称为局部索引$local_index$。节点完成局部排序后，同样汇报反馈信息

静态索引构建算法(Part 2)

14: 对于计算节点$\{C_1, C_2...C_m\}$，coordinator节点挑选表T数据集第一副本所在的节点
15: 向挑选出的节点发送构建局部索引的调度命令
16: **对于挑选出来的计算节点，每个计算节点并发执行如下调度任务**
17: **for all** $sst \in \{sst_1(S, V_1), sst_2(S, V_1)...sst_k(S, V_1)\}$ **do**
18:　　**while** $row \leftarrow get\_next\_row(sst)$ **do**
19:　　$map\ row : (k_1, k_2..k_i, r_1, r_2..r_j)$ to $row' : (r_m, k_1, k_2..k_i)$
20:　　add row into $SortHeap$
21:　　**end while**
22:　　flush $SortHeap \rightarrow Disk$ called $local\_index$
23: **end for**
24: report to coordinator
25: **节点调度子任务结束**
26: $F'(I) \rightarrow \{I_1, I_2...I_k\}$

给coordinator。coordinator 在确认局部排序全部结束后，确定索引I在集群中的分布信息，为每个计算节点分配一些索引记录的范围区间$range$，将索引的$range$集合发送到计算节点，并进行全局排序。索引的划分将在3.2小节系统论述。

全局排序阶段，计算节点有两种类型的调度任务待执行，一种是接收到来自其它计算节点的索引记录请求，根据范围区间$RANGE$ 查询本机中的$local_index$文件找到符合条件的记录，返回结果集$result_set$（伪代码27 $\sim$ 33行）。另一种任务是来自coordinator 的任务调度，计算节点接收到数个索引分片的范围区间，对于每个范围区间，访问其它节点获取该区间内的索引记录，汇总$result_set$后进行全局排序，最后将得到的结果集写入磁盘成为全局的静态索引（伪代码34 $\sim$ 44行）。

4.3.3　并行计算作优化

在数据规模比较大的情况下，某些表的数据分片数目会很大，为了提高静态索引构建效率，需要做一些优化改进。分析算法1，局部排序阶段对数据分片

静态索引构建算法(Part 3)

对于计算节点$\{C_1, C_2...C_m\}$，coordinator节点挑选索引划分后，索引第一副本所在的节点
对于挑选出来的计算节点，每个计算节点并发执行如下调度任务
if receive request for index record with $RANGE$ **then**
    for all $p \in local\_index$ **do**
        while $row \leftarrow search\_local\_index(p, RANGE)$ **do**
            add row' into $result\_set$
        end while
    end for
    response $result\_set$
else if receive schedule for global index with a set of $RANGES$
    for all $range \in RANGES$ **do**
        for all node which is pick out by coordinator **do**
            send request for $result\_set$ of index record with $range$
            while $row \leftarrow get\_next\_row(result\_set)$ **do**
                add row into $SortHeap$
            end while
        end for
        flush $SortHeap \rightarrow Disk$ which is $sst - index_1(I, V_1)$
    end for
end if
节点调度子任务结束
批量更新流程结束

进行映射、排序，各个分片上的作业各自独立，没有冲突和依赖关系；同样的，全局排序阶段，各个节点接收各自的索引范围区间、读取其它节点的索引，以及shuffle作业也互不冲突。因此，可以将局部排序和全局排序使用多线程并行执行。

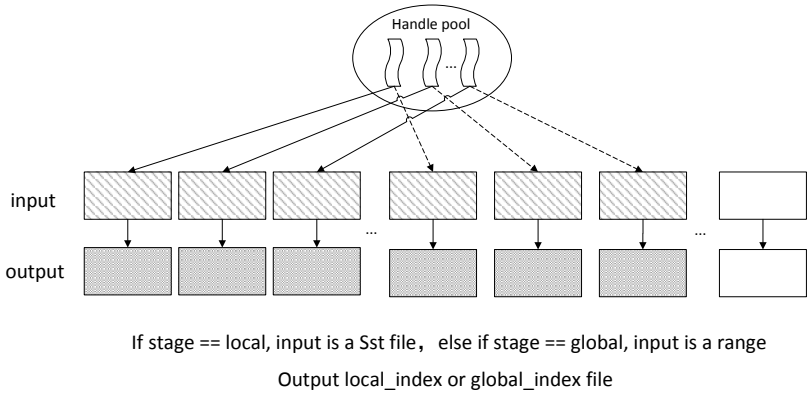

图 4.9: 并行任务调度模型

多线程的作业调度模型如图4.9，每个节点都维护一个线程池，每个线程根据coordinator的消息选择一个基准数据文件或者索引范围区间进行局部或全局的排序。当所有的作业完成后线程阻塞，直到下次任务到来时被唤醒。

4.4　索引划分

由问题定义3.2，分布式系统需要对索引进行分片，将索引的子集分布式存储在计算节点上。本文在算法1介绍过，LSM-Index执行划分索引是在局部排序阶段结束后，计算每个节点上维护的索引范围区间，并调度各节点获取该区间内的索引记录进行全局排序。本节将介绍关于索引划分的工作。

4.4.1　索引分片数量确定

划分索引区间，首先的工作是确定索引分片的数量。对于数据分片，其在节点内的总分片数量与两个参数相关：数据的存储开销，数据分片文件大小上限。对于读写分离、批量更新架构的分布式数据库系统，数据分片过大会影响性能，

因此当内存中的更新数据写入磁盘量到达上限时，数据库将新打开另一个文件供数据写入。每个数据分片大小是大致相同的，接近于最大分片上限。

定义4.4.1. *表T在分布式数据库中的分片数目N为：*

$$N = \lceil \frac{size(T)}{M} \rceil \times replication$$

其中$size(T)$为表T实际所占用的存储空间，M为数据分片的空间使用上限，$replication$表示冗余的副本数。

现在来考虑索引的分片数目，由定义3.1.3，数据表S上的索引I集合空间大小相等，且LSM-Index方法将索引集成于底层存储，其存储形式和数据相同。因此，分片大小上限、副本数等限制对索引也是有效的。根据定义4.1.1，结合以上推论可以得到：

定义4.4.2. *表T上的索引I在分布式数据库中的分片数目N'为：*

$$N' = \lceil \frac{size(I)}{M} \rceil \times replication$$

其中$size(I)$为索引I实际所占用的存储空间，M为数据分片的空间使用上限，$replication$表示冗余的副本数。

从而推导出：

$$N' = N \times \lceil \frac{size(I)}{size(T)} \rceil,$$

因为索引和表的记录项数相等，亦即是:

$$N' = N \times \lceil \frac{size(r_m, k_1, k_2...k_i)}{size(k_1, k_2...k_i, r_1, r_2...r_j)} \rceil,$$

索引与数据的分片数量之比等于索引列数据在整行记录中的存储占用比率。求出这个比率并不困难，对于定长存储，只需比较表与索引的行结构就可以求出结果。而对于变长存储，则需要在实际中计算，得出一个估计值。LSM-Index的

局部排序阶段，在将数据记录映射成索引记录的时候（算法1,伪代码18行），会计算出索引列在整行的存储比值。索引与数据的分片数量之比等于索引列数据在整行记录中的存储占比。求出这个存储占比并不困难，对于定长存储，只需比较表与索引的行结构就可以求出结果。而对于变长存储，则需要在实际中计算。即在LSM-Index的局部排序阶段中，把数据记录映射成索引记录的时候（算法1,伪代码18行），将计算出索引列在整行的存储比值。

4.4.2　基于采样的索引区间划分

划分索引的目标是保证索引表的均匀分布，各个索引分片的大小应大致相等。最简单的划分方法是将全部索引记录在一个节点上完成全局排序，再将排序结果均匀划分给其它节点，但实际并不可行，计算节点无法独自完成大数据规模的排序工作。可以看出，分布式索引分片的难度在于无法获取索引列记录的整体分布信息，LSM-Index使用基于局部索引采样的方法计算索引记录的范围区间。

采样的主要思想是查看集合中的一小部分数据以获得记录的近似分布，LSM-Index使用两次采样来最终划分全局索引的范围区间。首先，各个节点对局部索引文件进行第一次采样，并将采样结果汇报给总控节点coordinator；coordinator对汇总的采样结果进行第二次采样，使用第二次采样的结果划分数个索引的范围区间。统计学中有三种常用的采样方法，见图4.10：

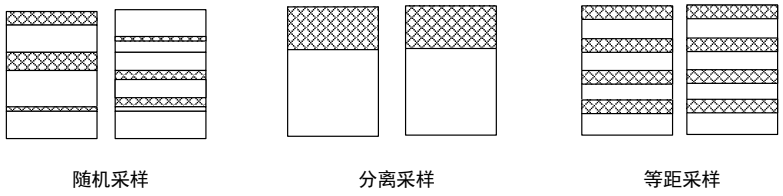

图 4.10: 三种采样方案

- 随机采样（random sampling）。该方法以一个采样率参数$freq$作为输入，所有的记录有$freq$大小的概率被采样，该方法有两个限制条件：样本数足够或者扫描分块数达到了最大限制，满足了任意一个采样过程即停止。

- 分离采样（split sampling）。每个分块仅采样前n条记录，直至样本数足够或者访问分块数目达到上限。因为很可能未对所有分块广泛抽样，这类方法不适用于排好序的样本。

- 等距采样(interval sampling)。对于每一个分块，每间隔一定的距离d，定期执行采样，适合用于已经排好序的数据分块，可以提高样本的均匀程度。

Algorithm 2 基于采样的索引分片算法

输入: 索引的分片数量N',数据的分片数量N,局部索引集合$\{local_index_1, local_index_2...\}$

输出: 索引I的分区与对应的分配节点

$P \leftarrow N' - 1$
$num \leftarrow P^2/N$
$i = 0$
coordinator节点对所有计算节点执行采样调度
对于计算节点$\{C_1, C_2...C_m\}$，每个计算节点并发执行如下调度任务
for all $idx \in \{local\_index_1, local\_index_2...\}$ **do**
    take num samples of idx in interval strategy
    add samples in set $sample\_result$
end for
report $sample\_result$ to coordinator
节点调度子任务结束
coordinator receive all $sample\_result$, that is P^2 samples.
add P^2 samples into $SortHeap$;
for all $c \in SortHeap$ **do**
    if c is $P * i_{th}$ sample **then**
        pick out c and $i++$
    end if
end for
use sample sequence $c_1, c_2...c_P$ to assemble N' index ranges $(MIN, index\_key_{c1}], (index\_key_{c1}, index\_key_{c2}]...(index\_key_{cP}, MAX)$
allocate every range to a node

本文的静态索引创建阶段，其局部排序和全局排序使用了map-reduce思想，

且索引分区在局部排序完成后开始执行，即每一个需要采样的数据都是局部有序的，所以LSM-Index使用等距采样来实现索引分区。

为了使各个索引分片的记录范围连续，本文用一个连续的区间序列来表示索引分片的关键字取值范围，如下所示：

$$(MIN, index\_key_1], (index\_key_1, index\_key_2]...(index\_key_i, MAX)$$

$index_key$代表索引记录的关键字，即定义4.1.1中的$\{r_m, k_1, k_2...k_i\}$索引记录与原表主键的组合。索引的分片由一系列左开右闭（最后一组除外）的连续区间表示，MIN和MAX表示极小值和极大值。依据定义4.4.1和定义4.4.2，在计算出索引分片数量N'之后，将索引I划分成数个子集，并分配给各个节点执行分区作业的流程如算法2所述：

在算法伪代码中，对每个局部索引进行等距采样，采样的样本数为$\frac{(N'-1)^2}{N}$，这样在计算节点返回所有的样本之后，coordinator节点就得到了$(N'-1)^2$个样本，对这些样本再进行一次排序（伪代码13行），并进行第二次等距采样后，最终得到$N'-1$个抽样点，使用这些样本两两组合成为连续区间序列（伪代码$14 \sim 19$行）。

从算法2看出，LSM-Index抽样的最终目的是得到$N'-1$个样本，两两组合这些样本的关键字组成连续的范围区间，如图4.11。第一次抽样的样本数$P=(N'-1)^2$，这个值让第二次等距抽样能够得到$N'-1$个样本，实际上，参数P可以设置得更大。显然，P越趋近于集合I的空间大小$|I|$，其样本空间越接近于索引全集，但同时，样本空间也会增大，从而给采样带来难度。

对于将连续的索引区间序列分配给节点，本文采用了十分直观的策略，对于每个区间，遍历节点$C_1, C_2..$，若发现某个节点维护的局部索引集合与全局索引区间的交集不为空，则这个区间就分配在该节点，这是为了尽量为节点进行全局排序时提供一些本地化数据。同时，分配的时候以每个节点维护索引分片数目大致

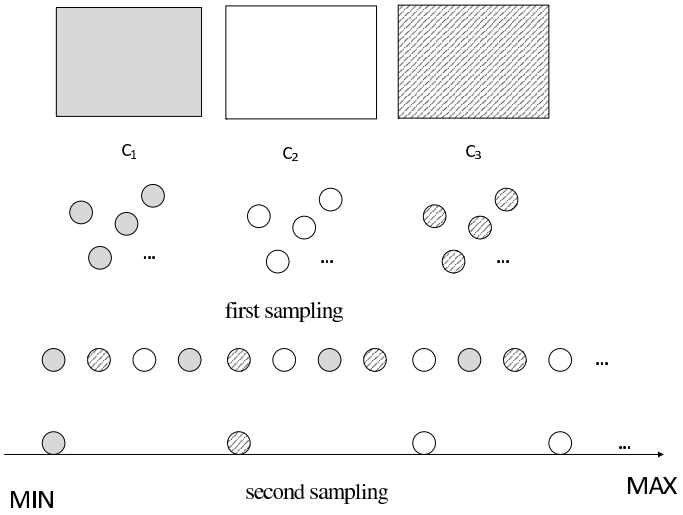

图 4.11: 基于采样划分索引区间

相等为优先原则，优先选择当前索引分片数目较少的节点。当然，此采样算法并不适用于所有的索引列分布状况，如果该采样方法无法满足应用需求，可以根据需要调节参数P大小，或者根据实际采用不同的采样算法。

4.5　本章小结

本章系统地介绍了LSM-Index索引的特点，构建方式以及查询处理过程。为了能够实现高效查询，且满足分布式数据库的负载均衡和高可扩展需要，LSM-Index集成于读写分离、批量更新架构的底层存储之上，按照数据的组织方式组织索引，对索引列和原表主键的组合进行全局排序。因为集成于读写分离架构，LSM-Index 包括两个组件，内存中的增量索引和磁盘上的静态索引。LSM-Index灵活运用了批量更新的核心思想，基于索引延迟生效的策略使得系统能够在不阻塞事务的情况下动态创建索引，同时使用基于采样的两阶段排序和并行任务调度，高效地实现了均匀的索引划分和全量数据的索引构建。

第五章　索引的查询和维护处理

在章节四中，本文给出了在读写分离架构下创建LSM-Index的批量更新算法，该算法能够高效地创建出均匀分布的全局索引。本章重点讨论LSM-Index 的数据访问的开销、更新维护的代价、可行的优化策略，并总结索引的分布式特性。

5.1　查询处理

LSM-Index采用与数据相同的存储结构，一段时间内索引记录的增量修改被维护在内存中。对于读写分离、批量更新架构的分布式数据库而言，内存中增量数据的分布方式有两种：

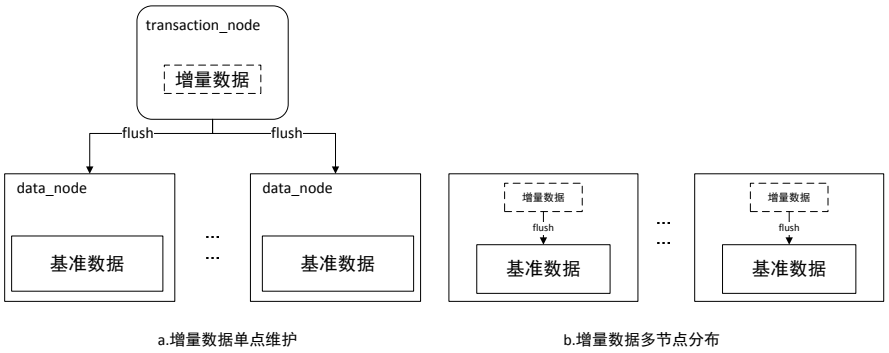

图 5.1: 两种维护增量更新的方式

第一种是所有的增量数据都维护在单节点内存中，见图5.1.a 中事务节点（transaction node）。单点维护增量更新可以避免分布式事务，将事务处理和底层存储（数据节点data node）彻底分离，是一种能够实现跨行跨表长事务的经典解决方案。但是为了保证单点事务性能，对事务节点的内存与计算能力要求较

高，且事务节点和数据节点之间的交互会有网络开销，Deuteronomy数据库以及OceanBase 均采用此方式维护增量数据；

第二种是多节点共同维护增量更新，如HBase。此时系统能够让各节点分担数据写入和查询的负载，见图5.1.b。但为了保证多节点更新的一致性，需要额外的维护代价。

读写分离架构的分布式数据库，为了将基准数据和增量更新合并后返回结果，不可避免地要产生网络和磁盘交互。本节将给出此架构中LSM-Index基于回表的查询处理方法，以及使用冗余列策略的不回表优化方式。

5.1.1 回表查询处理

查询记录的类型有等值查询和范围查询两种：

get(C,Key):以Key为关键字，查找记录项，满足记录中属性C的值与Key相等。

scan(C,Filter):对属性C进行以$Filter$为过滤条件的范围查询。

依据定义4.1.1，索引记录以$(r_m, k_1, k_2..k_i)$为组合键存储在内存组件（通常是树形结构，如B+树）中。为便于描述，使用SK表示创建索引的属性(r_m)，PK代表原数据表的主键($k_1, k_2..k_i$)，$ENTRY$ 为二者的组合，因为PK的唯一性，使用此组合(SK, PK)作为关键字能区分每一条索引记录。

LSM-Index根据(SK, PK)的组合关键字进行全局排序，因此对此关键字的前缀查询是高效的。如算法3所示，处理以非主键属性SK为过滤条件的查询Q时，若该属性列有索引，则首先在LSM-Index上对索引列执行前缀范围查询$scan(SK, Filter)$（伪代码第2行）。如果Q 是等值查询，则Filter 此时没有任何限定。范围查询会得到满足条件的索引记录结果集，对于结果集中的每个记录，使用原数据的主键值pk 再以$get(PK, pk)$ 的方法对数据表T 进行等值查询，得到最终的结果进行汇总并返回查询客户端（伪代码4 $\sim$ 7），第二次等值查询原表的过程称为回表。

Algorithm 3 基于回表的查询方法

输入： 表T中非主键查询属性SK

输出： 包含SK的结果集$result_set$

1: scan $mem - index$ and $sst - index$ with prefix Filter
2: get tmp_result of $scan(SK, Filter)$
3: **for all** row in tmp_result **do**
4: 　pick out pk from $row : (SK, PK)$
5: 　query mem and sst of T with func $get(PK, pk)$
6: 　get row' from response
7: 　add row' in $result\_set$
8: **end for**
9: return $result\_set$

基于回表的查询处理中，无论是前缀查询$scan(SK, Filter)$，还是等值查询$get(PK, pk)$，都会产生网络交互和磁盘I/O。根据维护增量数据的方式的不同，分析不同系统中一次非主键查询在最坏的情况下的代价，假设系统中有N个节点，则最差的情况是两阶段的查询都访问了所有节点。用$Client_NLatency(C_i, phase)$表示发生$phase$阶段的因为客户端与节点$C_i$交互产生的网络时延，$phase$ 表示处于查询的第一次scan 请求，或者第二次回表get请求。$SSt_NLatency(C_i, phase)$表示单点维护增量数据的系统处理某阶段请求时，事务节点与数据节点交互产生的网络时延。使用符号$DLatency(C_i, phase)$表示发生在C_i节点上因为读取基准数据造成的磁盘时延。

单点维护增量数据的系统的查询网络时延为：

$$\begin{aligned} L_{net} = &\max\left(Client\_NLatency(C_i, scan)\right) + \max\left(Sst\_NLatency(C_i, scan)\right) \\ &+ \max\left(Client\_NLatency(C_i, get)\right) + \max\left(Sst\_NLatency(C_i, get)\right) \end{aligned} \tag{5.1}$$

多点维护增量数据的系统查询网络时延为：

$$L'_{net} = \max\left(Client\_NLatency(C_i, scan)\right) + \max\left(Client\_NLatency(C_i, get)\right) \tag{5.2}$$

二者的磁盘时延相同，都为：

$$L_{disk} = \max\left(DLatency(C_i, scan)\right) + \max\left(DLatency(C_i, get)\right) \tag{5.3}$$

上述式子中，都有$1 \leq i \leq N$。

由等式可知，单点维护增量数据的系统，在查询LSM-Index上的网络时延要高于多点维护增量数据的系统，其原因在于单点写入的架构中，基准数据和增量数据的合并需要网络传输。

5.1.2 不回表查询处理

由等式5.1，5.2，5.3对网络和磁盘的代价计算可以得出，第二次回表的等值查询的磁盘和网络代价都占了很大比重，为了避免回表造成的巨大开销，LSM-Index使用冗余列（Storing Column）的方式实现了不回表的优化。

冗余列优化方式将更多的记录信息存入索引，使用章节5.2.1的符号定义，$ENTRY$表示属性列与数据表主键的组合，另外，索引记录中冗余存储一部分数据表中的其他列属性，用$STORING_CLOUMN$表示，这时索引记录的形式可以写为$(ENTRY, STORING_COLUMN)$。如算法4所示，Q查询记录中的RES属性，且记录中的索引列SK满足范围或等值查找条件，则首先判断RES 是否满足$RES \in STORING_COLUMN$，如果满足，则在对索引表查询得到记录结果之后，就能够直接从索引记录集里直接返回RES。如果不满足，那么继续进行第二次回表查询。

因为索引冗余存储了一些列，索引的存储空间开销增大。对于扩展集群容量极为简便的分布式数据库来说，用空间换取不回表查询的性能提升，是可以接受的。在工程实践中，应当根据不同的应用场景，选取查询热点的列值冗余存储。

5.1.3 基于规则的索引选择

由于不同的应用需求，系统可能会在一张表上建立多个索引，而一个查询的查询条件也可能涉及到多个索引，此时系统需要从多个可用的索引中选取一个以提供查询服务。目前有两种方案实现基于索引的优化，基于规则和基于代价的优化方法[47]。前者是在解析查询请求时，遵循系统内部预定的索引使用规则选择索引；后者主要是由优化器来预计查询语句的执行代价，根据计算的结果不同，

选择查询代价最小的索引构建查询计划。LSM-Index实现了基于规则的索引选择方案，其规则概要如下：

Algorithm 4 基于不回表的查询方法

输入： 表T中非主键查询属性SK

输出： 包含SK的记录中的RES结果集$result_set$

```
1: bool query_back = RES ∈ STORING_COLUMN ? true : false
2: scan mem − index and sst − index with prefix Filter
3: get tmp_result of scan(SK, Filter)
4: if query_back then
5:     for all row in tmp_result do
6:         pick out res from row : (ENTRY, STORING_COLUMN)
7:         add res in result_set
8:     end for
9: else
10:    for all row in tmp_result do
11:        pick out pk from row : (SK, PK)
12:        query mem and Sst of T with func get(PK, pk)
13:        get row′ from response
14:        add res from row′ in result_set
15:    end for
16: end if
17: return result_set
```

一个查询SQL语句的形式如下，其中Cond()表示某列值需要满足一个过滤条件：

SELECT $C1$，$C2$ FROM T WHERE Cond($C3$) and Cond($C4$);

系统在处理此查询SQL时，选择符合下列规则之一的索引执行查询：

1. 查询条件中包含全部主键属性或主键前缀时，不用索引

2. 若查询条件只包含了一个索引列，如$C3$，则选择在$C3$上建立的索引。

3. 若查询条件可以使用两个索引，分别在$C3, C4$上创建且都可用于服务，优先选择拥有冗余列$C1$或者$C2$的索引

4. 若查询条件可以使用两个索引，且都包含（或者都不包含）冗余列$C1, C2$，则依照查询条件最左匹配原则选择索引，例如，此时应该选择在$C3$ 上建立的索引。

基于规则的索引选择有一定缺陷，缺少足够的灵活性，有时会选择错误的索引影响效率。这和索引列的区分度有关，某索引列的区分度越高，属性值相同的记录越少，相对地，在该索引列上的查询效率也就越高。若索引列$C3$的区分度远小于$C4$，则选择$C3$上的索引并不是最佳方案。

为了使索引的选择更具有合理性，未来的研究工作中，LSM-Index将进一步实现分布式环境下使用统计信息进行代价估算的索引优化。

5.2 索引的更新

5.2.1 算法介绍

分布式数据库通常提供两种类型的数据访问方式，一是标准的SQL查询语言，第二是访问数据库的接口函数，后者在NoSQL类数据库较为常见。本文定义三种数据的更新操作：

- $op - insert$:向数据库写入一条记录，如SQL中的insert语句、调用键值系统的put()接口。

- $op - update$:对数据库中已有记录某个属性的值进行修改，如SQL中的update语句，键值系统中用put()写入相同的主键记录造成的更新等。

- $op - delete$:将数据库中的一条记录删除。

为了维护索引与数据之间的一致性，当一个更新发生时，先修改数据，再更改对应的索引。图5.2 表示数据和索引之间不同的修改顺序可能造成的影响。虽然

并不是所有的分布式数据库都支持ACID的事务，但都能通过行锁保证数据行级别上的原子性。图5.2中写事务$operation-1$和$operation-2$都试图修改$data_row$，得到的索引记录分别是$index_row_1$和$index_row_2$，图5.2.a表示执行顺序不同可能造成的错误，写事务$operation-2$可能先于$operation-1$完成所有更新并释放行锁，造成索引与数据不一致。在图5.2.b 中，由于$data_row$ 首先被上锁，其余的任何操作都必须等到$operation-2$执行完成后才能开始。因此，需要先修改索引以维护其和数据是一致性。

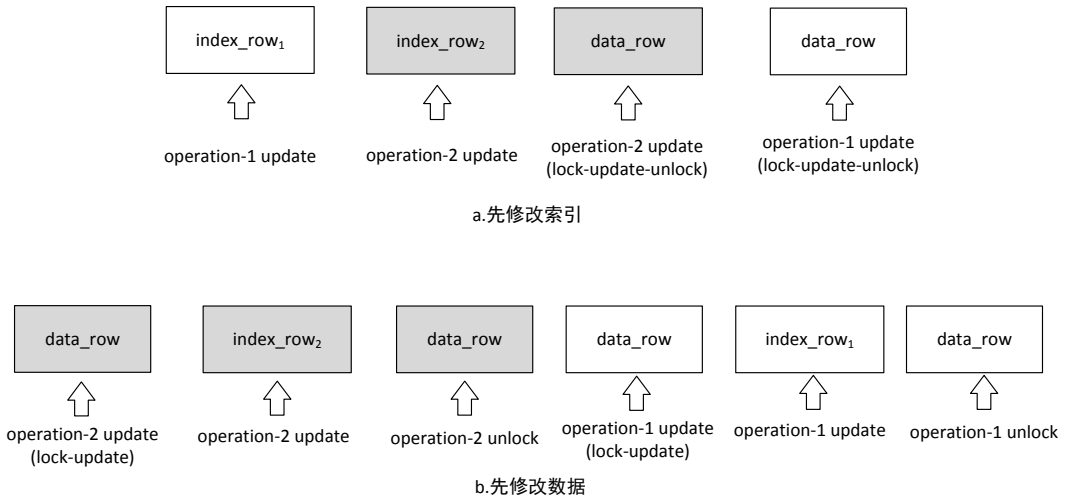

图 5.2: 执行顺序对索引一致性的影响

op-insert（算法5）：写入新的记录row之前，对PK所在的记录行上锁（伪代码第1行），之后对内存中的增量数据组件作修改，插入一个新的节点。接着将row变换成(SK, PK)形式的记录，并在$mem-index$中加入索引节点$ENTRY$（伪代码$2 \sim 4$ 行）。如果期间某个环节失败，则抛回错误（伪代码$8, 11$行）。将上述更新写入日志后，释放行锁，返回客户端写事务成功（伪代码$5 \sim 6$ 行）。

op-delete(算法6)：读写分离、批量更新架构的分布式数据库中，内存中的增量更新以追加日志的方式维护，即对于任何操作，都在内存组件当中新写一条记录。删除一行数据项时，通常是给这行记录追加一个删除标记（tombstone），直到批量更新之后，才真正将这行数据项删除（伪代码3行）。对于索引记录的删

Algorithm 5 op-insert更新索引算法

输入： 数据行$row : (PK, SK...)$

lock node of PK entry
if (SUCCESS == insert_node(row)) **then**
    map row into (SK, PK),SK and PK make up a entry of new node
    if (SUCCESS == insert_node($ENTRY$)) **then**
        write all mutation into $commit\_log$
        unlock node of entry PK and report success to client
    else
        abort() and unlock node of entry PK
    end if
else
    abort() and unlock node of entry PK
end if

Algorithm 6 op-delete更新索引算法

输入： 数据行主键PK

lock node of PK entry
read mem and Sst get $row : (PK, SK, ...)$
if (SUCCESS == add_tombstone_to_node(PK)) **then**
    map row into (SK, PK),SK and PK make up a entry of new node
    if (SUCCESS == add_tombstone_to_node($ENTRY$)) **then**
        write all mutation into $commit\_log$
        unlock node of entry PK and report success to client
    else
        abort() and unlock node of entry PK
    end if
else
    abort() and unlock node of entry PK
end if

除，也使用相同的策略，在给原数据增加tombstone之后，构造新的$ENTRY$，在键为$ENTRY$ 的索引记录上写一个tomstone（伪代码4 $\sim$ 5行）。另外，因为内存中只维护一段时间内的更新，为了防止将要删除的数据行的索引列不在内存中，导致变换$ENTRY$ 错误，需要读取磁盘上的基准数据中的原记录（伪代码第2行）。

op-update(算法7)：如果$op-update$修改的属性为索引列（伪代码第3行），即试图将$row:(PK,SK,...)$通过函数$update_value()$ 更新为$row':(PK,SK',...)$，那么索引增量$mem-index$ 中的记录(SK,PK)就不该存在，取而代之的是记录(SK',PK)。因此，在修改数据行row之后（伪代码第7行），对索引的修改应为首先删除记录(SK,PK)，再插入一行(SK',PK)(伪代码8 $\sim$ 11行)。与$op-delete$类似，需要读取磁盘的基准数据确保能得到原数据的索引列。

算法5，6，7中，写恢复日志的流程发生在修改内存增量成功后。而一部分系统中写日志先于数据结构的更新，这种日志被称为WAL(先写日志，Write Ahead Log)，需要指出的是，执行写恢复日志时机的差异不会对整体的算法产生影响。

5.2.2　分析与优化

上述算法中索引的更新都发生在内存中，在以B+树作为内存组件的系统中，算法平均复杂度为$O(\log n)$。对于单个索引，单点维护增量数据的系统中三种更新的代价开销如表5.1所示，虽然所有写事务由单个节点处理，但一些更新需要得到基准数据中的记录，因此会产生网络交互和磁盘I/O，此外，写事务日志也需要访问磁盘，因此$op-delelte$与$op-update$的操作，其访问磁盘与网络交互的次数均为2次和1次。在该表中$op-insert$的开销括号后面的数字代表其网络交互可能为1，原因是因为对于分布式关系数据库，处理insert 类型的SQL 语句需要拒绝掉写重复主键的请求，因此需要读取基线数据判断主键是否重复，磁盘交互同理。

因为架构上的限制，LSM-Index无法从网络和磁盘交互次数方面优化索引更新性能，因此本文通过减少磁盘I/O来降低维护开销，由于LSM-Index 的存储机制与数据相同，索引的修改日志也一同写入磁盘，记$COST_{disk}(size)$为写日志的开

Algorithm 7 op-update更新索引算法

输入：数据行主键PK，更新属性的修改值col。

read mem and Sst get $row : (PK, SK, ...)$
lock node of PK entry
bool index_flag = $col \in SK$? true : false
if (index_flag) **then**
    map row into (SK, PK),which is $ENTRY$
end if
if (SUCCESS == update_value(PK, SK, SK')) **then**
    if (index_flag) **then**
        map row' into (SK', PK),which is $ENTRY'$
        if (SUCCESS == add_tombstone_to_node($ENTRY$))
and (SUCCESS == insert_node($ENTRY'$)) **then**
            write all mutation into $commit\_log$
            unlock node of entry PK
            report success to client
        else
            abort()
        end if
    else
        write all mutation into $commit\_log$
        unlock node of entry PK
        report success to client
    end if
else
    abort()
end if

销，$size$为更新操作的日志大小，则对于LSM-Index 写日志的代价为

$$COST_{disk}(size(row) + N * size(ENTRY))$$

其中N为索引的个数，随着N和$ENTRY$的增大，其磁盘开销所占比重会加大。写恢复日志的目的是故障恢复，能够让系统通过回放日志重构内存中的数据和索引，但实际上每一行索引记录都能通过数据再重构，因此，可以忽略掉写LSM-Index的日志操作，使得写日志磁盘开销变为

$$COST_{disk}(size(row))$$

执行这样的优化，需要对日志回放的流程进行改动，从而通过回放的数据重构索引，使索引能在节点的故障修复完成、上线后依然可用。

表 5.1: 不同更新类型造成的代价

更新类型	修改mem复杂度	磁盘交互(次)	网络交互（次）
$op-insert$	$2 * O(\log n)$	1(2)	0(1)
$op-delete$	$2 * O(\log n)$	2	1
$op-update$	$3 * O(\log n)$	2	1

5.2.3　增量数据多点分布的情况

对于增量数据多点分布的系统，LSM-Index依然采用上述算法维护索引，但是需要一个观察者（Index ObServer）来协调更改数据和索引的流程，其作用类似于Hbase中的Coprocessors，如图5.3，Index ObServer 在修改数据之后，通过网络通信发送请求更改可能位于其他节点的增量索引，只有两个修改都成功后，Index ObServer才会返回客户端修改成功。随着索引表个数的增加，与其他节点的通信开销也会增大，最差的情况下，N个索引的更新将导致N 次网络通信，这会影响索引的更新性能。

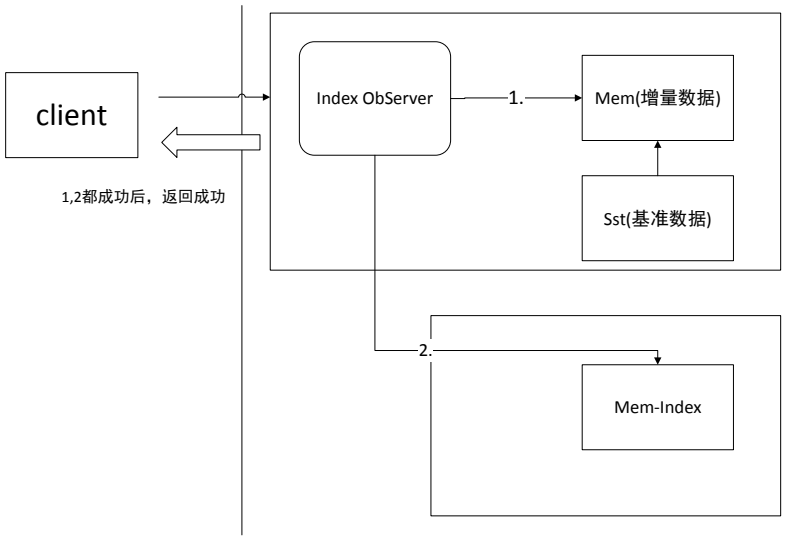

图 5.3: 多节点维护增量数据的系统的更新索引流程

5.3 负载均衡

集群环境中，根据各个处理器的I/O，CPU计算资源以及网络带宽开销的情况寻求负载均衡，已经被证明是一个NP问题[48]。目前并没有最通用优秀的解法。不同实现方式的分布式数据库系统，其负载均衡的策略也有区别。LSM-Index的实现，使得对索引的查询处理同样满足于系统的负载均衡，主要从两个方面保证：

1. LSM-Index使用基于采样的索引区间划分方法，索引均匀分布在各个计算节点，最大程度避免了索引在集群中的分布倾斜，防止发生由索引分布不均匀引起的系统负载不均衡问题。

2. LSM-Index集成于读写分离、批量更新的底层存储，组织方式与数据相同，在这类架构的数据库中索引可以等价视为另一种形式的数据，因此也就可以适用于系统里有关负载均衡的所有算法。

综上所述，索引的均匀划分和存储架构，保证了索引查询、维护处理时的系统负载均衡，本文将在章节六中根据实验评测结果证实负载均衡对索引的有效

性。

5.4　故障处理

与小节5.3类似，LSM-Index也具有同数据一样的高可靠性。冗余副本的机制确保了一个节点下线后，该节点的索引备份依然处于可用于服务的状态。在实现LSM-Index后，一些原先在系统里从未出现的问题应该考虑。本小节讨论两种添加索引后可能出现的故障处理情况：在创建静态索引时某个节点故障，以及小节5.2.2中移除索引恢复日志后的索引重构流程。

5.4.1　创建静态索引时节点下线

在创建静态索引期间，节点可能由于本身故障或者网络断开的原因从集群中下线，此时，出于分布式数据库高可用和高可靠的要求，整个系统应该还是可提供服务的，索引也需要继续进行创建工作。LSM-Index利用数据表的冗余副本，实现了高可靠的索引构建流程。

构建静态索引时可能在两个阶段发生节点故障:

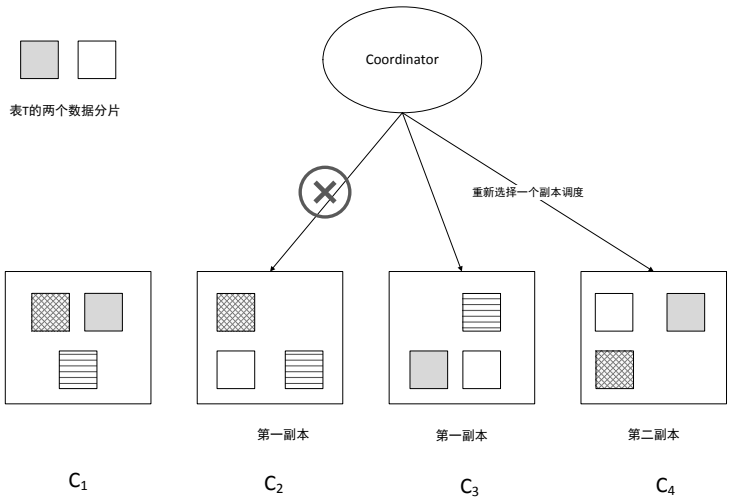

图 5.4: 局部排序阶段节点下线的处理

在局部排序阶段节点下线。如图5.4，假设表T的第一副本保存在节点C_2与C_3上，在局部排序阶段，两个节点对本地维护的数据进行局部排序，

某个时刻节点C_2故障下线，系统继续选择丢失分片的第二副本所在节点即C_4进行局部排序，到构建静态索引流程结束后再将索引复制为3个备份，构建静态索引依然可以完成。

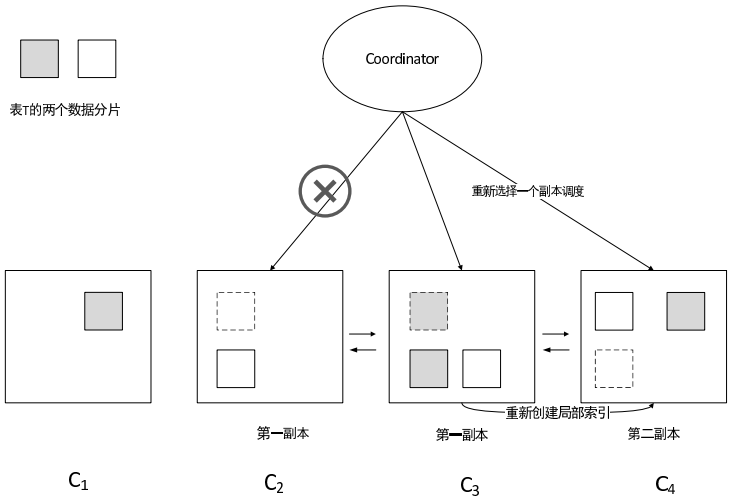

图 5.5: 全局排序阶段节点下线的处理

在全局排序阶段节点下线。如图5.5所示,图中虚线矩形代表排序后的局部索引文件，coordinator节点为其他节点分配划分完成的全局索引范围区间，调度全局排序任务。全局索引包含两个分片，第一副本分别在C_2和C_3上。某个时刻节点C_2下线后，coordinator会继续选择丢失的索引区间第二副本所在的节点，即C_4节点，进行全局排序任务调度。C_3与C_4 进行数据shuffle通信时确认C_4的局部索引仍未构建，会发送请求让C_4重新构建局部索引，以完成后续的工作。LSM-Index通过多副本的节点调度策略，实现了创建静态索引期间的故障恢复。

5.4.2 日志回放处理

日志回放指某个节点从故障状态恢复正常，需要在集群中重新上线，通过解析恢复日志重构内存增量数据的过程。如小节5.2.2描述，如果数据库系统采用了移除索引日志的优化方式，而每条索引的记录都可以由原数据变换得到，因此可以根据数据的回放日志重构增量索引$men-index$。

如图5.6，日志回放点(replay_point) 标记之前的日志已经写到磁盘，成为基准数据，这部分日志不需要回放。节点恢复时从replay_point 开始解析日志、重构数据。如果日志里包含索引的更新，则将这部分日志重构成索引记录；反之，对于每条数据操作日志，都判断是否涉及索引的更新，进而构建出索引记录。

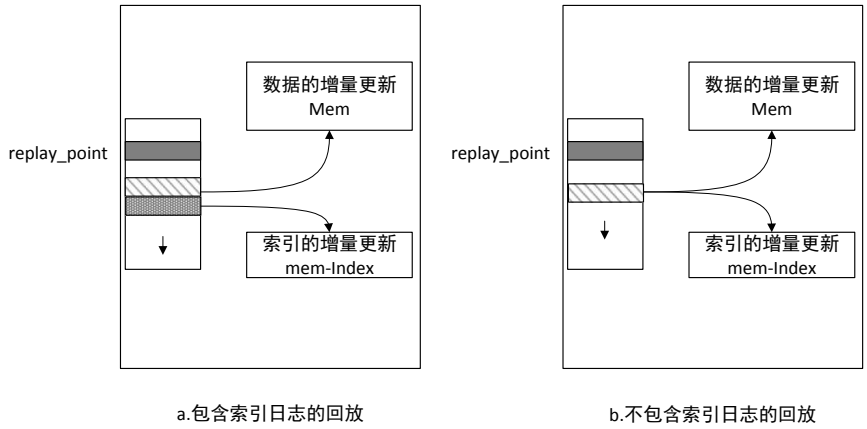

图 5.6: 两种日志回放方式

5.5　本章小结

本章节分析了读写分离架构下LSM-Index查询和维护的网络开销与磁盘交互，使用了基于冗余列的优化方法，避免回表查询产生的额外代价，提升了查询性能；提出了移除恢复日志的索引更新优化，降低了索引的维护代价；讨论了LSM-Index在两种节点下线场景中的故障处理。对于多节点维护增量更新的数据库，其索引维护需要做更多的优化；同时，基于规则的索引选择方法有一定的局限性，可以进一步实现基于代价计算的索引优化方案。这都是未来值得深入的研究工作。

第六章　实验

本文设计了一系列实验，来验证LSM-Index既能提升非主键查询的性能，也满足分布式系统负载均衡和可扩展性。本章将给出这些实验过程，并对结果进行评估和总结。

6.1　实验设置

6.1.1　实验环境

本文描述的实验全都是在由五个服务器节点组成的集群环境下进行的，服务器的硬件信息如表所示。LSM-Index索引方法最终在分布式数据库OceanBase的开源版本（Version：0.4.2.21）上实现。OceanBase是单点维护增量数据的系统，部署时将更新服务器进程（UpdateServer）和主控服务器进程（RootServer）在同一服务器启动，剩余四台服务器均启动数据存储服务器（ChunkServer）和查询处理服务器（MergeServer）两个进程。

表 6.1: 服务器节点配置表

组件	说明	备注
CPU	Inter(R)Xeon(R)E5-2680 2.00GHZ 2*12核	
内存	165G	主机内存
以太网	BCM5719 Gigabit Ethernet	千兆带宽
操作系统	CentOS release 6.5(Final)	64位

OceanBase是分布式关系数据库，能够支持跨行跨表的长事务，为了对其性能有客观的评价，实验的对照系统为MySQL 加上中间件组成的分布式方案。MySQL 系统的版本为MySQL Community 5.6.24（InnoDB引擎）[49]，部署在硬件

配置如6.1的四个服务器节点，中间件选择一款商业软件，单独部署在一台服务器节点上，配置也如同表6.1。在MySQL集群中，中间件将数据库表依据主键进行Hash划分，每个MySQL节点上存储一个分片。MySQL部署方案见图6.1。

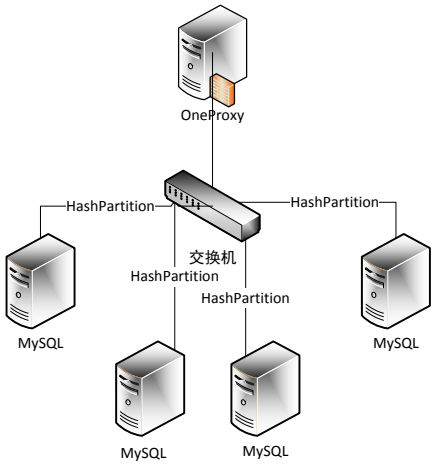

图 6.1: MySQL+ 中间件扩容方案

6.1.2　基准测试工具和数据集

实验使用两种基准测试工具，雅虎公司的测试工具YCSB[50] （Yahoo Cloud Serving Benchmark，使用github 上的最新源码编译），以及开源的多线程性能测试工具sysbench[51]（0.5版本）。实验分别用到了YCSB 和sysbench 的两个核心负载集，这些负载依据各自的测试表生成测试数据。YCSB的测试运行在表usertable上，usertable的主键属性和非主键属性信息如表6.2 所示。sysbench的测试负载基于10张测试表sbtest1∼sbtest10，测试表的属性信息均相同，见表6.3。两张表生成的测试数据集大小统一限定在1千万行，进行实验时，使用系统检测工具nmon[52]采集各节点CPU、磁盘等信息。

表 6.2: YCSB测试用表usertable信息

属性名称	是否为主键	数据类型	数据长度（Byte）
ycsb_primarykey	是	varchar	24
field0...field9	否	varchar	100

表 6.3: sysbench测试用表sbtest1...sbtest10

属性名称	是否为主键	数据类型	数据长度（Byte）
id	是	int	8
k	否	int	8
c	否	varchar	120
pad	否	varchar	60

6.2 静态索引构建性能

我们使用YCSB测试工具集，生成并导入1千万行数据到OceanBase数据库当中，并对usertable中的非主键列field0构建索引，用本文提出的算法创建静态索引。nmon检测工具监控整个过程中数据存储服务器Chunkserver的节点状态。我们选取一个服务器的状态变化结果，来评估静态索引构建的性能。

通过阅读系统日志，我们整理出批量更新过程的大致几个阶段，归纳在表6.4里，首先是发生在11:42:42至11:51:02时刻的数据表的批量更新，这个过程与原系统的流程相同，本研究并未做任何改动。之后，时刻11:53:29开始接收到coordinator （对于OceanBase 系统，指RootServer）对表usertable 创建静态索引的任务调度，进行索引列的局部排序工作，并在时刻11:54:57完成。11:55:08时刻则开始进行全局排序任务，在一分钟内完成(11:55:36)。

表 6.4: 静态索引构建过程中各个时刻的状态

标签	详细值	状态
t_1	11:42:42	对数据批量更新
t_2	11:51:33	数据的批量更新过程完成
t_3	11:53:29	局部排序开始
-	11:54:57	局部排序完成
t_4	11:55:08	全局排序开始
t_5	11:55:36	全局排序完成

这段时间内的CPU以及磁盘I/O的信息采集分别如图所示，图6.2 以及图6.3 所示。我们可以从这两幅图中得到如下结论：

1. 在包含索引构建的批量更新过程中，静态索引构建的时间相对于原数据

的增量合并过程来说，所占的比例很小，仅是后者的23%。这是因为使用了map-reduce思想的两阶段排序策略，加快了整个索引的排序过程。

2. 图6.3 显示了一个节点从批量更新阶段到构建索引阶段才会有磁盘的读入操作，这是因为局部排序需要读取数据文件，而全局排序需要读取局部索引文件。局部排序和全局排序因为要都要写排序文件，所以会产生I/O 代价。

3. 从CPU的状态变化可以看出，在局部/全局排序构建索引过程中，其CPU使用率明显高于数据的增量合并（800%）,这是因为在两个排序阶段使用了多线程并行优化，充分利用了CPU 资源。

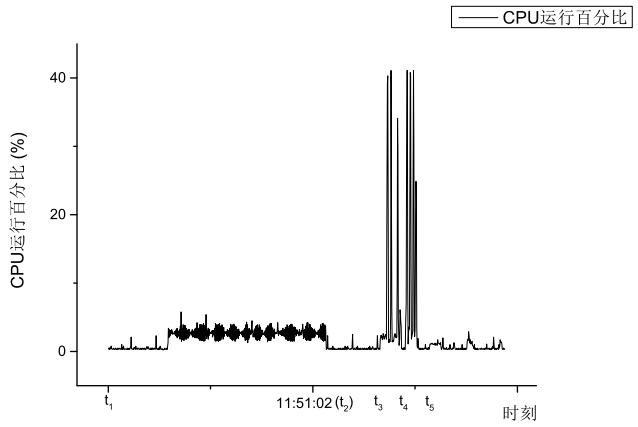

图 6.2: 创建静态索引时CPU状态

实验还观测了系统批量更新结束后，所有节点维护的索引存储空间的使用情况，因为测试数据每一项记录都是定长的，所以每个节点的索引存储空间大小比重也等于索引行数的比重。图6.4显示了1000万行数据构建的索引（三个备份）在各个节点的分布情况，实验结果说明索引在集群的节点中分布是均匀的，基于采样的分布决策方法是有效的。

至此，我们可以得到，两阶段排序的索引构建以及多线程的并行优化方法可以高效地实现索引的创建，相比于数据的增量合并过程，计算资源得到了更充分

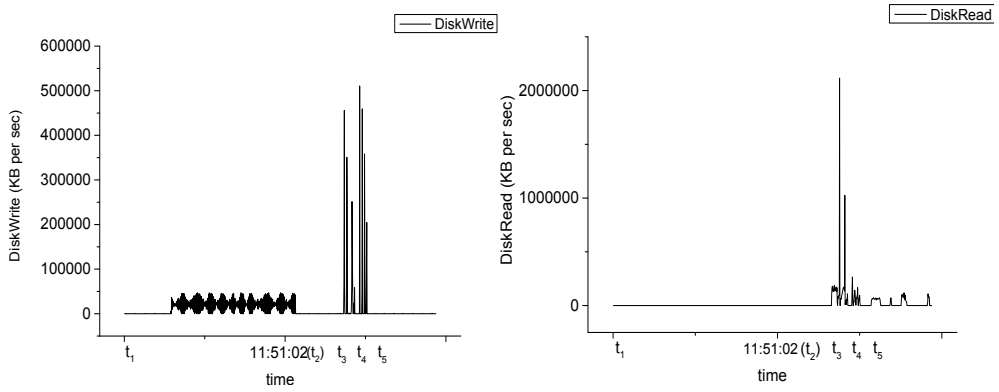

图 6.3: 创建静态索引时磁盘状态

的利用；用本文算法构建的索引是均匀的，这也有利于系统对索引访问的负载均衡。

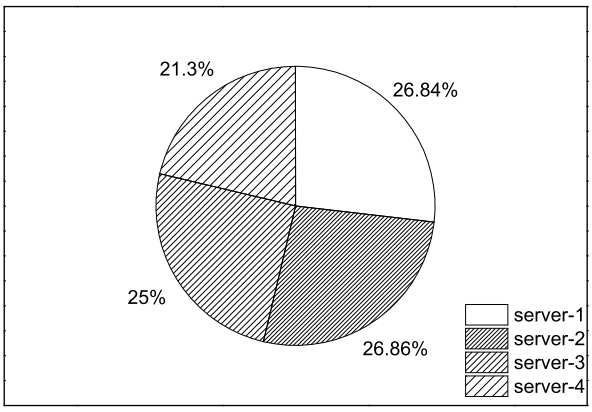

图 6.4: 批量更新后索引的分布

6.3 写入性能

因为需要维护索引与数据之间的一致性，加入索引后对系统造成的额外开销是无法避免的。本节的实验对LSM-Index造成的写入代价进行评测。我们使YCSB通过JDBC接口访问集群，并启动900个线程运行YCSB的核心数据加载任务，加载的数据表为usertable，在field0 属性上建立索引。我们对索引列数目从0增

长到15的不同负载场景进行了测试，评价的指标为每秒钟写入的次数，记为OPS（operation per second）。得到的结果如图6.5，其中优化后的写入性能指的是，使用移除索引日志的方法后系统的性能。测试结果显示，索引维护的代价是不可回避的，自索引的数目从0 增加开始，整个系统的写入性能受到了影响，在创建了五个索引时，系统写入性能是原来的88%，但通过优化手段可以将性能维持在90%以上(93%)。这是因为OceanBase 是单点维护增量更新的系统，其写入的网络通信代价较低，且索引的更新也是维护在内存中的增量数据结构中，这降低了维护索引的成本。而随着索引数目继续增长，二者的写入性能差距增大，到创建16 个索引时，未优化和优化的写入性能分别为原先的67% 和76%。

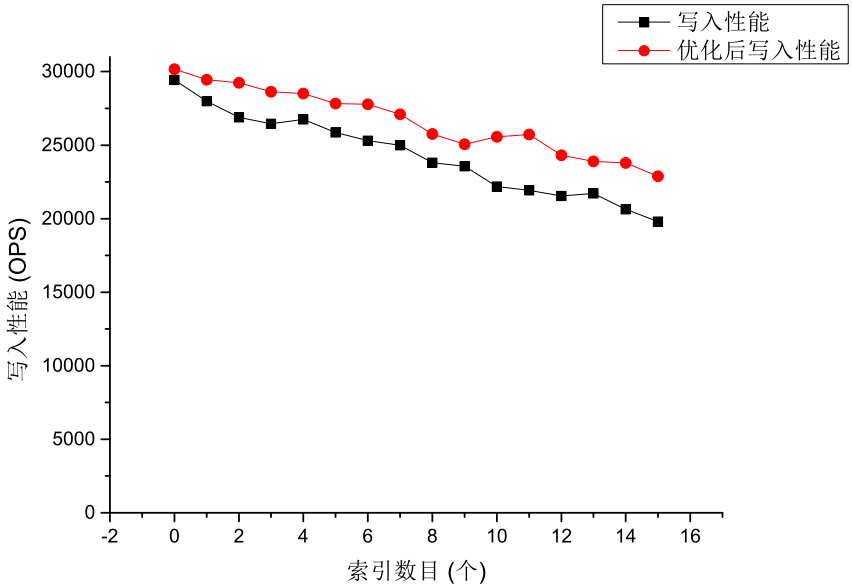

图 6.5: 900 线程下索引对系统写入性能的影响

至此可以得出的结论是，在内存中维护索引的增量更新是有效的，一张表创建五个索引时，系统以不到10% 的性能折损维护了分布式索引全局的一致性，以及，控制恢复日志的写入开销优化是有效的，优化后系统的OPS比未优化的性能明显要高。

6.4 查询性能

实验使用多线程性能测试工具sysbench来评价LSM-Index的非主键查询优化效果，sysbench 将负载运行在10张表（sbtest1至sbtest10）上，每张表的结构信息是相同的，实验在十张表的第二个属性k上创建了索引，生成的数据集大小是每张表1千万行。测试时，每次查询都随机选择一张测试表进行非主键列的查询，同样地，使用OPS作为系统性能的衡量指标。

6.4.1 非主键属性查询优化效果

表6.5描述了创建索引前后的非主键读取性能，sysbench执行随机的非主键等值查询，实验控制并发的线程数目来评测系统OPS，可以看出，因为要执行全表扫描，没有索引时的查询代价是很高的，在线程数目增长到150之后，导致了sysbench因为长时间没有收到结果回复信息而断开连接。索引的加入极其显著地提高了查询效率，150线程的并发查询系统拥有36949.07的OPS。

表 6.5: sysbench多线程测试系统读取性能结果表

线程数	OPS（未创建索引）	OPS （使用索引）
50	14.26	23407.51
75	24.98	29091.45
100	33.13	32676.57
125	40.18	33330.84
150	-	36949.07
175	-	35227.50

6.4.2 批量更新对查询性能的影响

本节实验在读写分离数据库的两种极端场景下，对索引性能进行评测，一种是索引和数据全部维护在数据库的内存组件当中（记为ALL-IN-MEM），另一种则是索引和数据全部写入到存储节点的磁盘里（记为ALL-IN-DISK）。sysbench测试工具分别测试这两个场景下非主键列等值查询的系统吞吐，控制线程数变量由50增至300，测试结果见图6.6。

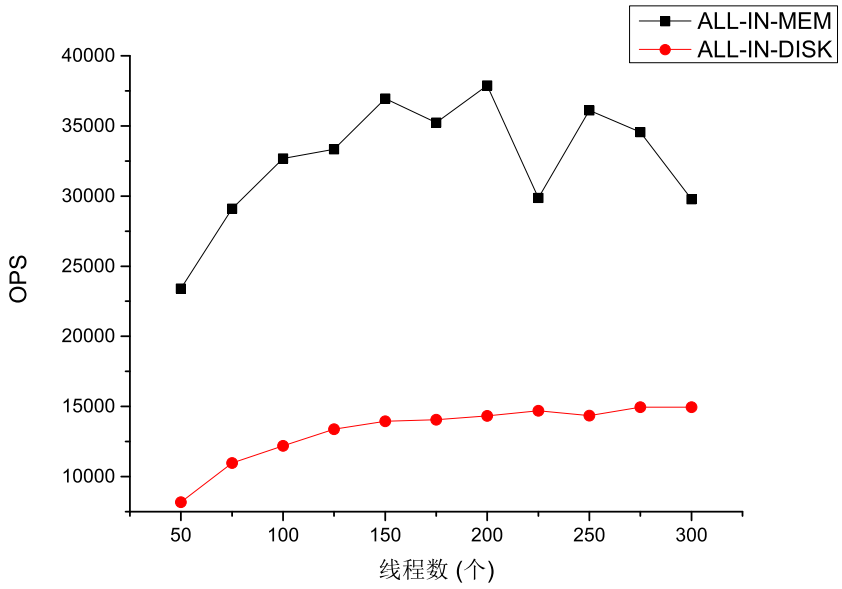

图 6.6: 两种极端场景下系统的索引查询性能

ALL-IN－MEM场景下系统的吞吐为ALL-IN-DISK的2.5倍左右，其原因在于，ALL-IN-MEM场景下所有的索引和数据都在内存中，对于OceanBase来说，每次查询都要让数据存储节点（ChunkServer）向增量数据维护节点（UpdateServer）请求更新增量，因此网络通信次数是相同的，此时内存和磁盘读取速度的差距在评测中显现出来。同时，可以观测到，当并发线程数增大时，由于触及CPU的处理上限，系统的在ALL-IN-MEM场景下的性能有所下降，而ALL-IN-DISK场景的性能瓶颈并不在CPU处理能力，所以依旧可以提高性能。

6.4.3　冗余列优化效果

在章节5.1中我们讨论了LSM-Index的查询代价，并提出了基于冗余列避免回表查询的优化方法，本实验对这种优化方法的效果进行评估。我们在十张测试表的k属性上创建索引，并在索引中追加第二个非主键信息c作为冗余列（STORING_COLUMN），使用sysbench执行并发查询，随机地从十张测试表中查询信息c，并对系统的OPS进行监测。我们测试了等值查询和范围查询两种负

载，对于范围查询来说，因为数据是随机生成的，索引列k和主键属性id存在着1对多的映射关系，平均的对应关系大致为$1:30$。实验进行得到的检测结果如图6.7。图例中with-storing表示使用冗余列优化，without-storing则表示没有优化，point-query和range-query分别代表等值查询和范围查询。我们可以从评测结果中得到如下的结论：

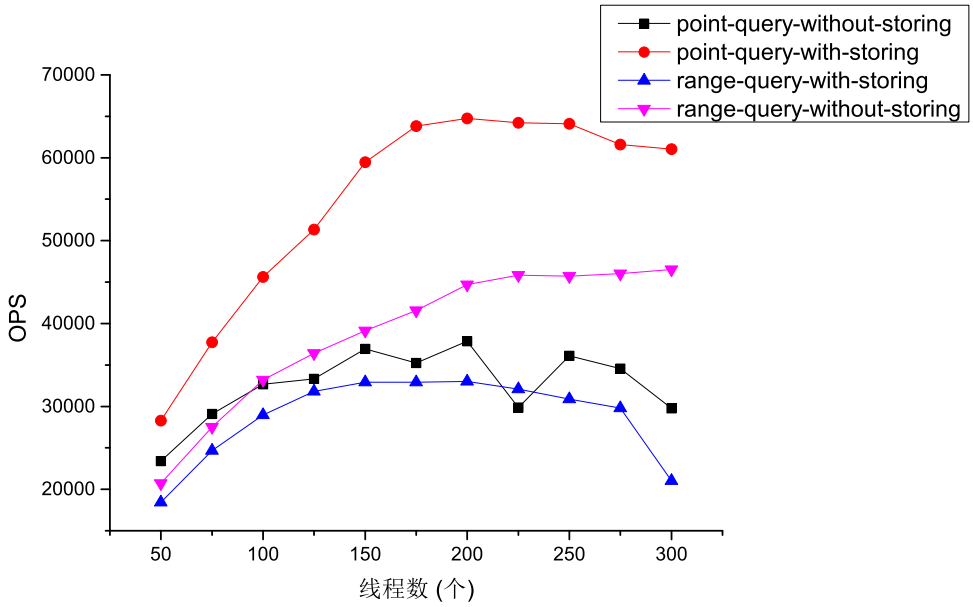

图 6.7: 冗余列避免回表的优化效果

1. 因为索引列和主键属性存在一对多的关系，因此，无论是否进行优化，范围查询的系统吞吐都没有点查询高。

2. 经过不回表优化的非主键查询性能明显高于未进行优化的性能，呈倍数关系，对于等值查询，300线程并发查询时的系统吞吐，经过不回表优化是未优化的205%；对于范围查询，优化的查询性能是未优化的220%。

可以看出，使用冗余列避免回表的优化方式是有效的，能够避免回表造成的多余网络传输和磁盘读取的开销。

6.4.4　热点数据查询对性能的影响

在数据库服务的实践中，许多情况数据的查询并不是均匀地随机选择，很多应用当中都存在读取概率很高的热点数据。本节实验模拟了在均匀分布的数据中，对满足某些分布的非主键查询的性能评测。以sysbench负载中的查询语句为例，对于SQL语句，

$$\text{SELECT } c \text{ FROM } sbtest1 \text{ WHERE } k = value;$$

在进行等值查询时，我们分别用不同的分布函数生成一个值传入$value$，实验中分别采用了均匀分布、高斯分布以及帕累托分布三种概率函数来模拟热点查询。针对10 张表共计1 亿行的数据，sysbench 启动200 个线程并发执行查询，持续时间20分钟以达到最稳定状态，并对系统读取的OPS 进行观测，实验结果如图6.8。

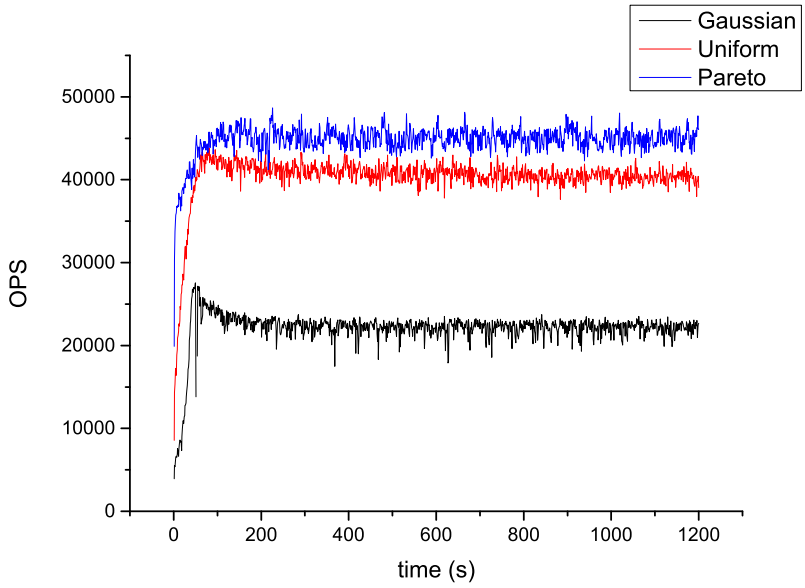

图 6.8: 不同概率分布的等值查询条件下的系统性能

查询条件服从高斯分布和均匀分布两种方式相比，高斯分布因为存在较明显的热点数据，某些非主键列的查询集中在部分存储节点，并且，因为查询条件的

倾斜度不高，服务器读取数据时的缓存效果也比均匀分布要差，所以，其查询性能要低于均匀分布。

而帕累托分布虽然索引查询的负载也并不均匀，但其查询条件的倾斜度很高，存储节点的缓存能够充分的利用起来，因而系统吞吐仍然比较高。

以上可以总结出，存在热点数据查询条件对系统的整体性能会有较为明显的影响。

6.5 事务处理能力

OceanBase是典型的读写分离架构数据库，同时也是支持ACID类型长事务的数据库系统，本小节的实验是测试系统添加索引后，其带索引的事务处理能力。我们选择MySQL Community Server与中间件的扩展方案作为对照系统，评测两种数据库扩容方案的索引性能，在两个系统上分别运行sysbench 的负载集，逐渐增加数据库连接并发数，使用每秒事务处理量（TPS）作为系统性能的衡量指标，每改变一次并发线程数，让sysbench负载测试进行10 分钟以上，并记录平均TPS。因为MySQL与OceanBase对sysbench负载的支持并不全面（如OneProxy 不支持跨库的事务，OceanBase不支持自增主键等），我们对运行的负载集做出改动，最终负载集合中的长事务包含如下操作语句：

1. 从测试表集合中随机选择一张表，记为T
2. 对表T进行十次主键上的等值查询
3. 对表T进行一次索引列上的等值查询
4. 对索引列进行一次更新操作
5. 对非索引列进行一次更新操作
6. 插入一条新的记录进入表T
7. 根据主键删除表T中的一行记录

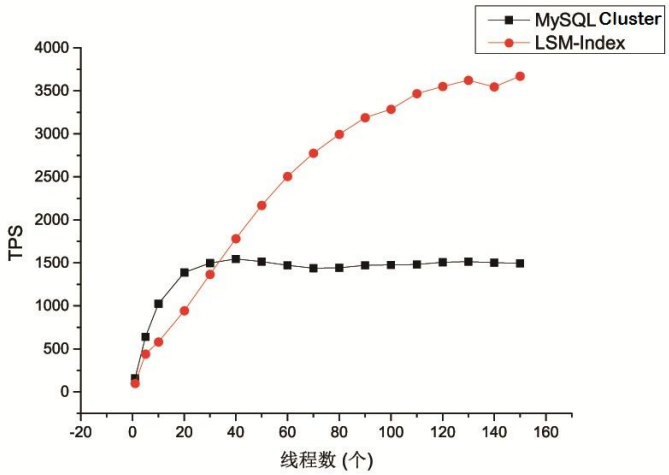

图 6.9: 使用LSM-Index时系统的可扩展性

图6.9展示了LSM-Index方案和MySQL扩容方案的性能处理测试结果，横坐标是测试工具所启动的线程数，纵坐标是系统事务处理的TPS，每个点代表运行十分钟后的平均TPS。由图可以看出，事务并发数较少的时候，MySQL与中间件组合的方案TPS比LSM-Index要高，这是因为OceanBase 分布式数据库每次事务处理都会有计算存储节点和增量更新维护节点之间的通信。当线程数增加的时候，虽然两种方案的性能均有所提升，但使用LSM-Index 索引的事务TPS增长速度明显高于MySQL集群的方案，LSM-Index索引事务处理最高达3600TPS，MySQL集群则只有1500TPS，可以总结出，LSM-Index在OceanBase上的实现在并发数和负载增加的场景上具有明显的优势。

6.6　负载均衡及可扩展性

本小节的实验将观测增加节点对带索引的事务处理性能造成的影响。

实验依旧运行小节6.5中的sysbench负载，负载中包含对索引的更新和查询，线程数设定为常量100，通过控制集群中的节点个数，观测系统使用索引时的事务处理性能。实验使用OPS和CPU负载两个指标评测系统的负载均衡和可扩展性，每增加一个存储计算节点（在OceanBase中指MergeServer/ChunkServer共同部署的

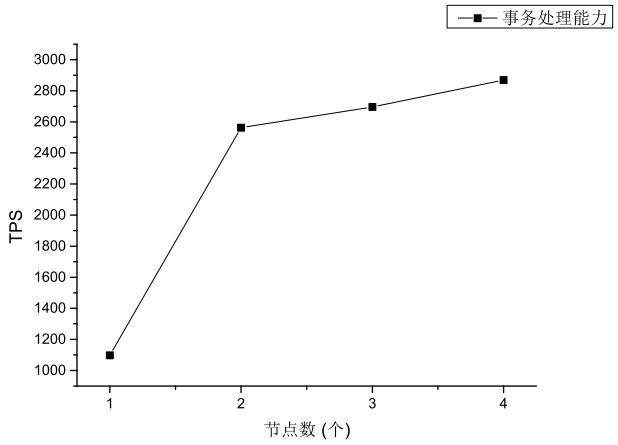

图 6.10: 使用LSM-Index时系统的可扩展性

服务器节点），在持续二十分钟的负载测试后观测系统的平均TPS，同时，整个过程记录新增节点的CPU运作信息。

实验评测的结果见图6.10及图6.11。图6.10 中的结果表明，在事务中维护和使用了LSM-Index的情况下，系统从1个节点扩展到4个节点的过程中，TPS 增长至260%。这是因为在增加计算节点后，系统执行事务的并发度得到了提高，且索引的存储分布和查询负载被新增的节点均衡，从而改善了系统的事务处理性能。

图6.11a，b，c，d分别表示了整个测试过程中，各个节点在加入集群后的CPU负载变化，依次可以看出，每新增一个节点，都可以有效地降低其他节点的事务处理负载，第一个节点的CPU负载由近100%降至接近25%。这说明，LSM-Index 适应于这类分布式架构的负载均衡和可扩展性，通过增加节点的方式提升集群使用索引处理事务的能力。

6.7 UAT环境测试

我们还将LSM-Index应用于某国有银行企业的部分系统迁移工作，该银行企业尝试将一些非关键的离线业务从DB2系统[53]迁移到国产开源的数据库OceanBase当中，我们将实现LSM-Index的OceanBase运行在银行企业的真实UAT测试环境

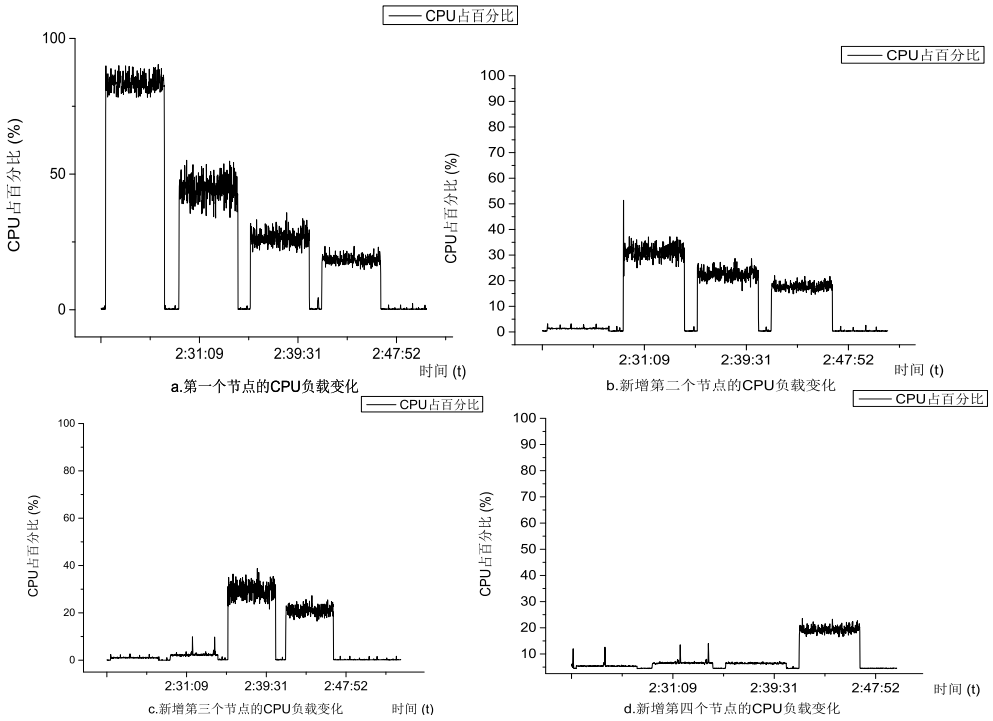

图 6.11: 使用LSM-Index时系统的负载均衡

下，并对比DB2和OceanBase对带索引事务的响应时间。由于企业环境的特殊性，出于保密目的本文隐去了业务系统的逻辑细节和实验环境，对于供应链系统的UAT 测试，有九个和索引相关的查询，实验数据包含银行企业在该系统上六个月的数据总量，原系统逻辑要求DB2的单次查询响应时间低于0.9 秒，我们观测了OceanBase 在响应这些和索引相关的查询的性能，如图6.12所示。

图中可以看出，将系统迁移到实现LSM-Index的数据库系统中，UAT-1，UAT-3，UAT-5，UAT-7，UAT-8的响应时间比DB2要短，这是因为大数据量下分布式并行查询体现出较好的性能。对于其余的交易查询，LSM-Index的性能比DB2要差，这是因为这些交易查询里包含大量的聚合运算与多表连接，而OceanBase对这些操作的优化程度并不高，导致了整体查询性能低于DB2，但即便如此，所有的交易查询均满足原系统的0.9秒标准，这说明了，带索引的交易查询由DB2迁移到分布式数据库OceanBase上是可行的。

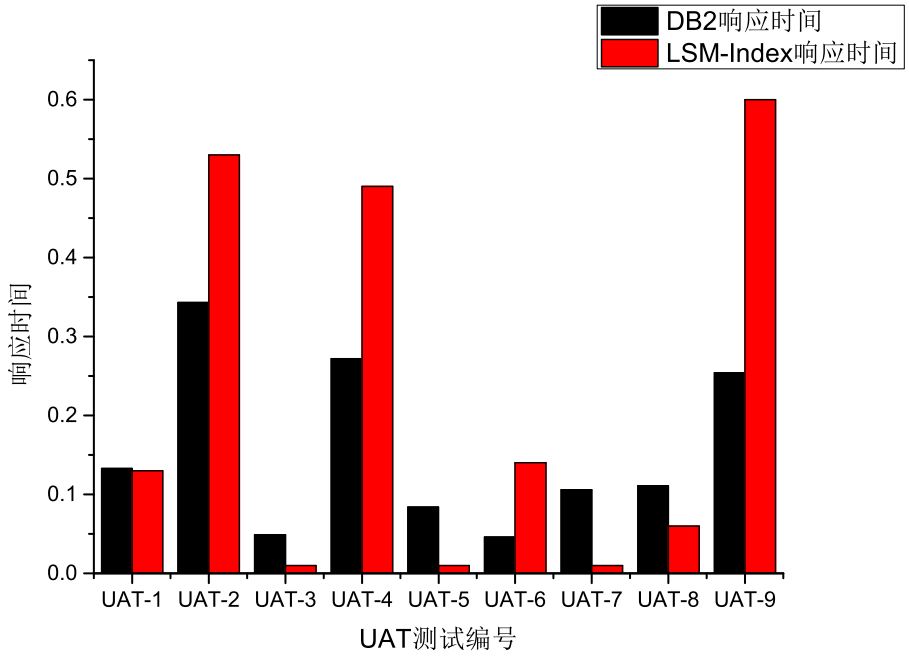

图 6.12: UAT测试环境单次查询响应时间

6.8 章节小结

为了充分说明LSM-Index对读写分离架构数据库非主键性能提升的有效性，以及适应于原系统的负载均衡和可扩展性，本文的研究工作设置了充足的实验，对观测结果均做出了总结归纳。

本文首先检测了新的批量更新算法创建索引的性能，实验结果显示，相比于原数据的增量合并，创建静态索引有效地利用了两阶段排序和并行计算能力，使其有很高的创建效率，1 千万数据量的表格，系统创建索引的过程不到3分钟；批量更新结束后，各个节点维护的索引数据量是均匀分布的，说明了基于采样的索引分布决策算法的有效性。

实验还测试了维护索引对系统性能的写入影响，并评测了不写索引恢复日志的优化效果，实验结果显示维护五张索引时系统的性能依旧保持在90%以上，说明了索引维护算法及其优化的有效性。在索引查询性能的实验里，本文从索引

对非主键查询性能的提升、批量更新对索引性能的影响、冗余列不回表的优化效果以及数据热点查询对性能的影响方面设计实验来评测LSM-Index方法的有效性，实验结果表明，LSM-Index能极大地提高非主键查询性能；冗余列不回表优化能将索引性能提升至200%以上；在不同的批量更新场景及数据热点查询场景中，LSM-Index 表现出不同的查询性能。

本文亦测试了LSM-Index在支持ACID长事务类型的分布式数据库中的性能，在与MySQL扩容方案的事务处理对比实验当中，LSM-Index在高并发和高负载场景下显现出更好的事务处理能力。

最后，我们测试了使用LSM-Index索引时系统的可扩展性和负载均衡情况，系统在处理带索引的事务时加入新的节点，能够有效地提升整体性能，降低其他节点的负载，说明LSM-Index对原系统的负载均衡和可扩展性有良好的适应性。

根据以上的实验结果，我们可以得出结论，LSM-Index方法将索引与数据存储机制集成，索引更新维护在内存当中，适时与永久存储中的索引记录合并更新的方法是正确有效的。LSM-Index能够高效、正确地解决读写分离、批量更新架构数据库的创建索引的问题，既有效地提升了系统非主键查询性能，对系统的负载均衡和可扩展性也表现出良好的适应性。

第七章　总结和展望

回顾数据库短短几十年的发展历史，其系统架构因为数据规模的高速增长、数据模型和应用需求的变化、计算机硬件和技术的进步等因素而进行着持续、多元的衍变。但无论何种架构的数据库系统都无法适用于所有应用，不同的数据库架构都有其独特的性质和最佳的应用场景。基于LSM-tree思想实现的读写分离、批量更新架构的分布式数据库，适用于对海量数据写入较集中且性能要求较高的场景，其高效的写入、可扩展性使得这类架构的数据库逐渐成为互联网企业、银行金融行业应对大数据的热门选择。然而，这类架构的数据库在非主键查询性能上有明显的缺陷，要提高读取性能，需要对非主键查询属性构建二级索引。由于读写分离、批量更新的特殊架构性质，在这类数据库上创建索引存在困难：无法基于海量数据高效创建索引、难以获得数据分布信息决定索引分布、索引不容易满足负载均衡性。本文针对读写分离、批量更新存储机制的数据库架构，提出高效的索引方法LSM-Index，能够在这类架构的数据库上有效的创建分布均匀、负载均衡的索引机制保证系统可扩展性和可用性。本文的工作内容如下：

首先，本文解决了在海量数据规模下针对读写分离架构数据库创建索引时的问题。主要方法有：设计与数据存储结构相同的索引，创建索引期间发生的更新维护在内存中，从而保证了索引的正确性和可用性；采用索引的延迟生效机制，通过批量更新在全局的数据版本更新后开始创建静态索引，确认索引与数据一致后才使索引有效，实现了不阻塞事务的索引创建，提高数据库可用性；基于采样的两阶段排序和并行任务调度方法，提高全局索引的创建效率，且使索引分布均匀，以满足负载均衡。

其次，本文分别对这类架构数据库的分布式索引设计了维护和查询算法。并根据系统特点，详细分析了这些算法的网络开销和磁盘代价，提出了两种优化方法：使用了移除索引恢复日志来降低索引维护成本，提出了冗余列方法避免索引的回表查询，提高了索引的查询性能。并设计了索引的回放方法使索引满足系统的故障恢复要求，同时，利用多副本调度任务的策略增强了索引创建阶段的容错性。

最后，本文选择在典型的读写分离数据库OceanBase上实现LSM-Index，并开展充足的实验；在企业真实的UAT环境验证了LSM-Index的高效性、可扩展性以及负载均衡特性。实验结果表明我们方法的可靠性、可用性和有效性。

与此同时，本文提出的方法在以下两个方面可以改进和优化：

1. **不同数据库变种的个性定制优化。** LSM-Index对读写分离、批量更新架构数据库具有通用性。但这类数据库种类繁多，实现细节不尽相同，如写恢复日志与修改内存组件的先后顺序，维护增量更新的方法等，在不同变种的数据库实现LSM-Index时，可以通过充足的实验测试，不断改进优化索引性能。

2. **基于查询代价的索引选择策略。**本文在索引的查询优化方面，使用的是基于规则的索引选择。在某些时候，对一个查询会有多个索引可以选择，基于规则的索引选择方式不适用于太过复杂的使用场景，容易出现错误判断，影响查询效率。而基于查询代价的索引选择，能够预估每个索引的查询开销，生成最适合的物理执行计划。

参考文献

[1] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.

[2] 阳振坤. Oceanbase 关系数据库架构. 华东师范大学学报: 自然科学版, (5):141–148, 2014.

[3] K Ganesh, Sanjay Mohapatra, SP Anbuudayasankar, and P Sivakumar. User acceptance test. In Enterprise Resource Planning, pages 123–127. Springer, 2014.

[4] E. F. Codd. A relational model of data for large shared data banks. Commun. ACM, 13(6):377–387, 1970.

[5] Charles W. Bachman. The programmer as navigator. Commun. ACM, 16(11):635–658, 1973.

[6] Michael J Kamfonas. Recursive hierarchies: The relational taboo. The Relational Journal, 27, 1992.

[7] David Kroenke and David Kroenke. Database processing. Technical report, Science Research Associates Reading, MA, 1977.

[8] Guy Harrison. Five database technology trends. the NoCOUG Journal, 29(1):4–9, 2015.

[9] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. IDC iView: IDC Analyze the future, 2007:1–16, 2012.

[10] Roderic Geoffrey Galton Cattell, Douglas K Barry, Dirk Bartels, Mark Berler, J-eff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. The object database standard: ODMG 2.0, volume 131. Morgan Kaufmann Publishers San Mateo, 1997.

[11] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. IEEE data engineering bulletin, 22:3, 1999.

[12] Michael Stonebraker. The case for shared nothing. IEEE Database Eng. Bull., 9(1):4–9, 1986.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):4, 2008.

[14] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. BMC bioinformatics, 11(Suppl 12):S1, 2010.

[15] Michael Stonebraker and Uğur Çetintemel. ” one size fits all”: an idea whose time has come and gone. In Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on, pages 2–11. IEEE, 2005.

[16] Kristina Chodorow. MongoDB: the definitive guide. ” O’Reilly Media, Inc.”, 2013.

[17] 叶纯敏. 中国去“ioe”之路. 金融科技时代, 22(8):27–32, 2014.

[18] 杨传辉. 大规模分布式存储系统-原理解析与架构实战. 机械工业出版社, 2014.

[19] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. Proceedings of the VLDB Endowment, 2(2):1664–1665, 2009.

[20] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. IEEE Data Eng. Bull., 36(2):21–27, 2013.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, pages 654–663. ACM, 1997.

[22] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In ACM SIGOPS Operating Systems Review, volume 41, pages 205–220. ACM, 2007.

[24] Rick Cattell. Scalable sql and nosql data stores. ACM SIGMOD Record, 39(4):12–27, 2011.

[25] Jim Melton and Alan R Simon. Understanding the new SQL: a complete guide. Morgan Kaufmann, 1993.

[26] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR), 15(4):287–317, 1983.

[27] J Chris Anderson, Jan Lehnardt, and Noah Slater. CouchDB: the definitive guide. " O'Reilly Media, Inc.", 2010.

[28] Claudio Tesoriero. Getting Started with OrientDB. Packt Publishing Ltd, 2013.

[29] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS), 31(3):8, 2013.

[30] apache.org. Apache GEODE. http://geode.incubator.apache.org/. [Online; accessed 4-April-2016].

[31] Rusty Klophaus. Riak core: building distributed applications without shared state. In ACM SIGPLAN Commercial Users of Functional Programming, page 14. ACM, 2010.

[32] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems (TODS), 8(4):465–483, 1983.

[33] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In ACM SIGMOD Record, volume 24, pages 1–10. ACM, 1995.

[34] Moshe Weinfeld. Deuteronomy 1-11: a new translation with introduction and commentary, volume 5. Anchor Bible, 1991.

[35] Panos Vassiliadis. A survey of extract–transform–load technology. International Journal of Data Warehousing and Mining (IJDWM), 5(3):1–27, 2009.

[36] Huawei-Hadoop. Secondary Index for HBase. https://github.com/Huawei-Hadoop/hindex. [Online; accessed 4-April-2016].

[37] 孟必平，王腾蛟，李红燕，杨冬青. 分片位图索引: 一种适用于云数据管理的辅助索引机制. 计算机学报, 35(11):2306–2316, 2012.

[38] Yongqiang Zou, Jia Liu, Shicai Wang, Li Zha, and Zhiwei Xu. Ccindex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries. NPC, 10:247–261, 2010.

[39] Wei Tan, Sandeep Tata, Yuzhe Tang, and Liana L Fong. Diff-index: Differentiated index in distributed log-structured data stores. In EDBT, pages 700–711, 2014.

[40] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. Proceedings of the VLDB Endowment, 7(10):841–852, 2014.

[41] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. Asterixdb: A scalable, open source bdms. Proceedings of the VLDB Endowment, 7(14):1905–1916, 2014.

[42] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient b-tree based indexing for cloud data processing. Proceedings of the VLDB Endowment, 3(1-2):1207–1218, 2010.

[43] Hosagrahar V Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In Proceedings of the 31st international conference on Very large data bases, pages 661–672. VLDB Endowment, 2005.

[44] Yufei Tao and Dimitris Papadias. Adaptive index structures. In Proceedings of the 28th international conference on Very Large Data Bases, pages 418–429. VLDB Endowment, 2002.

[45] Bruce Momjian. PostgreSQL: introduction and concepts, volume 192. Addison-Wesley New York, 2001.

[46] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. Advances in neural information processing systems, 19:281, 2007.

[47] M Tamer Özsu and Patrick Valduriez. Principles of distributed database systems. Springer Science & Business Media, 2011.

[48] Kapali P. Eswaran. Placement of records in a file and file allocation in a computer. In IFIP Congress, pages 304–307, 1974.

[49] Apache. MySQL.com. `http://dev.mysql.com/downloads/mysql/`. [Online; accessed 29-March-2016].

[50] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM symposium on Cloud computing, pages 143–154. ACM, 2010.

[51] Alexey Kopytov. SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`. [Online; accessed 29-March-2016].

[52] Nigel Griffiths. nmon performance: A free tool to analyze aix and linux performance, 2003.

[53] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: integrated automatic physical database design. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pages 1087–1097. VLDB Endowment, 2004.

致　　谢

三年时间很快，我的研究生生活接近了尾声，回想起自己刚来到华东师范大学时的情形和一路走来奋斗的点点滴滴，感慨不已。三年的研究生生活让我在各个方面有了很大的提升和收获，包括学习，工作和处世等等，在严谨的科研氛围中学会了自主研究，在协同合作的工作中得到了成功，在充实的校园生活中收获了惊喜。我三年的时间一半是在交通银行一半是在海量所度过的。在海量所这个大家庭中，我遇到了许多的良师益友，不仅在学术研究上给予我很多指导，而且在生活上也给了我不少关心和帮助；在交行的工作中，我不仅结识了志同道合的工作伙伴们，而且积累了工作经验。值此，我要向那些关心，支持和帮助过我的老师，朋友和家人们表示深深的感谢和美好的祝愿。

首先，我要向我的导师宫学庆教授表示感谢。三年来，宫老师不仅在学术上引导我研究，还在生活上给我关系和帮助。交通银行的共事期间，宫老师严谨的工作态度和渊博的专业知识让我受益匪浅，在忙碌的工作之余指导我完成论文并成功发表，让我感激不已。然后，我还要特别感谢指导师张蓉教授，本论文是在她的悉心指导下完成的。从论文的选题到撰写完成，每一步都倾注了她大量的心血。并且在三年读研期间，张老师无论是在生活上还是在学业上都给予我了无微不至的关怀。

同时，本论文的完成也离不开实验室的其他老师们。感谢钱卫宁老师在论文开题和论文前期开展时提出的建设性意见，感谢张召老师在科研项目开发过程中的指导。感谢周傲英老师每次发人深省的教诲，以及感谢王晓玲、何晓丰、金澈清、高明、蔡鹏等各位老师们的传道授业解惑之情。

另外，我还要感谢身边的小伙伴们。感谢隆飞和茅潇潇同学，感谢他们为开发项目做出的努力；感谢胡爽，庞天泽，李永锋，樊秋实，刘骁，张晨东，朱燕超等交行组的小伙伴们，两年的时间能和他们一起在浦东工作和生活，我感到非常开心；感谢刘孟占，赵琼，江晶等交行的同事们，感谢他们在工作期间对我的指点和关心；感谢晁平复，高祎璠等华为项目组的成员们，感谢他们在我刚入学时给我的帮助；还要感谢实验室项目组的所有成员们。总之，感谢一路走来关心和支持我的朋友们，让我的研究生生活变得充实又有意义。

最后，我要特别感谢我的父母，以及其他给我支持和关心的家人们，感谢他们含辛茹苦地抚育和对我求知深造的理解和支持，没有他们的鼓励和关怀，就没有今天的我。这些年来他们是我坚强的后盾，每当我迷茫失落时，总能在背后默默地给我支持，给我不断向前的动力。

谨以此文向所有关心和帮助过我的人们表示衷心的感谢!

翁海星

二零一六年三月二十九日

攻读硕士学位期间发表论文和科研情况

■ 已发表或录用的论文

[1] **翁海星**, 宫学庆, 朱燕超, 胡华梁 集群环境下分布式索引的实现, 计算机应用2016, 1-7. (CSCD)

■ 参与的科研课题

[1]国家自然科学基金重点项目（U1401256），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管理系统研究与开发（2015AA015307），2015－2017，参加人