2016届研究生硕士学位论文

分类号：_____ 学校代号: 10269

密　级：_____ 学　　号: 51131500007

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OceanBase的分布式大表连接与优化

院　　系:	计算机科学与软件工程学院
专　　业:	软件工程
研 究 方 向:	内存数据库
指 导 教 师:	周傲英　教授
学位申请人:	樊秋实

2016 年4 月5 日

Dissertation for master's degree in 2016 School Code: 10269
Student ID: 51131500007

East China Normal University

Title: DISTRIBUTED JOINS AND OPTIMIZATION FOR BIG TABLE BASED ON DATABASE OCEANBASE

Department:	Computer Science and Software Engineering Institute
Major:	Software Engineering
Research direction:	In-memory Database
Supervisor:	Prof. ZHOU Ao-ying
Candidate:	FAN Qiushi

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OceanBase的分布式大表连接与优化》，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OceanBase的分布式大表连接与优化》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）。

樊秋实 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
钱卫宁	教授	华东师范大学	**主席**
金澈清	教授	华东师范大学	
周敏奇	副教授	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	

摘　　要

在大数据时代的很多应用中，数据规模达到PB、EB甚至ZB量级，特别是像秒杀和抢购等“现象级”的互联网应用。在这些应用中传统的数据库系统显得力不从心，因此如何管理和利用大数据已成为工业界和学术界共同关注的问题。由于硬件的更新和应用的驱动，NoSQL 数据库和分布式数据库等技术得到了很大的发展。这些数据库部分解决了海量数据管理和利用所遇到的部分挑战，攻克了一些大数据应用的难题。但对一些复杂数据管理任务，如数据库上的大表连接，其表现还不太尽如人意。

连接操作是关系型数据库最重要的算子之一。如何在海量、分布式情形下保证连接操作的正确性与可用性是一个非常具有挑战的任务，特别是对读写分离架构的数据库来说。

为应对海量数据处理，可扩展性是分布式数据库研究的重点，如HBase,Shark都是基于这个目标而设计的。对偏向于OLTP的事务型数据库来说，分布式存储的数据可能导致效率非常低下的分布式事务。为了解决这一问题，近几年一种读写分离架构的数据库成为分布式数据库发展的一个分支。该架构把所有更新事务都放在一个节点上，形成增量数据，避免了分布式事务，同时把基线数据分布在多个节点上，实现了高可扩展性。OceanBase 数据库就是这一架构的典型代表。该架构具有高效响应事务的优点，但同时也增加了查询操作的复杂性。每一次查询都要首先将基线数据和增量数据合并，然后返回合并后的数据。可以看出，当执行大表连接时，大量的网络传输会导致该架构的处理性能明显降低。

为了优化该架构下大表连接的效率，本文以开源数据库OceanBase为基础，针

对其架构特点，设计并实现了两种大表连接优化算法：SemiJoin和分布式排序归并连接。并在传统连接算法的基础上，针对读写分离架构的特点，对这两种算法进行了优化。

本文的主要贡献如下：

1. 针对OceanBase数据库架构特点，设计并实现了SemiJoin算法。通过并行计算增加了对大表数据过滤的速度，同时减少网络上的数据传输量，从而极大地提高了半连接的效率。并通过一系列的对比实验证明了优化后算法的高效率。

2. 针对OceanBase数据库架构特点，设计并实现了分布式排序归并连接算法。该算法将连接属性分为多个范围，每个范围内的数据并行地做排序归并连接。特别针对增量数据，本文提出了“最大范围”算法，分别对待基线和增量数据，极大地提高了连接效率。并通过一系列的对比实验验证了优化后算法的高效率。

3. 提出了在读写分离架构下，不首先合并数据，而是对基线和增量数据分别处理的思路。这种思路不仅仅应用在连接操作上，而且对同类数据库的查询引擎进行优化时也可以借鉴上述思路。

关键词: 分布式数据库，查询引擎，查询优化，半连接，分布式排序归并连接，并行计算

Abstract

In the era of big data, data sizes of many applications reach PB, EB even ZB, especially for seckill, rush to purchase and other phenomenal applications in the Internet. In these applications, the traditional database systems appear to be inadequate. Therefore, how to manage and utilize big data has become a problem which industry and academia have to pay attention on. Because of the hardware updating and application-driven, NoSQL databases and distributed database techniques have been greatly developed. These databases are a part of the solution for the challenges of massive data management and they also solve some big data applications. But for some complex data management tasks such as big table joins on the database, its performance is still not satisfactory.

Join operator is one of the most important operators in a relational database. How to ensure the correctness and availability of join operation in the massive, distributed situation is a very challenging task, especially for the read/write splitting databases.

To handle massive data processing, scalability is the focus of research for distributed databases. Many databases, such as HBase, Shark and so on, are designed based on this goal. For OLTP transactional databases, distributed data may lead to a very serious problem: highly inefficient distributed transactions. To solve this problem, in recent years, read/write splitting database has become a branch of distributed database systems. This architecture puts all updating data on a single node to avoid the distributed transactions, and puts the baseline data on multiple nodes to achieve high scalability. OceanBase,

which is a distributed database, is a typical representation of this architecture. This architecture is able to efficiently response transactions, but also increases the complexity of the query operations. Every query can be done after merging baseline and incremental data. For joining multiple large tables, the performance of this architecture will be significantly reduced because of a large amount of network traffic.

To optimize the efficiency of large table joins in this architecture, we design and implement SemiJoin and distributed sort merge join on two large tables based on OceanBase. Furthermore, we optimize the traditional join algorithm based on the features of the read/write splitting architecture.

The main contributions of this paper are as follows:

1. Design and implement Semi-join algorithm based on the architectural features of OceanBase. A very efficient Semi-join operator is obtained by increasing the speed of table filtering and reducing the network traffic. The efficiency of the optimized algorithm is verified by a series of comparative experiments.

2. Design and implement distributed sort merge join algorithm in terms of architectural features of OceanBase. After dividing the values of join attribute into multiple ranges, the algorithm performs the sort merge join in each range parallelly. Especially for incremental data, a ”maximum range” algorithm is proposed. This algorithm treats baseline and incremental data separately, greatly improves the efficiency of join. The efficiency of the optimized algorithm is demonstrated by a series of comparative experiments.

3. Present a train of thought that we can treat baseline and incremental data separately instead of merging them at first. This point can be used not only on join operations, but also on the design of the query engine associated with the same architecture.

Key Words: *Distributed Database, Query Engine, Query Optimization, SemiJoin, Distributed Sort Merge Join, Parallel Computing.*

目录

插图

表格

第一章　绪 论

1.1　研究背景

在学术界，《Nature》于2008年推出Big Data专刊，旨在探讨如何应对处理海量数据时所面临的挑战。2011 年2月，《Science》也推出专刊《Dealing with Data》，试图探讨如何管理和利用海量数据。不仅仅学术界关注海量数据的利用和处理，当前如何更好地管理和利用大数据已经成为政府和工业界普遍关注的话题。

美国奥巴马政府在2012年3月提出了“大数据研究和发展倡议”，期望利用大数据技术在科学研究、生物医学、环境等领域得到相应突破。在国内，2015年李克强总理提出了“互联网+”行动计划，该计划旨在以创新驱动发展，实现互联网和其他传统行业的融合，发展大数据时代的信息技术。在技术上，如何支撑国家的大数据发展战略成为一个亟待解决的问题，特别是作为国家信息化程度衡量标准的数据库技术。

传统数据库产品在过去的几十年里获得了巨大的成功。然而，在大数据时代，由于硬件更新、成本控制和应用创新等缘故，传统关系型数据库产品正面临着极大的考验。近年来，互联网行业的创新推动了NoSQL[1] 数据库的迅速发展，出现了如Hbase[2]和MongoDb[3]等分布式数据库产品。顺应潮流，阿里巴巴首先提出“去IOE”（IBM[4]的主机、Oracle[5]的数据库和EMC的存储），旨在应对像“双11”和“秒杀”等现象级的应用。随后，出于国家安全的考虑，政府提出了建设“安全可靠、自主可控”信息技术的战略，国内掀起了一股“去IOE”的风潮，大型企业和学术研究机构纷纷参与研制分布式数据库产品。

在中国这样一个人口大国，很多的互联网应用都是非常庞大的，数据规模往往达到PB、EB或ZB级，我们亟需在数据库产品和技术方面获得突破。虽然NoSql数据库攻克了一些大数据应用难题，但是NoSql 数据库的数据模型比较简单，以牺牲数据一致性实现更高的数据库性能。这对于事务型互联网应用来说，是不能满足需求的。例如淘宝的“双11”促销活动，需要支持上亿用户的订单请求，每秒处理十几万的事务，不仅需要提高系统可用性，缩短用户响应时间以改善用户购物体验，而且还必须要保证数据的一致性，这是对数据库技术的极大挑战。

海量数据管理的需求，以及硬件的更新，使得分布式数据库成为一个合理的选择。但依据“CAP 定理”[6]，分布式数据库必然要在一致性和可用性之间取舍，想要提高数据库的一致性，就必须降低系统的可用性，反之亦然。在“CAP”理论的限制下，大家纷纷在数据库的架构上寻求突破。例如谷歌公司最新公布的基于Spanner[7]的分布式关系型数据库F1，由前Facebook工程师创办的号称世界上最快的分布式关系型数据库MemSQL[8]等。此外，一种新型的读写分离的架构逐渐流行开来，其代表有HBase和已经全面支撑阿里业务的分布式关系型数据库OceanBase[9] 等。这些数据库在一定程度上解决了业务和应用对数据库的需求，缓解了应用的压力，逐渐成为大家研究的热点。

对于关系型数据库，查询优化不仅是一项关键技术，而且体现了关系模型的优点。对用户而言，查询优化技术不仅缩短了用户响应时间，而且用户在提交查询时可以不必关注如何最好地表达查询，方便用户使用。故查询优化技术一直是数据库领域的研究热点。对于读写分离架构的数据库来说，查询优化还是有待进一步探索的领域。大量的实验证明：该架构在表连接操作上性能远低于传统数据库产品，特别是大表连接。因此，作者以此类架构上的查询优化为切入点研究大表连接优化的问题。

1.2 研究现状

连接操作是关系型数据库最重要的数据分析操作之一。以某国有银行历史库

交易为例，所有的查询sql中单表查询占11.3%，两张表的连接查询占65.2%，多表的连接查询占23.5%。表连接操作分为内连接和外连接。其中，内连接指连接结果仅包含符合连接条件的行，参与连接的两个表都应该符合连接条件。外连接是指连接结果不仅包含符合连接条件的行同时也包含自身不符合条件的行。外连接具体分为：左外连接、右外连接和全外连接。左外连接是指：左边表数据行全部保留，右边表保留符合连接条件的行。右外连接是指右边表数据行全部保留，左边表保留符合连接条件的行。全外连接则是左边表和右边表的数据都全部保留。

本文主要讨论内连接。实现内连接的算法有很多种，传统的内连接算法有嵌套循环连接、哈希连接和排序归并连接等，这些连接算法在集中式关系数据库中的表现很好。但是在分布式数据库场景下，数据可能分散地存到多个物理节点上，导致连接效率不仅仅受到磁盘读取、CPU利用率、网络传输速度和网络稳定性的影响，而且也受到基线与增量数据合并等因素的影响。传统的连接算法没有充分考虑到这些因素，因此，在分布式数据库上改进传统连接算法以提高数据库连接性能是非常必要的。例如数据仓库HIVE[10]，它构建在Hadoop[11] 基础设施之上，通过HQL[12]语言来查询存放在HDFS[13]上的数据。但是，HQL只是一种类SQL语言，它在HIVE内部最终转化为Map/Reduce[14] 任务。因此，针对HIVE上的表连接操作，本质上是通过Map/Reduce 并行计算的。虽然局部使用了嵌套循环连接或是哈希连接等算法，但是在整体上，它利用了分布式数据库的特点，在多个节点上并行地执行连接操作，通过并行计算来提高连接的效率。

如何在海量数据的情况下保证连接操作的正确性与实时性是一个很难解决的技术。这一问题对具有读写分离架构的数据库来说更为突出。该架构数据库的基线数据存储在磁盘上，增量数据采用类似于Log-Structured Merge Tree[15]的结构存储在内存中，当修改增量到达一定大小时，增量数据会被物化到磁盘上。与很多基于Log-Structured Merge Tree 架构的数据库（例如LevelDB[16]，HBase，Cassandra[17]等）相似，这种架构的数据库能够极大地提高事务的处理性能，并且具有良好的可扩展性。但是它的缺点是在做连接操作之前必须先把基线数据和

增量数据合并，这会导致连接操作性能的降低。OceanBase是典型的读写分离架构的分布式关系数据库，在超大表做连接的情况下，OceanBase的处理时间是在分钟级别的。对于一个实时系统来说，这显然是不能接受的。

不难发现，读写分离架构的数据库具有以下缺点：1. 查询处理效率低下，每次查询前都要先将增量数据和基线数据合并。2. 数据模式复杂，有些数据在内存中，有些数据在磁盘中，并且由于数据分布在不同的机器上，导致处理数据时产生了很多额外的开销。3. 网络负载远大于传统的数据库。

1.3 研究内容

由于读写分离架构本身的缺陷，该架构数据库连接操作的效率非常低下。以阿里巴巴开源的分布式数据库OceanBase为例，大量的性能测试表明：表的数据量达到10G 以上时（这个量级的表被称为大表），表连接耗时为分钟甚至小时级别。除了架构缺陷以外，OceanBase没有实现相应的连接优化算法也是其中一个很大的原因。对于连接操作，OceanBase 内部只有一种连接算法：排序归并连接。该算法的实现分为三个步骤：

1. 通过网络传输，分别将两张表的全部数据发送到一个节点上，在该节点的内存中缓存两张表的所有数据。根据OceanBase的数据存储策略，一张大表的数据是可能分布在多个节点上的，所以如果想要做排序归并连接，首先就要将各个节点的数据通过网络收发包机制集中到一个节点上，等所有的数据包都到达该节点后，将每个网络数据包的数据反序列化出来，存到内存的缓冲区中。

2. 数据全部集中到一个节点后，在该节点做排序归并连接。由于传统的排序归并连接算法只能在单点做，所以这里会有单点瓶颈：当一个节点的内存大小不足以存储两张表的所有数据时，将一部分数据先物化到磁盘中，需要时，再将数据从磁盘读取到内存中。

3. 返回连接结果。当内存中的排序归并算法完成后，做连接的节点会将连接的结果存到缓存区中，然后将缓存区的数据序列化成数据包，通过网络传输发送给客户端。

可以看出，排序归并连接算法虽然在实现上比较简单，在稳定性上表现很好，但是其效率非常低下，不能满足用户和应用的需求。首先，它的网络传输数据量太大，并且采用的是串行处理方法。众所周知，网络传输速度远慢于磁盘存取，更比不上内存操作，所以数据的网络间传输和延迟消耗了连接算法的大部分时间。其次，当一个节点的内存不足时，额外的磁盘存取操作也是很大一部分时间开销。

针对已观测到的问题，在国内外研究现状的基础上，本文的主要贡献有以下三点：

1. 结合读写分离架构的特点，提出了不先合并基线与增量数据，而是对二者分别处理的思路。该思路能够减少大量数据合并带来的开销，提高查询的效率。

2. 针对一张小表和一张超大表做内连接的情况，提出SemiJoin[18]优化算法。利用小表数据综合构造对大表的过滤条件。

3. 针对两张超大表做内连接且没有有效过滤条件的情形，提出分布式排序归并连接算法，利用并行计算提高两张超大表的内连接效率。

由于OceanBase 是目前比较著名的开源分布式数据库，它采用了读写分离式的架构，具有很好的代表意义，故本文选择在OceanBase 上实现两张连接优化算法，并通过大量的对比实验证明了两种算法大幅提高了连接效率。

1.4　本文结构

本文章节组织如下：

第二章，连接算法相关工作。主要介绍数据库连接操作的相关技术和概念，以及最近几年国内外对连接算法的一些研究与改进，主要集中在半连接和分布式连接。同时介绍下分布式数据库环境下影响连接效率的主要因素，以及基于不同视角的优化算法。

第三章，OceanBase简介与问题描述。主要介绍开源分布式数据库OceanBase的架构设计、数据存储、数据操作和数据更新等方面。同时分析OceanBase 架构的特点，包括它在事务和查询优化方面的缺点，以及读写分离架构带来的好处等。

第四章，优化的SemiJoin算法设计与实现。主要介绍针对读写分离架构，提出的一种改进的SemiJoin算法。并且详细介绍该算法的设计思路，算法在OceanBase源码上的具体实现等。同时，通过大量的对比实验表明改进算法大幅提高了连接的效率。

第五章，优化的分布式排序归并连接算法设计与实现。主要介绍针对读写分离架构，提出了一种改进的排序归并连接算法。并且详细介绍该算法的设计思路，算法在OceanBase 源码上的具体实现等。最后通过大量的对比实验，验证了改进算法的高效率。

第六章，对本文工作的总结和对未来工作的展望。

第二章　国内外连接算法相关工作

2.1　连接技术简介

传统的连接算法主要有嵌套循环连接、哈希连接和排序归并连接三种。这三种算法各具特色，随着数据库技术的发展，对这三种算法的优化也有很多。同时，半连接作为一种很高效的连接优化算法，一直以来也是学术界研究的热点。而随着分布式数据库的流行，分布式连接成为了一种新兴的连接算法。下面简要介绍一下这五种算法。

2.1.1　嵌套循环连接

嵌套循环连接（Nested Loop Join）是一种相对稳定、简单的表连接方法。它嵌套外层循环和内层循环，从而得到最终的连接结果集。该连接算法主要有三个步骤:

1. 查询优化器按照一定的规则决定表R和表S谁是驱动表、谁是被驱动表。驱动表做外层循环，被驱动表做内层循环。这里假设驱动表是R，被驱动表是S。

2. 以目标SQL中指定的谓词条件访问驱动表R，将得到的结果集记为驱动结果集R^1。

3. 遍历驱动结果集R^1，同时遍历被驱动表S。即先取出R^1 中的第1条记录，遍历被驱动表S，根据连接条件判断S中是否存在与该记录匹配的记录，之后再取出R^1 中的第2条记录，重复上述操作，直到取出R^1 中所有的记录为止。

这里外层循环是指遍历R^1所对应的循环，内层循环是指遍历被驱动表S所对应的循环。显然，外层循环所对应的R^1中有多少条记录，遍历被驱动表S的内层循环就要做多少次，这就是”嵌套循环”的含义。

在处理一些选择性强、约束性高，并且最终结果集较小的查询时，嵌套循环连接能够显示出较高的性能。嵌套循环连接由驱动表和被驱动表循环比较得到连接结果，当驱动表的记录较少，被驱动表连接列有唯一索引时，两张表记录比较的次数较少，所以嵌套循环连接的效率变得很高。当使用了嵌套循环后，数据库不需要等到全部循环结束再返回结果集，可不断地将查询的局部结果集返回，所以嵌套循环连接具有很快的响应时间。但是，当驱动表的记录很多，或者是被驱动表的连接列上没有索引时，两张表循环比较的时间变长，从而导致嵌套循环连接的效率变得十分低下。

可以看出，嵌套循环连接有以下几点特征：

1. 外部循环只执行一次，而内部循环一般会执行很多次。

2. 在全部数据处理完之前，就可返回结果集的第一条记录。

3. 可以有效利用索引结构来处理限制条件和连接条件。

4. 支持所有类型的连接。

传统的关系型数据库对嵌套循环连接做了大量优化。著名数据库Oracle在该连接算法上引入了向量I/O[19]。引入向量I/O后，Oracle可以将原先一批单块读所需耗费的物理I/O组合起来，之后用一个向量I/O去批量处理它们，实现了在不降低单块读数量的情况下减少所需要耗费的物理I/O数量，大幅提高了嵌套循环连接的执行效率。向量I/O的引入也反映在嵌套循环连接算法对应的物理计划上。对一个一次嵌套循环连接就可以处理完的查询语句，在引入向量I/O后的Oracle中，其执行计划有两个嵌套循环连接。

块预取[20]机制也能很好地提高嵌套循环连接的效率。当缓存没有命中时，基于单块处理（如row_id 访问、索引范围扫描等）的访问路径将导致一个单块的

物理读操作。这对嵌套循环连接来讲，特别是有很多条记录需要处理的时候，连接效率大幅降低。实际上，嵌套循环连接算法也会使用多个单块物理读操作来访问多个相邻的块。块预取功能是解决上述问题的一个方法。其优化技巧是：对多个相邻的块只使用一次多块物理读取，来代替之前的多次单块物理读取。块预取功能对于表和索引都是适用的。

随着内存数据库的发展，嵌套循环连接算法也得到了很多的改进与优化。在共享内存架构中，对该连接算法的优化主要集中在两个方向：降低缓存丢失率和提高SIMD[21]利用率。在对SIMD的研究中，Zhou J[22] 提出了三种优化方式：复制外层循环方式、复制内层循环方式以及旋转方式。在对缓存优化的研究中，Shatdal[23] 提出了基于块的处理方式，极大地降低了高速缓存缺失率。在无共享架构下，一般采用复制分片技术，将一张表的数据复制到所有节点上，Hadoop 框架中采用分布式缓存[24]，Spark 框架中则采用广播变量[25]。

2.1.2　哈希连接

哈希连接是大数据集连接常用连接方法之一。查询优化器首先选取两张表中相对较小的表，利用查询的连接属性在内存中建立该表的散列表，然后依次扫描大表并针对大表的每一行记录探测散列表，找出与散列表匹配的行。当较小的表可以完全放于内存中时，哈希连接的总成本就是对两张表访问一次的成本，这种情况下哈希连接的性能最优。当较小的表也不能完全放入内存时，查询优化器会将其切割成多个不同的分区，内存中放不下的分区被写入磁盘的临时段，这种情况要求系统有较大的临时段以提高I/O 的性能。Oracle 数据库中的哈希连接由以下阶段组成。

1. 首先选择出一个“小表”。这里的小表是指参与连接的表中数据量相对较小的表。其次遍历该表在连接列上的全部取值，对每一个取值调用哈希函数进行处理。由哈希函数的特点可知：相同的取值一定会被分到相同的哈希桶内，不同的取值也可能被分到相同的哈希桶内。

2. 将经过哈希函数处理过的小表连接列和数据一起存放到Oracle 的PGA空间中。PGA中有一块专门存放此类数据的空间：hash_area。然后，根据哈希函数的结果划分哈希桶，使得所有在某个哈希值上相等的小表数据被分到相同的桶内。最后，根据桶信息建立哈希键值对应位图。

3. 依次读取大表数据连接列，对该列的每一个数组，执行相同的哈希函数，并根据函数生成的哈希值，将该行记录对应到相应的桶上（应用哈希检索算法）；

4. 在每个哈希桶中进行小规模的精确匹配。因为哈希桶相对较小，所以匹配的成功率变高。同时，匹配操作是完全在内存中进行的，速度比其他连接算法要快很多；

哈希连接能够很好地应用于没有索引的大表和并行查询的环境中，并提供极高的性能，被称为是连接的重型升降机。该算法很适合于一张小表和一张大表之间的连接，特别是在小表的连接属性有非常好的可选择性情况下，此时哈希连接的执行时间可以近似看作与全表扫描大表所耗费的时间相当。由于哈希的特点，哈希连接只能应用于等值连接。并且因为要建立哈希表，需要大量内存，故第一次的结果返回较慢。

在多核并且数据全部存放在内存中的环境下，哈希连接具有很大的优化空间。Monetdb[26] 中提出的radix 连接算法就是充分利用了先进硬件技术的特征，极大地减少了数据的Cache 缺失，提高了连接操作对内存的利用率。但是，Blanas S在论文[27] 中提到，与小表数据能够全部存放在内存中的哈希连接相比，radix 算法的效率并没有显著的提高，因为在没有分区阶段时，该算法能充分利用硬件的预取而提高一定的效率。然而，Balkesen C[28]通过论文实验证明：即使是无分区阶段的连接算法，性能上还是比不上radix 连接算法。该论文指出Blanas S在其论文中将小表数据全部存放在内存中，故避免了对哈希表的分区，导致分区操作的开销在连接算法中的比例降低。这导致了其对radix 连接性能的低估。

2.1.3　排序归并连接

排序归并连接旨在处理两张表的连接操作，通过对连接列分别排序后，再归并连接得到最后结果集的方法。归并连接的具体操作是依次读取两张表的一条记录进行对比。如果两个记录符合连接条件，则输出连接后的行并继续读取下一行。如果不符合，则舍弃两个记录中较小的记录并继续读取，一直到某一个表的数据扫描结束，则执行完毕。所以该算法执行只会对每张表扫描一次，并且有时不需要扫描完整张表就可以停止。排序归并连接有两个前提条件：1. 必须预先将两张表在连接属性上进行排序；2. 两张表的连接条件中必须存在等值连接。

排序归并连接算法总的消耗是和表中的记录数成正比的，而且与嵌套循环连接不同，该算法对表最多读取一次。因此，排序归并连接对于两张大表连接是一个比较好的选择。对于排序归并连接可以从以下两点提高性能：

1. 连接列的类型。如果两张表的连接列都为唯一列（即不存在重复值），或者只有一张表在连接列上是唯一列也可以，这种情况下连接性能是最好的。这种方式为一对多关联方式，也是我们最常用的主从表关联查询；如果两张表的连接列存在重复值，则在两表进行连接的时候还需要借助第三张表来缓存重复的值，这里的第三张表叫做“worktable”，通常被存放在内存中，这会对性能有所影响。鉴于此，对于排序归并连接常用的优化方式有：连接列尽量采用聚集索引[29]（唯一性）。

2. 排序归并连接算法的前提是：两张表都经过排序。在使用该算法时，最好优先使用排序后的表。如果表没有排序，则尽量选择索引覆盖列作为连接列。因为对大表排序是一个很耗资源的过程，选择索引覆盖列进行排序的性能要远远好于对普通列排序的性能。

在对排序归并算法的研究中，Kim C[30]指出，排序归并连接的效率与SIMD宽度相关，在一定宽度的SIMD环境下，该连接够达到超过哈希连接的性能。而Albutiu M C[31]认为NUMA架构也能极大的提高排序归并连接的效率，

他在论文中提出了MPSM 并行排序归并算法，该算法能够超过哈希连接算法的性能，即使在没有SIMD 的支持下。

2.1.4 半连接

半连接（SemiJoin）是一种对内连接的优化算法，最早在AHY[32] 算法中被提出。该算法针对一张小表和一张超大表的内连接，旨在利用小表的数据特征过滤大表的数据，将连接操作中传输整张大表数据变成只传输大表的一部分数据，减少了传输代价，提高了查询的效率。

对半连接的优化主要集中在动态优化和静态优化两个方面：

1. 动态优化。文献[33]中介绍了目前应用比较广泛的SDD-1算法，该算法在Hill-climbing[34]算法的基础上发展而来，算法首先遍历全部半连接操作，计算每个连接节省和耗费的开销，然后采用动态穷举方式，直到确定了总开销最低的连接顺序，达到大幅减少查询响应时间的目的。但是，当查询图比较复杂时，该算法会使得查询的时间开销呈指数增长。其他的一些动态优化算法，如上述算法的变形算法R 算法[35]、R^* [36] 算法、基于查询图的贪婪算法LF 算法[37]，以及A 算法[38]、A^* 算法[39]等，也存在着相同的问题。即在查询图简单的情况下，算法效率较高，当查询图比较复杂时，算法性能急剧下降。

2. 静态优化。静态优化算法虽然优化效果没有动态算法好，但其最大的优点是执行效率高。1-PSJ算法[40]是其中的典型代表，该算法首先扫描每张表一次，得到所有的连接属性，然后对每个表找到其有最大收益的半连接操作集。最后将所有的连接属性并行传输到相应的节点上，每个节点并行地对表进行简化。此外，很多算法把优化重点放到半连接收益和连接顺序上，例如W 算法[41]和PERF 算法[42]等。这些算法首先将全部表按照数据量从大到小排序，接着计算每个半连接的节省或开销，对于节省的半连接，先执行该半连接，执行完后将其加到执行方案里。计算完开销后评估并执行最终的执

行方案。这种实现方式避免了动态优化算法中通过循环方式查找节省开销的半连接。无论表的个数多少，该算法只需要很少的扫描次数，减少了算法的复杂度。

在国内，对半连接的优化是学术界很感兴趣的一个方向。文献[43]中提出了将两次半连接进行对接的思路，该算法的优化思路集中在并行处理上，算法前提是查询的数据分布在多节点上，算法首先利用最小生成树算法生成多个连接对，然后利用多节点的特征并行地执行查询计划，达到减少查询响应时间的目的。文献[44] 针对多表连接查询，提出了改进后的半连接算法。该算法首先将查询中所有表按照半连接顺序构造成类树性结构，然后从该结构的叶节点向根节点遍历，对每个节点执行半连接操作以减少该节点表的数据量，当遍历结束时，根节点处表的数据量会被极大的减少，最后再从根节点开始，利用PERF 位向量依次减少其他节点处表的数据量。这样，在最终执行连接操作时，每张表都是其所能达到的最小状态，从而提高了多表查询的性能。

2.1.5　分布式连接

在NoSQL领域，Shark[45]数据库对连接性能做了很大的优化，Shark基于Spark内核，包含map连接和shuffle连接两种分布式连接算法。map 连接首先读取小表的全部数据，然后将其广播传输到所有节点上，每个节点并行地对小表的全部数据和大表的部分数据做连接，最终整合每个节点的连接结果。shuffle 连接首先对两个表在连接属性上进行哈希，把哈希值相同的记录传输到相同的节点上，然后在各个节点上并行地做连接，最后整合每个节点的连接结果。

对于读写分离架构的数据库来说，优化连接性能显得更加复杂。以HBase为例，HBase不能支持where 条件，Order by查询，只能按照主键和主键的range来查询。但是可以通过将Hive和HBase结合，使用Hive提供的HQL语言实现HBase大表连接查询，同时，使用底层的MapReduce计算框架处理连接查询任务，将满足条件的结果存放在HBase表中。

虽然国内外有很多对连接算法的优化，但是大部分优化算法都没有考虑到读写分离架构的特点。本文提出的分布式排序归并连接算法本质上是shuffle连接，但是在连接之前没有把所有的基线数据和增量数据都合并，而是对基线数据和增量数据并行处理。减少了数据的交互，同时也保证了数据的正确性。

2.2 本章小结

本章主要介绍了嵌套循环连接、哈希连接和排序归并连接在国内外的研究现状，可以看出，这三种算法的优化大多都是在单点环境下，大多数研究都集中在内存使用、CPU利用率以及一些硬件特征上，很少有针对分布式数据库的。而分布式计算作为处理和存储海量数据的必然选择，也逐渐被应用到连接算法上来了。对半连接的优化算法虽然很多，但是基本思路都是一致的，只不过在具体的实现上存在差别。但是，可以看出，很少有针对读写分离架构下大表连接算法的优化，所以，这是本文的一个创新点，也是本文的贡献之一。

第三章　OceanBase简介与问题描述

3.1　背景介绍

互联网企业的很多应用都有这样一个特点：数据量很大，但是一段时间内的修改增量相对很小。这种情况下，读写分离架构的数据库是很好的选择。这类数据库的设计思路很早以前就被提出了，例如谷歌的BigTable[46]。对于BigTable来说，每一个节点有一个memtable，当该节点接收到一个写请求，它会将数据写到内存的memtable中。当memtable大到一定规模会被冻结，冻结的memtable会转换成SSTable [47] 形式写入GFS。当该节点接收到一个读请求，这个读操作会查看所有SSTable文件和memtable的合并视图。

在国内对数据库的研究中，OceanBase是最典型的读写分离的数据库。从2010年起，阿里巴巴开始自主研发数据库系统OceanBase，截至2015年，OceanBase已经可以支撑淘宝、天猫和聚划算的所有日常交易。经过五年的开发，OceanBase经历了多个版本的更新，系统架构更加的成熟。在通过了收藏夹、支付宝和“双十一”活动等应用的考验后，OceanBase 已经成为了国内目前最著名、最成熟、真正在线上支持海量数据的分布式数据库。

在阿里巴巴之前，中国有大批的专家学者和企业曾致力于数据库系统的研究，但是几十年来没有任何一个商业化的数据库产品能够替代传统的高端数据库。最终反而是阿里巴巴在商业需求的驱动下实现了这一目标。与传统的数据库产品不同，OceanBase的升级维护十分简单，它不需要昂贵的共享存储，也不需要高可靠的服务器，由于OceanBase是开源的数据库，故用户也不需要购买软件的许可费，

这些因素导致用户使用OceanBase的成本远低于商业数据库。同时，OceanBase采用分布式的系统架构，该架构的高可靠性和容错特性可以保证在服务器、存储端或者网络通信出现异常的情况下，用户的业务几乎不受影响。

3.2 OceanBase架构介绍

作为一个分布式关系型数据库，OceanBase最大的架构特点就是把基线数据分布式地存储在多个节点上，把增量数据集中式地存储在一个节点上，实现了读写分离，从而得到了集群的高可扩展性和数据的高一致性，避免了分布式事务。

一个OceanBase集群由四种角色组成：

1. RootServer: 集群的管理者，管理集群中所有节点的上下线以及UpdateServer的选主。同时也存储系统所有表的Schema信息、所有Tablet 的分布信息、各个节点的地址、所有节点的状态信息等。

2. UpdateServer: 处理系统所有的事务，存储系统所有的增量数据。由于一个集群中只有一台主UpdateServer，所以集群中的所有事务都集中在一个节点上进行，避免了分布式事务，同时也保证了事务的ACID特性。

3. MergeServer: 集群与用户进行交互的接口。负责接收客户端发送的SQL语句，并对该语句进行词法分析、语法分析、生成逻辑计划、生成物理计划、执行物理计划等操作。最后将处理的结果返回给客户端。

4. ChunkServer: 分布式的存储集群所有的基线数据，并且可以十分方便的动态扩展。数据分为多个副本存储在ChunkServer上，即使有一台机器发送故障，数据也不会丢失。

这四种角色以进程的形式运行在服务器端。其中，RootServer进程与UpdateServer进程一般放在一台服务器上，MergeServer进程与ChunkServer进程一般放在一台服务器上。

OceanBase集群大都由普通的、成本比较低的PC服务器搭建而成。因此对于一个企业来说，搭建一个OceanBase集群在成本上是有优势的。尤其是对于中小型企业，它们无法支持DB2的大型机、小型机高昂的成本以及维护费用，但是可以选择一个由几十个廉价PC服务器组成的OceanBase集群，在节省了成本的同时，也满足了它们对庞大数据的处理需求。由于上面的介绍可知，一个OceanBase 集群最少有PC 四台服务器，如图3.1所示：

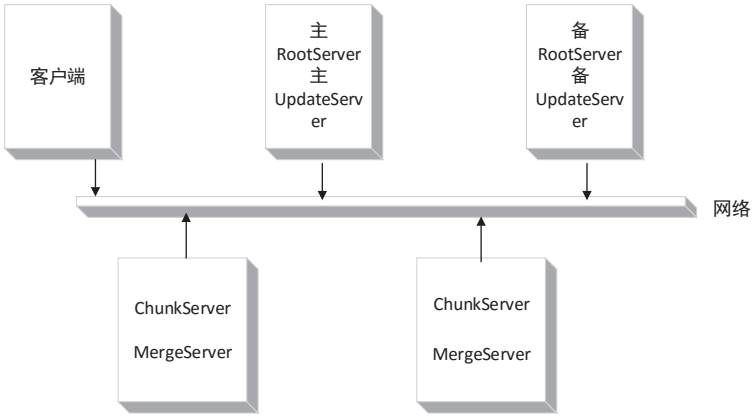

图 3.1: OceanBase集群架构

由图可知，一个OceanBase集群是由一个主RootServer、一个备RootServer、一个主UpdateServer、一个备UpdateServer、多个MergeServer和多个ChunkServer组成的。下面详细介绍下OceanBase的集群运行机制。

3.2.1　主备机制

由于OceanBase的架构设计，一个OceanBase集群中只有一台RootServer和一台UpdateServer。这是一个明显的单点瓶颈，当RootServer和UpdateServer发生了宕机之后，整个集群就会处于不可用的状态，同时一些存储在机器内存中的数据也会丢失。为了可用性地解决问题，OceanBase 采用了主备机制，分别为RootServer和UpdateServer准备了一台备机。备机与主机是实时同步的，当主机宕机后，备机能够快速地替换原主机，成为新主机。

下面分别介绍下RootServer和UpdateServer的主备机制。

1. 每个集群中都有一主一备两台RootServer，分别分布在不同的服务器端。主备之间保持数据强同步，所有的操作都要先同步到备机，然后才能修改主机，返回修改操作成功。例如集群中ChunkServer或者MergeServer上下线等操作会引起RootServer主机的内部数据发生变化，这些变化将以操作日志的形式同步到RootServer备机。而备RootServer会实时回放这些同步过来的操作日志，从而保持了与主RootServer的数据同步。OceanBase的主备RootServer可以实现自动切换，这是通过Linux HA软件实现的：RootServer对外部只提供一个访问接口：Virtual IP。ChunkServer、MergeServer等都是通过Virtual IP 向RootServer进行网络通信的。当集群处于正常状态时，Virtual IP指向的主RootServer，当主RootServer发生故障时，部署在主备RootServer上面的HA软件能够检测到主机发生了故障，并将Virtual IP漂移到备机。备RootServer的后台线程检测到了Virtual IP漂移到自身，自动地把自己切换为主机状态并对外提供服务。

2. 一个OceanBase集群中只能有一台主UpdateServer，但是可以有一台或多台备UpdateServer。正常状态下，只有主UpdateServer对外提供服务。当主UpdateServer发生故障时，系统自动地从与主机一致的备机中选取一台当做主UpdateServer。UpdateServer 的主备机之间是通过操作日志来保持数据的强一致性。主UpdateServer在执行事务时，首先将事务日志同步到备机，日志同步成功后主机才能认为事务已完成。而备UpdateServer 通过日志回放线程，不停地在本机回放主机同步过来的日志，保持与主机的实时同步。

3.2.2 ChunkServer可扩展性与数据正确性

OceanBase最大的特点之一就是可以实时地向集群中增加ChunkServer，用来缓解数据的增长带来的存储压力。增加ChunkServer的操作很简单，不需要停掉正在运行的集群，直接在新增服务器上起一个ChunkServer进程，集群中

的RootServer会检测到该ChunkServer的上线信息，自动地执行负载均衡策略，把其他ChunkServer的数据分发到新增的ChunkServer上。

对于一个分布式系统来说，TCP协议[48]传输、磁盘读写，程序Bug等因素都会导致数据错误甚至损毁。OceanBase为了保持数据的强一致性，采取了以下数据校验措施：

1. 数据存储校验。每一个存储记录，都会保存64位CRC校验码。当数据被访问时，重新计算和比较校验码。

2. 数据传输校验。每个传输记录会有64位CRC[49]校验码。当数据被接收时，重新计算和比较校验码。

3. 数据副本校验。ChunkServer在生成新的子表时，会为每个子表生成一个校验码，并将该校验码汇报给RootServer，RootServer会核对同一张子表的不同副本的校验码是否一致。

3.2.3 事务执行流程

OceanBase的所有事务都在一台主UpdateServer上执行，避免了分布式事务，同时也支持事务的ACID[50]特性。对于一个写事务，OceanBase的处理流程如图3.2。

可以看出，写事务分为以下步骤：

1. MergeServer解析用户的SQL请求，生成物理计划。

2. MergeServer向ChunkServer请求事务所需要的基线数据，并将基线数据和物理计划一起传给UpdateServer。

3. UpdateServer根据物理计划执行写事务。包括修改内存表、存入增量数据、同步事务日志到备机，提交事务等。

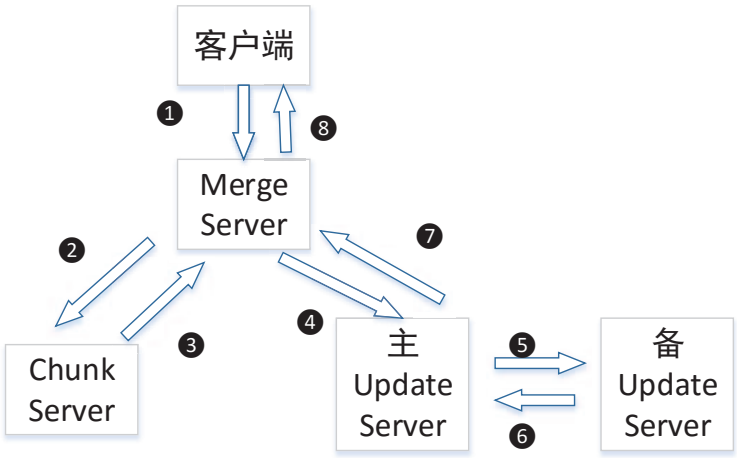

图 3.2: OceanBase写事务流程

4. UpdateServer向MergeServer返回操作成功或者失败。MergeServer 把事务执行结果返回给客户端。

3.2.4　查询执行流程

由于OceanBase采用了读写分离的架构，它的查询操作与传统的数据库相比更为复杂。不仅涉及到多个节点之间的网络通信，还必须执行基线与增量数据的合并，才能得到正确的结果。对于一个简单的查询操作，OceanBase的处理流程如图3.3。

可以看出，查询分为以下步骤：

1. MergeServer解析用户的SQL请求，生成物理计划。
2. MergeServer将根据数据的分布信息，将数据查询请求发送给多台ChunkServer。
3. ChunkServer向UpdateServer请求增量数据，并将得到的增量数据与本身的基线数据进行合并，将合并后的数据返回给MergeServer。
4. MergeServer整合所有ChunkServer发送的合并后的数据，对数据进行处理后将其返回给客户端。

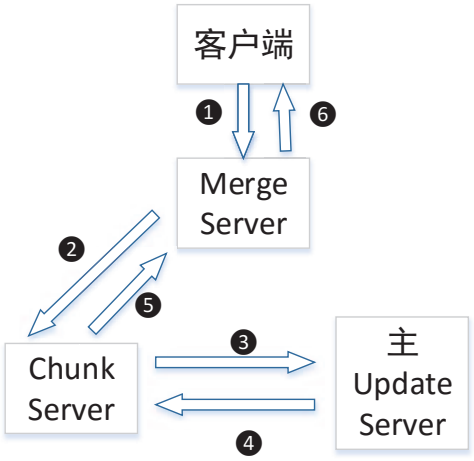

图 3.3: OceanBase查询流程

3.2.5　定期合并

UpdateServer作为OceanBase集群中唯一存放增量数据的节点，必然会有单点瓶颈，即增量数据是不断地增加的，而UpdateServer的内存不能动态扩展，所以经常会有UpdateServer内存不足以存储所有增量数据的情况出现。OceanBase采用定期合并的方案解决这个问题。

定期合并本质上是一种大规模的基线与增量数据合并操作，目的是为了减少增量数据的规模，并生成新的基线数据。具体分为三个步骤：

1. UpdateServer冻结当前的活跃内存表（存储增量数据的结构），生成冻结内存表，并开启一个新的活跃内存表，后续的更新操作都会写入到新的活跃内存表中。

2. UpdateServer通知RootServer数据的版本发生了变化，RootServer 通过心跳消息通知ChunkServer。

3. 每台ChunkServer启动定期合并操作，向UpdateServer发送拉取每个子表对应的增量数据的请求。

4. ChunkServer将本地SSTable中的基线数据与UpdateServer发送来的增量数据执

行一次多路归并，融合后生成新的基线数据并将之存放到新的SSTable中。

定期合并后，UpdateServer中只会有一张新建的增量数据为空的活跃内存表，之前的增量数据直接被丢弃。ChunkServer中将会保存新生成的基线数据，之前的基线数据直接被丢弃。

由于定期合并需要调用大量资源，对系统的服务能力影响很大，生产上往往将其安排在每天服务低峰期执行（例如凌晨1点）。

3.3 OceanBase优缺点分析

作为一个已经支持了大量线上应用的成熟分布式数据库，OceanBase具有很多优点：

1. 成本低。性价比高是OceanBase在企业中流行的一个根本原因。与价格昂贵的大型机，小型机相比，OceanBase只由几十台廉价的PC服务器组成，却能支持海量数据的存储与管理。

2. 高可扩展性。相比传统的关系数据库，OceanBase的最大亮点就是其可自动扩展的特点，它不仅仅可以扩展到一个数据中心，甚至同城，在未来，OceanBase可能成为跨地域多数据中心的全球数据库。对于大型应用来说，后台系统中只有数据库容量最难提升。传统数据库的运维人员需要花费大量的时间与精力来做数据库扩容，包括读写分离、垂直拆分、水平拆分等。不过，对于OceanBase运维人员来说，扩容是一件非常简单的事情，唯一要做的事情就是加入更多的服务器。加入新的服务器，系统容量就提升了，OceanBase 系统内部可以实现自动扩容，这解决了工业界的燃眉之急。

3. 数据强一致性。OceanBase的主备机制、数据副本机制和数据校验与传输校验机制等保证了其数据的强一致性。一个OceanBase集群能够自动容忍多台服务器甚至是整个数据中心故障，而不会丢失一条记录。这对于支付宝，淘宝等金融类应用来说，是十分必要的。

4. 负载均衡。分布式系统中存在一个著名的“短板理论”，即一个集群如果出现了负载不均衡，那么负载最大的机器将成为影响系统整体表现的瓶颈和短板。为了解决这种问题，OceanBase内部自动把数据切分为一个个小的分片，每台机器只服务若干个分片，当某台服务器的分片成为热点时，系统自动触发迁移操作，将该分片从负载较高的服务器迁移出去，这样就避免了“木桶效应”。所以，无论应用的热点怎么变，OceanBase都可以把热点数据均衡到整个集群，不会因为一两台服务器把整个集群压垮。

没有任何系统是完美的，OceanBase也存在着很大的缺点，有待进一步的完善。OceanBase 的主要缺点有以下几点：

1. UpdateServer单点瓶颈。UpdateServer的内存容量决定了集群最大能够处理的事务数。虽然OceanBase对简单事务的处理速度很快，但是当事务数达到UpdateServer的瓶颈时，集群处于不可用状态，需要手动执行定时合并操作，增加UpdateServer空闲内存容量。

2. 复杂事务处理慢。OceanBase在短事务的处理上具有很大的吞吐量，但是当处理复杂事务时，大量的冲突导致集群的TPS出现明显的下降，而目前的OceanBase版本对这一问题还没有很好的解决方案。

3. 没有查询优化机制。对于大表查询，OceanBase只有在使用第一主键查询时能够得到很短的响应时间。当使用全表扫描时，大量的网络传输时间和基线与增量数据合并的时间会导致查询的效率极低。此外，由于OceanBase 没有二级索引，连接优化等查询优化模块，导致了其在数据查询方面出现了明显的短板。

可以看出，虽然OceanBase具有高可扩展性、数据高可靠性以及高事务吞吐量，但是由于其架构本身的缺陷，OceanBase在查询优化方面的表现不太尽如人意。首先，OceanBase没有一套基本的查询优化机制，每个查询语句的物理执行计划都是固定的。其次，OceanBase没有连接优化算法，所有的连接操作都是采用排

序归并连接算法，该算法虽然稳定性较高，但是在处理超大表连接上的表现极差。这些因素导致了OceanBase在查询优化，尤其是大表连接上的处理效率不能满足用户的需求，这也是OceanBase目前最大的短板。

3.4 问题分析与定义

本文主要研究的是如何提高读写分离架构的数据库在大表连接方面的效率。OceanBase作为该架构的代表，大量的实验证明，OceanBase 在大表连接上的响应时间为分钟甚至小时级别，远远不能达到用户的需求。针对这一现象，本文提出了两种大表连接优化算法：SemiJoin 和分布式排序归并连接，并在OceanBase 源码上实现了这两种算法。为了更好地描述这两种算法，本文对OceanBase 进行一些形式化的定义。

对于SemiJoin做以下定义：

1. OceanBase集群。现有一个集群，集群有n台机器。节点M代表MergeServer，节点U代表UpdateServer，节点Ci代表ChunkServer。其中，节点U集中地存储了所有的增量数据，节点Ci分布式地存储基线数据。

2. 数据分布。现有R表和S表在属性R.A=S.B上做自然连接。R表为小表，S 表为大表。R 表的增量数据全部存在节点U上。R表的基线数据共有1个tablet：R1，存在节点Ci 中。S 表的增量数据全部存在节点U 上。S表的基线数据共有m个tablet：S1，S2……Sm，分别存在基线数据节点Ci上。

3. 连接语句。SemiJoin语句：select * from R inner join S on R.A=S.B；

4. 连接结果。将满足条件的R表和S表的数据返回给节点M。节点M 再将结果返回给客户端。

对于分布式排序归并连接做以下定义：

1. OceanBase集群。现有一个集群，集群有n台机器。节点M代表MergeServer，节点U代表UpdateServer，节点Ci代表ChunkServer。其中，节点U集中地存储所有的增量数据，节点Ci分布式地存储基线数据。

2. 数据分布。现有R表和S表在属性R.A=S.B上做自然连接。R表的增量数据全部存在节点U 上。R表的基线数据共有m个tablet：R1，R2 ……Rm，分别存在节点Ci上面。S表的增量数据全部存在节点U 上。S表的基线数据共有m个tablet：S1，S2……Sm，也分别存在节点Ci上。

3. 连接语句。现有R表和S表在连接属性R.A=S.B上做分布式连接：select * from R inner join S on R.A=S.B；

4. 连接结果。将满足条件的R表和S表的数据返回给节点M。节点M 再将结果返回给客户端。

3.5　本章小结

本章首先介绍了OceanBase出现的背景以及动机。其次详细介绍了OceanBase的架构设计与集群整体框架。接着分析了OceanBase的一些优缺点，强调了其高可扩展性和数据强一致性，并分析OceanBase能够支撑大量在线业务的原因。最后结合OceanBase的集群架构与本文的研究工作，对两种连接优化算法进行了形式化定义，方便在后续章节中更加清晰地介绍这两种优化算法。

第四章　SemiJoin算法优化

SemiJoin算法，又叫半连接算法，是对两表做内连接的一种优化方法，特别是针对分布式关系数据库中一张大表和一张小表的内连接。该算法原理是利用小表在连接列上的取值，实现对大表的过滤。目的是减少网络通信量，从而缩短查询响应时间。

4.1　优化动机

OceanBase是一个读写分离的分布式数据库，当表的记录数很大时，OceanBase会将一张表的数据分为多个tablet，每个tablet大小为256M，多个tablet 会按照负载均衡的策略分布在不同的节点上，不同节点间通过网络收发包进行通信。但是，OceanBase 处理表连接操作的节点只有一个，所以网络通信是一对多的关系：两张表的数据分布在多个节点上，但是最终都要在一个节点上做连接。

在具体的实现中，OceanBase首先将两张表的数据通过网络传输全部都传到一个节点上，然后在该节点的内存中对两张表的数据做排序归并连接。如果该节点的内存不足以容下所有数据，则首先将一部分数据物化到磁盘上，需要时再把数据从磁盘中读到内存中继续做排序归并连接。

显然，上述处理方案的效率是十分低下的。首先，网络传输的数据量过多。网络通信是比磁盘读取还要慢的操作，当网络传输的数据量过多时，所花费的时间也显著增多。上述方案几乎在网络上传输了两张表的全部数据，如果碰到了传输超过10G以上的超大表时，网络传输消耗的时间是用户所不能接受的。其次，连接存在单点瓶颈。连接操作只在一个节点上进行，考虑到该节点的内存不足以

存放所有数据的情况，其中的磁盘存取操作又会消耗很多的时间。

针对以上问题，本文提出了一种对内连接的优化算法：SemiJoin。该算法充分利用OceanBase的架构特点，尽量减少网络上传输的数据量，解决了连接操作的单点瓶颈，大幅地提高了内连接效率。

4.2 算法流程与优化效果理论分析

SemiJoin 算法主要是通过并行过滤的思想减少网络传输量。由第三章的形式化定义可知，关系R和关系S在属性R.A=S.B上做连接操作，使用SemiJoin方法表示该连接操作：

$$R\infty_{A=B}S = (R \ltimes_{A=B} S)\infty_{A=B}S \tag{4.1}$$

或：

$$S\infty_{A=B}R = (S \ltimes_{A=B} R)\infty_{A=B}R \tag{4.2}$$

其中SemiJoin运算$R \ltimes_{A=B} S$定义为：

$$R \ltimes_{A=B} S = \Pi_R(R\infty_{A=B}S) = R\infty_{A=B}(\Pi_B(S)) \tag{4.3}$$

SemiJoin运算$S \ltimes_{A=B} R$定义为：

$$S \ltimes_{A=B} R = \Pi_S(S\infty_{A=B}R) = S\infty_{A=B}(\Pi_A(R)) \tag{4.4}$$

由上述公式可以得出：

$$R\infty_{A=B}S = (R\infty_{A=B}(\Pi_B(S))\infty_{A=B}S \tag{4.5}$$

或

$$S\infty_{A=B}R = (S\infty_{A=B}(\Pi_A(R))\infty_{A=B}R \tag{4.6}$$

其中，∞表示连接，⋉表示SemiJoin，Π表示投影操作。下面通过图4.1来具体分析下该优化算法的流程。

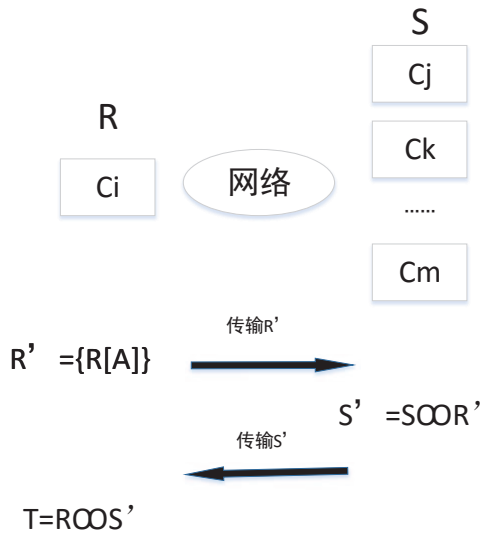

图 4.1: SemiJoin算法流程图

其中，R'表示R表在属性A上的不重复值集合，S'表示S表被集合R'过滤后的数据。T表示最终的连接结果。由图可知，SemiJoin算法流程主要分为四个步骤：

1. 小表数据扫描：首先，构造对小表的数据扫描操作符。通过该操作符将小表的数据从不同节点上传输到做连接的节点上，并存到内存中。由于小表的数据量较少，所以该操作不会花费很多的时间。当数据都存到内存中之后，在内存中对小表的所有数据进行去重操作，具体操作如下：首先根据连接列对小表的数据进行快速排序，其次遍历排序后的数据，将连接列上的值存到一个集合里面，如果碰到重复的值，则只取一个存到集合中。去重操作完成后，我们最终会得到一个连接列的值集合：R'。

2. 构造对大表的过滤操作符：通过第一个步骤得到的集合R'，可以构造对大表的过滤操作符。过滤条件一般都是用表达式的形式构造的。最简单的处理就是构造一个in表达式，将集合中的值全部作为in表达式的元素。

3. 大表数据读取：当过滤操作符构造好之后，就可以通过网络传输将该操作符发送到所有存有大表数据的节点上，每个节点收到过滤操作符之后，把大表的数据从磁盘中读取出来，在内存中对大表的每一条记录进行判断：如果该记录的连接列上的值满足表达式的条件，则将该记录存到缓冲区里，如果不满足，则继续判断下一条记录。缓存区满了之后，把里面的数据封装成数据包，通过网络传输发送给做连接的节点。

4. 排序归并连接：当做连接的节点收到其他节点发送的过滤后的大表数据，对这些大表的数据进行合并和排序。排完序之后再和之前存在内存中的小表的数据做排序归并连接。并将连接的结果返回给客户端。

由以上四个步骤可以计算OceanBase做一次SemiJoin需要的开销：

首先是小表R计算在属性A上的不重复值集合，并把该集合发送到S表所有的节点上：

$$T_1 = (I_0 + I_1 * SIZE(A) * N(R[A])) * N_S \tag{4.7}$$

其中I_0为两个节点之间启动一次网络传输的固定耗时，I_1为网络内的单位传输耗时，$SIZE(A)$为A属性的长度，$N(R[A])$为R表在属性A上的不重复值个数，N_S 为S 表数据存储所使用的节点个数。

其次是S表把过滤后的数据传输到R表所在的节点上：

$$T_2 = \sum_{k=j}^{m}(I_0 + I_1 * SIZE(S) * card_k(S^{'})) \tag{4.8}$$

其中$SIZE(S)$为S表中一条记录的长度，$card_k(S^{'})$为S表在节点C_k上经过滤后的记录数。

由上述公式可知，一次SemiJoin的总开销：

$$T = T_1 + T_2 = \sum_{k=j}^{m}(2I_0 + I_1 * SIZE(A) * N(R[A]) + I_1 * SIZE(S) * card_k(S^{'})) \tag{4.9}$$

可以看出，除了$N(R[A])$和$card_k(S')$为变量，其他的都是常量。而由于R是小表，故$N(R[A])$的值较小，因此总开销和$card_k(S')$是正相关的。

当$\sum_{k=j}^{m}(card_k(S')) \ll \sum_{k=j}^{m}(card_k(S))$ 时，算法可以获得很好的优化性能。举个简单的例子：当大表的数据为10G，小表的数据为100M 的时候，通过SemiJoin过滤后的大表数据可能为200M 左右，故最后变成了两张分别为100M 和200M 的小表在一个节点的内存中做排序归并连接，可以明显看出这种操作是很高效的。

4.3 算法在OceanBase源码上的实现

SemiJoin的设计框架分为三个部分，第一部分为hint的解析；第二部分为对小表的处理；第三部分为对大表的过滤。代码实现的流程则按照传统数据库对sql语句的处理流程：首先把sql 语句解析成语法树，然后根据语法树生成逻辑计划和物理计划，最后执行该物理计划树的open 函数，通过不停的调用物理计划树的get_next_row函数将两张表的连接结果返回给客户端。

4.3.1 hint解析子模块设计

由于OceanBase现在还没有一套完整的查询优化框架，所以暂时不能自动地判断是否使用SemiJoin，需要用户在sql语句中使用hint来显式指定某两张表做SemiJoin。hint的结构设计：*+SEMI_JOIN(parameter1，parameter2，parameter3，parameter4)*

其中parameter1为小表的表名，parameter2为大表的表名，parameter3 为小表的连接列的列名，parameter4 为大表的连接列的列名。

hint模块的代码实现如下：

1. 新增词法节点和语法节点，生成语法树。

2. 解析语法树。在OceanBase源码中新增数据结构ObSemiTableList用来存

储hint中的所有信息，并把ObSemiTableList存到逻辑计划树里。

3. 解析逻辑计划。根据ObSemiTableList中存储的hint信息判断用户是否正确地使用了hint。如果是，则生成新的SemiJoin 物理计划，否则，生成OceanBase原有的内连接物理计划。

4.3.2 小表处理子模块设计

SemiJoin中对小表的处理较为简单，主要有三个步骤：

1. 新增物理操作符ObSemiLeftJoin，把小表的所有数据通过网络传输读到MergeServer的内存中。

2. 调用do_sort函数：对小表的数据按照连接列进行快速排序。

3. 调用do_distinct函数：对小表在连接属性上的数据去重，并将去重后的结果缓存到集合里。

4.3.3 大表过滤子模块设计

SemiJoin对大表的处理主要集中在如何对大表的数据进行过滤。主要步骤如下：

1. 修改物理操作符ObMergeJoin，调用函数get_left_semi_join_result获取小表在连接属性上的集合。

2. 根据集合构造对大表的过滤条件。

3. 将构造好的过滤条件分发到存储大表的所有节点上，每个节点根据过滤条件读取大表的数据，并将过滤后的数据发送到MergeServer。

显然，过滤条件的选取十分重要。过滤条件直接关系到过滤的效果，进而影响查询的整体响应时间。本文在SemiJoin算法的实现过程中先后对过滤条件设计

了三个版本，每个版本都在之前的基础上进行了改进，以获得更好的过滤效果。下面具体介绍下三个版本的实现。

1. in表达式版本。将小表集合中的值作为in表达式的元素，对大表的每一条记录判断连接列上的值是否在in 表达式里面。如果不在，则直接丢弃，如果在，则把该记录传输到做连接的节点上。通过测试发现：当集合里面的值过多时，in表达式的过滤操作十分缓慢。

2. between表达式版本。当小表集合中不同值的个数小于一个阈值时，依旧采用in 表达式进行过滤，当超过了该阈值时，采用多个between 表达式进行过滤。between 表达式也是通过集合进行构造的：当集合较大时，将其中比较相近的值压缩成一个范围，每个范围构造成一个between表达式。例如集合中的数据如下：2,3,4,5,7,23,25,27,28,30,100,108,110。则可以构造三个between 表达式，分别为：between 2 and 7；between 23 and 30；between 100 and 110。

3. in和between表达式综合版本。如果只采用between表达式，对集合中比较分散的值的处理就显得不足了。因此，把in 表达式和between表达式进行结合可以更好地适应这种情形：过滤条件中有多个between 表达式和一个in表达式，这些表达式之间通过or操作符连接起来。例如集合中的数据如下：2,3,4,5,7,23,25,27,28,30,100,108,310,900。则可以构造三个between 表达式和一个in表达式，分别为：between 2 and 7；between 23 and 30；between 100 and 108；in(310,900)。

在代码实现SemiJoin算法的过程中，也存在着一些使用限制条件和注意事项，有待以后的工作去完善，具体如下：

1. 多表连接时，只能指定前两张表做semi_join。

2. 只有当两张表原来是inner join时，才能指定这两张表做SemiJoin。

3. 只有当大表的数据远远超过小表，并且两张表在连接列上的交集很小时，semi_join 优化的效果最好，其他情况下的优化效果可能不是很好。

4. 插入数据后，最好在每日合并后再做查询操作，这样的优化效果会更好。

4.4 实验设计与结果分析

为了证明SemiJoin算法的优化效果，本文做了大量的对比实验，分析了不同因素下，使用SemiJoin算法与未使用SemiJoin算法在连接响应时间上的差别。对于OceanBase集群来说，ChunkServer节点个数、大表记录个数、小表不重复值集合的大小和大表在连接列上的数据分布这几个因素对连接性能有很大的影响。故本文针对这四种因素，设计了相应的对比实验，具体如表4.1。 其中，1表示变量，

表 4.1: 四种变量选取策略

变量 / 组号	节点个数	大表记录个数	小表不重复值集合大小	数据分布
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

0表示不变量。

4.4.1 实验环境介绍

本文实验都是在HP DL360服务器上进行的，该服务器配有192GB内存和1TB硬盘，装有CentOS-6.4 操作系统。每台服务器有两个4核处理器，每个核的频率为2GHz，有两层私有高速缓存，容量分别为16KB和64KB，同一处理器内的4核共享一层2MB 大小的L3 高速缓存。不同的节点间以千兆以太网连接，两个节点间实际网络传输速率在110MB/s 到118MB/s 之间。

服务器上部署开源OceanBase0.4.2版本，该数据库源码使用C++编写，总代码量为30万行左右。本文在该源码的基础上实现了SemiJoin算法，算法所占代码量为1285行。系统采用的gcc 和g++版本均为4.4.7。

实验搭建了5个OceanBase集群，每个集群的节点个数不同。以集群O1为例，该集群由4台服务器构成：节点1上运行主RootServer和主UpdateServer，节点2上运行备RootServer 和备UpdateServer，节点3上运行ChunkServer和MergeServer，节点4上运行ChunkServer 和MergeServer。

4.4.2 实验数据介绍

实验使用两张表做连接，分别为小表R和大表S。两张表的数据由数据生成器生成，该生成器由java语言编写，生成数据总量为20G左右。两张表的属性相同，只是记录数不同，如表4.2 所示： 实验共生成一张小表，多张大表。小表记录数

表 4.2: 表属性分布

列名	数据类型	主键信息	数据分布	小表不同值集合大小
C1	int	0	均匀分布	5000
C2	int	0	正态分布	5000
C3	int	0	正偏态分布	5000
C4	int	0	负偏态分布	5000

为5万，大表记录数分别为1000万到9000万不等。

4.4.3 实验结果分析

在小表数据量为5万行，ChunkServer个数为5，小表集合大小为5000，连接列的数据分布为均匀分布时，不同的大表数据量对查询性能的影响如图4.2所示：

结果分析：该图的横坐标表示不同的大表数据量，分别从1000万行记录到9000万行记录不等。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位，这里由于响应时间差距较大，故对纵坐标采用log精度。通过对该图的分析可以得出以下

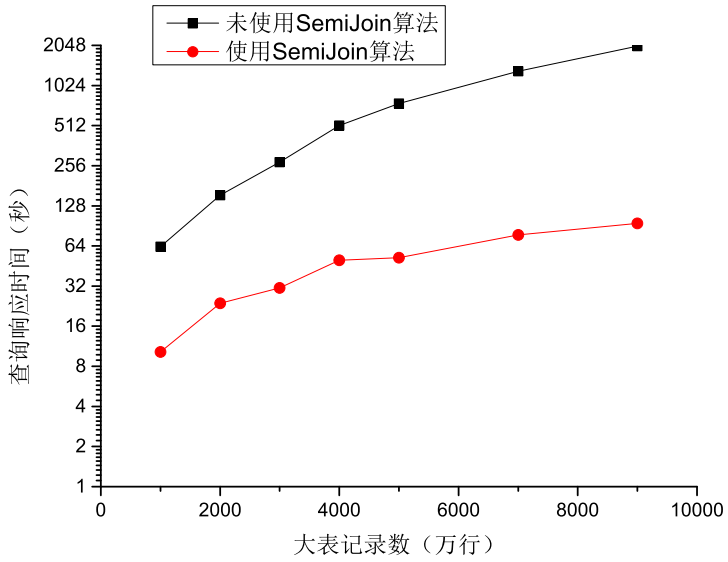

图 4.2: 不同大表数据量对性能的影响

结论：随着大表数据量的线性增加，SemiJoin算法的响应时间呈近线性增长，而未使用SemiJoin算法的连接响应时间为近指数增长。大表数据量越大，SemiJoin算法的优化性能越高。

具体原因分析：由于实验数据是均匀分布的，故当大表数据量线性增加时，每个节点过滤后的大表数据量也在线性增加，导致网络上传输的大表数据量也在线性增加。由于SemiJoin的查询响应时间受网络传输影响最大，故整个SemiJoin连接的响应时间也呈近线性增长。对于未使用SemiJoin算法的连接来说，当大表数据量超过3000万行时，出现了单点瓶颈：一个节点的内存不足以存下大表的所有数据。这导致了连接的响应时间呈近指数增长。

在小表数据量为5万行，大表数据量为3000万行，ChunkServer个数为5台，连接列的数据分布为均匀分布时，不同的小表不重复值集合大小对查询性能的影响如图4.3所示：

结果分析：该图的横坐标表示小表在连接列上不同值的个数，分别从1000到13

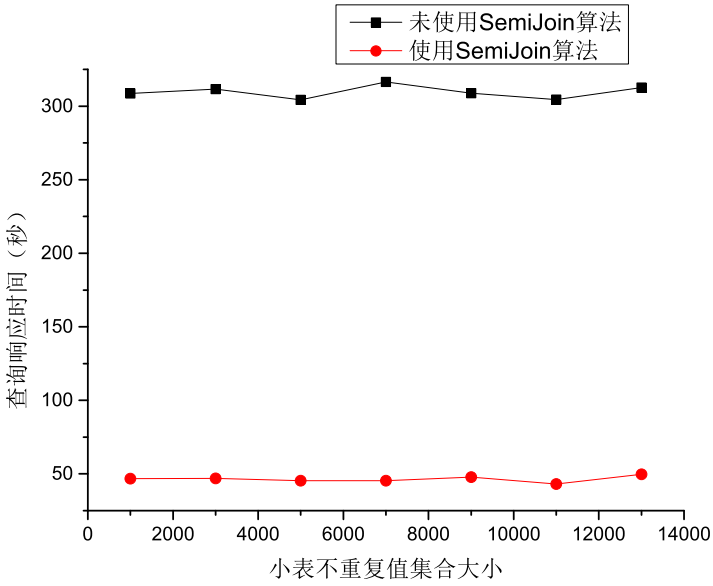

图 4.3: 不同的小表不重复值集合大小对性能的影响

000不等。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位。通过对该图的分析可以得出以下结论：小表在连接列上不同值的个数对SemiJoin的性能几乎没有影响，并且SemiJoin算法与OceanBase原有的连接算法相比，近10 倍地提高了连接操作的执行效率。

具体原因分析：当小表不同值集合不断增大时，对未使用SemiJoin算法的连接来说，查询的响应时间是基本不变的，这是因为OceanBase原有的排序归并连接不考虑小表在连接列上的数据特征，该连接只是简单的将大表和小表的全部数据通过网络通信传输到一台主节点上，然后在该主节点的内存中做排序归并连接。对于SemiJoin算法来说，由于采用的是between 表达式作为大表的过滤条件，故当小表不同值集合不断增大时，between 表达式的个数没有明显的增加，只是between 表达式的范围稍微增大了。这对过滤效果的影响不是很大，故SemiJoin的响应时间也没有很大的变化。

在小表数据量为5万行，大表数据量为3000万行，小表集合大小为5000，连接列的数据分布为均匀分布时，不同的ChunkServer个数对查询性能的影响如图4.4所示：

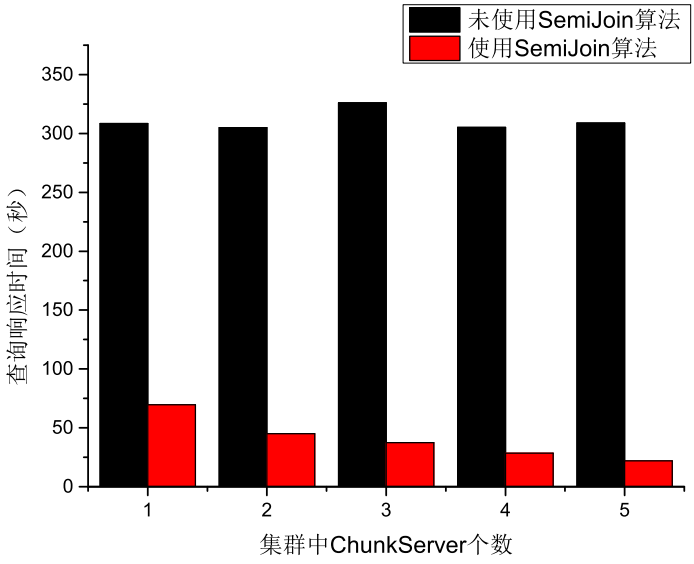

图 4.4: 不同节点个数对性能的影响

结果分析：该图的横坐标表示集群中ChunkServer个数，分别从1到5不等。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位。通过对该图的分析可以得出以下结论：随着ChunkServer个数的不断增加，SemiJoin的连接响应时间近线性减少，未使用SemiJoin算法的连接的响应时间几乎没有变化。ChunkServer的个数越多，SemiJoin的优化性能越好。

具体原因分析：当集群中ChunkServer个数不断增加时，对未使用SemiJoin算法的连接来说，查询的响应时间是基本不变的，这是因为OceanBase原有的排序归并连接只在单个节点上完成，即使集群中ChunkServer个数增加，也无法并行地做连接。对于SemiJoin 算法来说，由于大表的数据分布在多个节点上，每个节点并行地对大表数据进行过滤，故当ChunkServer个数增加时，SemiJoin的过滤操作的

并行度也在增加，每个节点的网络传输量相对减少，连接整体的响应时间也在不断降低。

在小表数据量为5万行，大表数据量为3000万行，小表集合大小为5000，ChunkServer个数为5时，不同的连接列数据分布对查询性能的影响如图4.5所示：

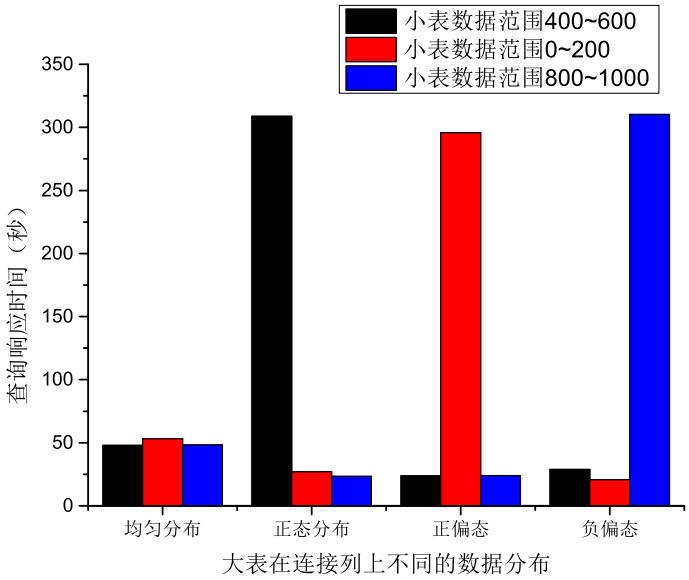

图 4.5: 不同的连接列数据分布对性能的影响

结果分析：该图的横坐标表示不同的数据分布，分别为均匀分布、正态分布、正偏态分布和负偏态分布。其中，均匀分布的数据范围分布在0到1000之间，正态分布的数据主要集中在500附近，正偏态分布的数据主要集中在100附近，负偏态分布的数据主要集中在900附近。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位。通过对该图的分析可以得出以下结论：不同的数据分布对SemiJoin的性能影响很大，当大表在连接列上的数据集中在一个峰值上时，如果该峰值在小表的数据范围内，SemiJoin的优化效果几乎为零；如果该峰值不在小表的数据范围内，则SemiJoin的优化效果达到最优。

具体原因分析：SemiJoin的优化效果与其过滤大表数据的个数是正相关的。由于SemiJoin算法采用小表数据范围进行过滤，故当大表连接列上的数据都集中在一个峰值上时，如果该峰值在小表数据范围内，则只能过滤掉很小一部分数据，网络上传输的数据量几乎为大表的全部数据，相应的连接时间也无限接近未使用SemiJoin的情况。如果该峰值不在小表数据范围内，则过滤条件能过滤掉大表的大部分的数据，网络上传输的数据量远小于大表的数据总量，相应的连接时间会被大大降低，算法优化性能达到最优。

4.5　本章小结

本章主要介绍了SemiJoin算法的设计思路，分析了SemiJoin的优化效果，并且详细介绍了在OceanBase源码上的算法实现。最后通过大量的对比实验证明：SemiJoin 算法使一张大表和一张小表的连接性能得到了显著提升。

第五章　分布式排序归并连接算法优化

排序归并连接是传统的数据库连接算法之一，在集中式数据库中，该算法性能较好，但是放在分布式数据库中，大量的网络数据传输使得该算法的效率明显降低。为此，本章在读写分离架构的分布式数据库基础上，设计了一种对排序归并连接的优化算法，大幅提高该算法在分布式数据库中的效率。并以开源分布式数据库OceanBase为基础，实现了优化后的算法。

5.1　优化动机

OceanBase的架构特点导致了其在大表连接上的效率较低。影响OceanBase中排序归并连接的响应时间的因素主要有两个：网络间传输所消耗的时间和基线数据与增量数据合并所消耗的时间。

对于第一种因素，并行处理机制是一个很有效的优化方式。在网络传输速度固定的情况下，网络间传输所消耗的时间与网络传输数据量正相关。OceanBase中的排序归并连接采用串行处理机制，故其网络间传输的数据量是两张表的数据量总和。并行处理机制的本质是Map/Reduce方法，利用多个节点传输数据，在网络带宽允许的情况下，每个节点传输的数据量被大大地减少。故采用并行处理的方式能够很好地减少网络间传输所消耗的时间。

对于第二种因素，本文提出了不先合并数据，而是对于基线和增量数据分别处理的思路。OceanBase读写分离的特点导致其在做连接之前，首先要将两张表的所有基线数据与增量数据进行合并，因为只有合并后的数据才是用户真正需要的数据，如果不合并则会导致连接产生错误结果。数据合并的具体实现是针对每一

条记录，首先从磁盘中读取该记录的基线数据，其次从其他节点中拉取该记录的增量数据，最后将该记录的基线数据与增量数据合并，产生一条新记录。可以看出，当两张表的记录数达到千万甚至上亿级别时，数据合并所花费的时间和资源是非常多的。对此，本文采用的优化方式是：对于符合连接条件的记录，合并该记录的基线数据和增量数据，而对于不符合连接条件的记录，不合并该记录的基线数据和增量数据。这样，当两张表的连接结果较少时，数据合并所耗费的时间将会被大大减少。

5.2　算法介绍

分布式排序归并连接基于MapReduce框架，在多个节点上并行地做排序归并连接。首先对连接属性做范围（range）切分，为每个节点分配一个范围。其次在多个节点之间做基线数据混洗（shuffle），将连接属性值在同一范围的记录传输到相同的节点上，同时根据连接属性范围信息对增量数据做切分，将一个范围内的所有增量数据发送到该范围对应的节点上。最后在每个节点上并行地做排序归并连接，并将连接的结果发送给主节点。

由于OceanBase是国内著名的读写分离的分布式数据库，并且已经在阿里巴巴公司的很多线上系统得到了应用。所以从实用性上考虑，算法在开源的数据库OceanBase 上做了实现。算法包括一个主节点，一个存放增量数据的节点，多个存放基线数据的节点。其中主节点负责发送请求与合并各子节点返回的连接结果。在OceanBase 中，MergeServer充当主节点角色，UpdateServer充当存放增量数据的节点，ChunkServer充当存放基线数据的节点。算法流程图如下：

如图5.1所示,算法分为五个阶段：

1. 并行统计各个ChunkServer上面两张表在连接属性a上的数据分布，并将各个节点上的数据分布信息统一发送到一台MergeServer上面。由于ChunkServer存的数据只是基线数据，想要得到正确的连接属性a 上的数据分布，还需要获得增量数据中连接属性a的数据分布信息；

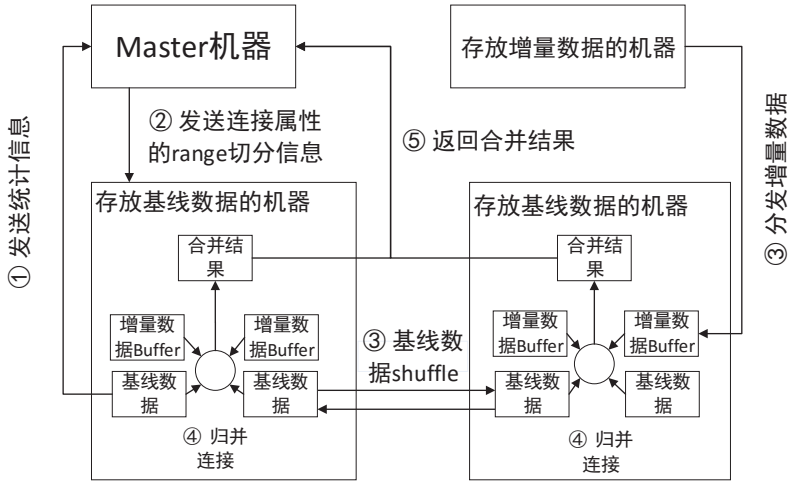

图 5.1: 分布式排序归并连接算法流程图

2. MergeServer根据每个ChunkServer传输来的a的统计信息，对连接属性a进行范围划分，每个ChunkServer对应一个范围。并把范围划分的结果发送给所有的ChunkServer以及UpdateServer；

3. 各个ChunkServer之间做数据的交互，最终每个ChunkServer 都得到了对应的连接属性范围内里的R表和S表的基线数据。同时，UpdateServer也根据连接属性的范围信息，将增量数据发送到相应的ChunkServer 上；

4. 每个ChunkServer都获得了自己对应的连接属性范围上的四部分数据：R表的基线数据和增量数据，S表的基线数据和增量数据。所有的ChunkServer并行地对自己的四部分数据做排序归并连接。并将连接的结果发送给MergeServer；

5. MergeServer整合所有的ChunkServer发送过来的合并的结果，并将结果返回给客户端。

下面主要介绍算法的第一，第三，第四阶段。

5.2.1　统计信息计算

在算法的开始阶段，主节点向各个ChunkServer发送计算统计信息的请求，也会同时把R 表和S表的在各个ChunkServer上的数据分布信息发送给UpdateServer。这里的数据分布是以R表和S表在各个ChunkServer 上的主键范围来标识的。

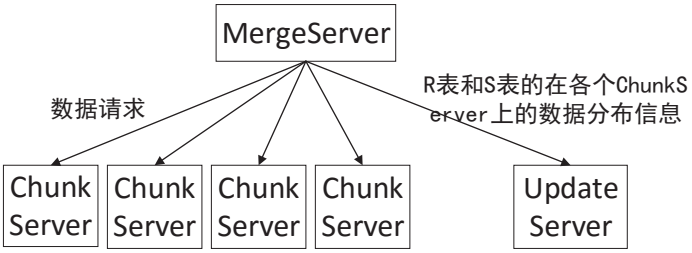

图 5.2: MergeServer发送请求

这样，UpdateServer和ChunkServer就会并行地处理请求。首先看ChunkServer这端，每个ChunkServer要把自己在相应主键范围内的R表和S表的基线数据从磁盘中读到内存里面，并计算连接属性上的统计信息。由于基线数据是以SSTable的形式存储的，一个SSTable 最大为256M。所以在一个主键范围内可能有多个SSTable。算法的实现是在ChunkServer 上开启一个线程池，用多个线程并行地计算统计信息。统计信息的计算有两种方法：抽样统计和直方图统计。前者扫描的数据少，速度快，但后者比前者更精确。这里由于数据已经被读到内存中并且是多线程计算，所以算法采用了直方图的形式来计算统计信息。每个ChunkServer计算完基线数据的统计信息后，不能立刻将结果发送给MergeServer，因为还需要获得增量数据的统计信息。

再看UpdateServer的处理，UpdateServer根据R表和S表的主键范围，对增量数据进行范围遍历。增量数据存在UpdateServer内存中的数据结构memtable里面，memtable 的本质是一个B+ 树，每张表的每一个主键对应一个叶节点，一个叶节点指向一个行操作链表，链表的每一个元素都是一个cell，记录了对该行的某一列上的修改。

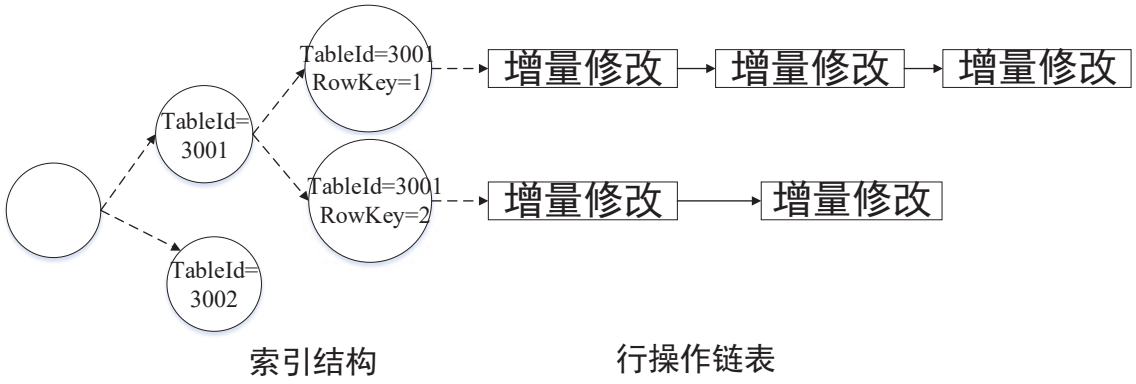

图 5.3: memtable的结构

算法使用一个类似bitmap的数据结构来存储对memtable遍历的结果：每个主键对应一个值，如果该行上面的修改没有涉及到连接属性，则该行对应的值为0，否则，该行对应的值为1，并且将该行在连接属性上的最新值也存起来。

遍历结束后，UpdateServer把遍历结果发送给相应的ChunkServer。由于在mem-table中，只有被修改的行才会作为叶节点，没有被修改的行不会在memtable上出现，所以最后遍历的结果不会很大，同时遍历都是在内存中完成的，整体速度不会太慢。

ChunkServer会根据接收到的UpdateServer上的遍历结果，对基线数据的直方图进行修改，将修改后的统计信息发送给MergeServer。同时，ChunkServer也会根据遍历结果中所有对应的值为1的主键，来修改基线数据中的相应行。如图5.4所示。

5.2.2　增量数据划分

在算法的第三阶段中，UpdateServer要根据连接属性的范围分布信息，将每个连接属性范围内的增量数据发送到对应的ChunkServer上。但是根据memtable的构造，有些行只修改了连接属性之外的属性，所以这些行的增量数据中没有连接属性的值。这时没有办法确定把这些行的增量数据发到哪个ChunkServer上。基于这种情况，本文提出了“最大范围”算法，来保证UpdateServer发给ChunkServer的

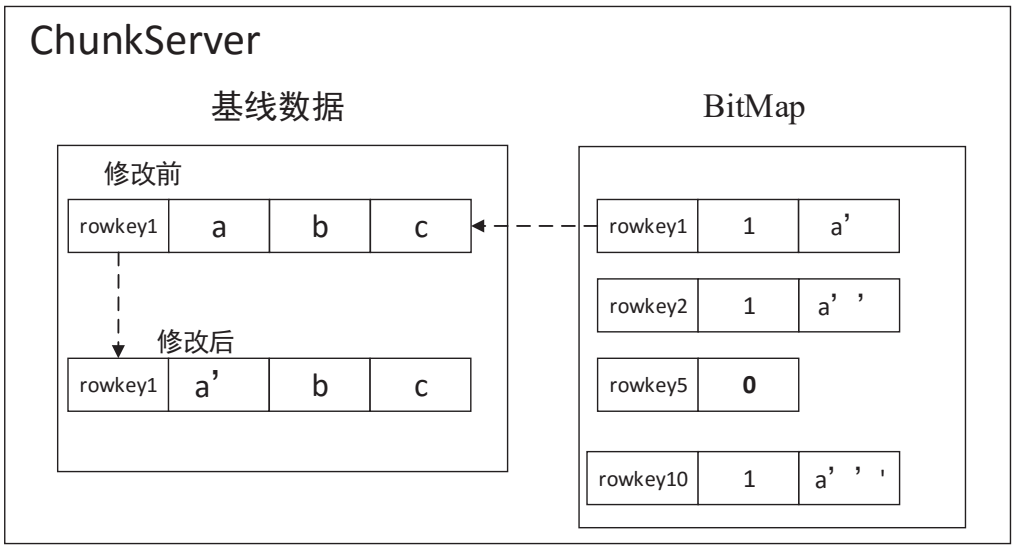

图 5.4: ChunkServer合并连接属性上的增量

增量数据包含了该ChunkServer在自己的连接属性范围上的所有增量。

最大范围算法：由于每个ChunkServer对应一个连接属性上的范围，而Update-Server只能把一个主键范围上的增量数据发给ChunkServer。所以如何确定这个主键范围是该算法的核心。最大范围算法的思路是由各个ChunkServer 发给UpdateServer多个子主键范围，UpdateServer整合所有的主键范围，生成每个ChunkServer对应的最终的主键范围，并保证该ChunkServer对应的连接属性范围上的所有增量都在这个最终主键范围内。

最大范围算法分为两个阶段。第一阶段：ChunkServer生成多个子主键范围。具体如伪代码1所示：

算法分析：在做数据交互前，每一个ChunkServer的内存中都有R表和S表在相应主键范围上的基线数据。并且这些基线数据在连接属性上的值都是最新的（上文中计算统计信息时已经将连接属性上的增量数据与基线数据合并了）。每个ChunkServer同时也拥有连接属性范围的切分信息：CS1对应[a_min1，a_max1]，CS2对应[a_min2，a_max2]，CS3对应[a_min3，a_max3]，CS4对应[a_min4，a_max4]，CS5对应[a_min5，a_max5]。

Algorithm 1 最大范围算法第一阶段

输入： $data\_scanner\_$(存储基线数据)，$join\_column\_id$(连接属性的列id);

1: **while** $data_scanner_$不为空 **do**
2: 从$data_scanner_$中取一行记录：row;
3: $ObObj\ join\_value$ = $row.raw\_get\_cell(join\_column_id)$; //获得该行在连接属性上的值;
4: $Int\ location$ = $join\_range\_partition(join_value)$; // 判断该值属于哪个范围;
5: **if** $location == 1$ **then**
6: $Buffer1.add_row(row)$; //将该行记录保存到对应的范围内;
7: **else if** $location == 2$ **then**
8: $Buffer2.add_row(row)$; //将该行记录保存到对应的范围内;
9: .....;
10: **else if** $location == 5$ **then**
11: $Buffer5.add_row(row)$; //将该行记录保存到对应的范围内;
12: **else**
13: do nothing;
14: **end if**
15: **end while**
16: $RowKeyRange\ sub\_range1$ = $Buffer1.get\_sub\_join\_range()$; //获得子主键范围;
17: .....;
18: $RowKeyRange\ sub\_range5$ = $Buffer5.get\_sub\_join\_range()$; //获得子主键范围;

每个ChunkServer在做数据Shuffle之前会在内存中申请五个Buffer: Buffer1，Buffer2，Buffer3，Buffer4，Buffer5。这五个Buffer分别对应上述的五个连接属性上的range。在做数据交互前，ChunkServer会遍历一遍内存中的基线数据（R表和S表的数据并行处理，现以R表为例），根据每一行数据在连接属性上的取值，将该行追加到相应的Buffer里面。如图5.5所示。

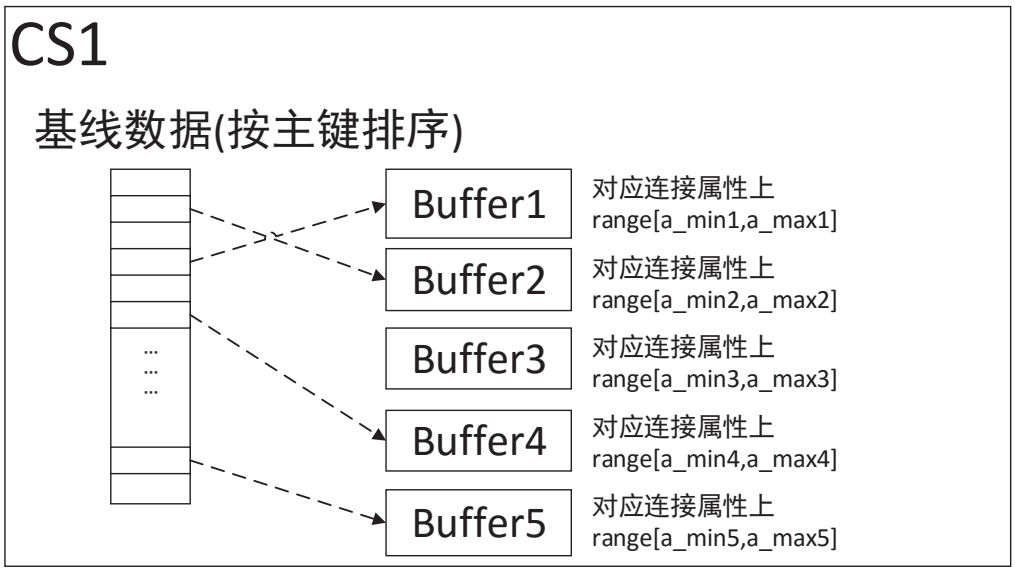

图 5.5: CS1根据连接属性的值将自己的基线数据分成5块

这样在基线数据遍历结束的时候，五个Buffer里面都被填充了很多行，由于基线数据是按照主键排序的，所以此时每个Buffer中的数据也是按照主键排序的。此时将每个Buffer内的第一行和最后一行的主键取出来，这样就产生了五个子主键范围：[k_min1，k_max1]，[k_min2，k_max2]，[k_min3，k_max3]，[k_min4，k_max4]，[k_min5，k_max5]。例如对于Buffer1来说，Buffer1内存储的主键集合为2,3,5,7,9,11,15，则Buffer1生成的子主键范围就是[2,15]。

子主键范围生成后，每个ChunkServer封装一个特殊的包（SP）发送给UpdateServer，特殊包SP里面包含的信息有：机器ip，表的id，每个连接属性上的range对应的主键range。ChunkServer发送完SP包之后再做数据的shuffle，将Buffer里的全部数据发送给相应的ChunkServer。

第二阶段：UpdateServer接收到了所有ChunkServer发送的SP包之后，根据连接属性的range，把所有SP包的信息整合起来。如图5.6所示。

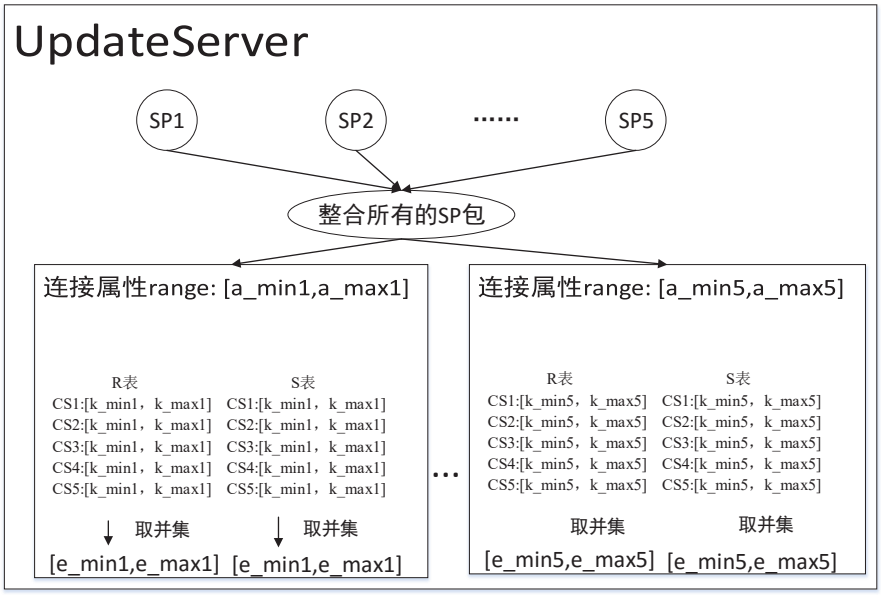

图 5.6: UpdateServer整合所有的SP包

整合结束时，5个连接属性的range分别对应了R表和S表的5个最终主键范围。例如[a_min1，a_max1]对应了R表的[e_min1，e_max1]和S表的[e_min1，e_max1]。此时，UpdateServer会根据这些最终主键范围遍历memtable，将范围内的增量数据发给相应的ChunkServer。

5.2.3　基线数据与增量数据归并连接

当算法进行到第四阶段的时候，每个ChunkServer的内存中已经有了四块数据。如图5.7所示。

对R表和S表的基线数据做归并连接（如算法2所述）：从每张表的基线数据中取一行记录开始匹配，如果符合连接条件，则根据该行的主键到该表的增量数据中二分查找，如果找到，则将该行的增量数据和基线数据合并，并把合并后的结果放到结果集中。如果不符合连接条件，则将连接字段较小的记录抛弃，从这条记录对应的表中取下一条记录继续进行匹配，直到整个循环结束。

Algorithm 2 局部归并连接

输入: row_store_r (存储R表数据), row_store_s (存储S表数据), $join\_column\_id$(连接属性的列id);

while row_store_r不为空或者row_store_s不为空 **do**
  从row_store_r中取一行记录：row_r;
  从row_store_s中取一行记录：row_s;
  根据$join_column_id$从row_r里面获得该行在索引列上的值：$join\_value\_r$;
  根据$join_column_id$从row_s里面获得该行在索引列上的值：$join\_value\_s$;
  if $join\_value\_r < join\_value\_s$ **then**
    从row_store_r中取下一行记录;
  else if $join\_value\_r > join\_value\_s$ **then**
    从row_store_s中取下一行记录;
  else if $join\_value\_r == join\_value\_s$ **then**
    根据row_r和row_s的主键到R表和S的增量Buffer中查找这两行;
    if R表的增量Buffer中找到了row_r的主键 **then**
      将row_r与该行的增量数据合并;
    else if S表的增量Buffer中找到了row_s的主键 **then**
      将row_s与该行的增量数据合并;
    else
      do nothing;
    end if
    将合并后的row_r与row_s存到$result_buffer$中。
  else
    do nothing;
  end if
end while

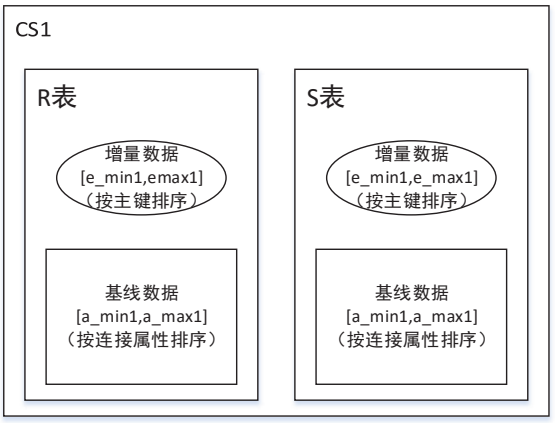

图 5.7: CS1内存中准备进行连接的数据

5.3　分布式排序归并连接算法分析

5.3.1　算法正确性

算法通过以下细节保证了算法结果的准确性：

1. 在统计连接属性上的信息的时候，不是只把对基线数据的统计结果发送给UpdateServer，而是将连接属性上的增量数据也算入到统计信息里面。这样保证了在切分连接属性range的时候，能够根据真正的统计信息，做出正确、均匀的range切分。

2. 最大范围算法的正确性：在基线数据交互过程中，每个ChunkServer对应一个连接属性上的range，每个ChunkServer会接收到其他ChunkServer发送来的属于该range里的数据块，以及自己本地的属于该range里的数据块。这些数据块都有主键范围，取这些主键范围的并集，得到的最终的主键范围。虽然最终的主键范围并不精确，因为它是不连续的。但是该ChunkServer在连接属性range上的所有数据都在这个主键范围内。所以最终在做排序归并连接的时候，每一行基线数据肯定能够从增量Buffer中找到对应的增量数据（如果该行被修改了的话）。

5.3.2 算法效率分析

与传统的排序归并连接算法相比，改进后的算法极大地提高了连接的效率。设两张表的基线数据行数为$nBase$，增量数据行数为$nIncrease$，连接结果的行数为$nResult$，每一行数据在网络上的传输时间为$tnet$，每一行增量数据和一行基线数据合并的时间为$tmerge$，内存中每排序一行数据的时间为$tsort$，排序归并连接时每处理一行数据的时间为$tjoin$，节点的个数为N。则可得到传统的连接算法下的连接时间T_old:

$$T\_old = (nBase + nIncrease) * (tnet + tmerge + tsort + tjoin) \tag{5.1}$$

优化后的算法处理时间T_new:

$$T\_new = \frac{(nBase + nIncrease) * tnet}{N} + \frac{(tsort + tjoin) * nBase}{N} + \frac{nResult * tmerge}{N} \tag{5.2}$$

由公式可以看出，优化后的算法明显地减少了基线与增量数据合并所用的时间，当节点个数N较大时，T_new远小于T_old。

下面从三个方面详细介绍下算法如何提升了连接效率：

1. MergeServer在切分连接属性范围时，会根据每个ChunkServer上的统计信息，尽量使得切分后的每个范围里的数据量是平均的。如果总的数据量是m，对m行数据做排序归并连接的时间为s，范围个数为r。则如果每个范围里的数据量是平均的，最终的连接处理时间为$\frac{s}{r}$。如果每个范围里的数据量是不平均的，假设最大的范围的数据量为m^1，则最终的连接处理时间为$\frac{s*m^1}{m}$。同时，在给每个CS分配范围时，如果某个范围内的数据大部分都在ChunkServer1上，则就把该范围分配给ChunkServer1。这样就使得网络间交互的数据量最少，减少做数据交互所用的时间。

2. 使用最大范围算法避免了UpdateServer将所有的增量数据发给每个ChunkServer。如果没有最大范围算法，为保证结果的正确性，UpdateServer需要将每张

表所有的增量数据发送给每个ChunkServer，这样在做merge join时，基线数据才能够从增量Buffer中找到自己的增量数据。但由于每张表的所有增量很多，这会导致UpdateServer与ChunkServer之间的网络开销特别大。如果使用最大范围算法，UpdateServer 只需要将一个主键范围内的增量数据发送给ChunkServer，虽然这个主键范围内的数据会比ChunkServer真正需要的增量数据多，但是与发送所有的增量数据相比，最大范围算法提升了一定的效率。

3. 算法在很多处地方都采用了并行的实现：第一阶段ChunkServer和UpdateServer并行地处理MergeServer发送的请求；第三阶段基线数据的交互和增量数据的划分并行执行；第四阶段多个ChunkServer之间是并行地做排序归并连接。并行处理能够充分地利用分布式数据库的特点，极大地提高算法的效率。

5.3.3 算法适用性

之所以选择在OceanBase上实现该算法，是因为OceanBase的增量数据全部集中在一台机器上，基线数据分布式地存储在多台机器上。基线数据和增量数据不在一台机器内，真正实现了物理上的分离。这一特性更好地符合了该算法的特点，使得该算法在OceanBase 上的优化效果更好。对于其他分布式数据库来说，该算法也可以使用，只需要在实现时根据数据库的特点做不同的修改。以HBase为例，Hbase的基线数据和增量数据是存储在一台机器上的，则只需要在算法的第三阶段增加不同机器之间增量数据的交互，算法的整体思路就会在Hbase上得到实现。

5.4 算法对增量数据的特殊处理

增量数据一般存储在内存中，不同的数据库采用不同的数据结构来存储增量数据。以OceanBase为例，增量数据是存储在一个叫做内存表的结构里面。下面详细介绍一下内存表的结构。

5.4.1　内存表结构

内存表本质上是一个B+树。它的非叶子节点是由表id和主键组成，叶节点则存储某张表的某一行的所有增量。例如想要更新表t1中主键为R1的行的某一个非主键列c1，则首先根据t1的表id查找B+树，找到t1对应的节点，再通过R1 查找该节点的孩子节点，如果找到某个叶子节点对应的主键为R1，则在该叶子节点指向的链表的尾部再增加一个链表节点，该链表节点的值为c1更新后的值。这里需要指出的是，叶子节点指向的链表不存储该行的所有列的值，它只存储该行被修改的列的值。如果某一列未被修改过，它不存在于内存表中。

可以看出，一张内存表可以存储数据库中所有表的增量数据。数据库刚启动的时候，内存表初始为空。当有事务来临时，如果事务更新的行在内存表里面不存在，则在内存表内新增节点。随着事务增多，内存表的大小会逐渐增加，当增加到一个阈值的时候，会触发内存表的冻结操作。冻结操作会将该内存表转储到磁盘上，并且在内存中新建一张空的内存表。所以，OceanBase系统运行时最多只有一张内存表，但是可能有多张被转储在磁盘中的冻结内存表。

针对OceanBase内存表的这些特性，本文在算法设计的时候提出了最大范围算法来进行增量数据的分发：

由于内存表的结构限制，OceanBase只支持通过主键或者主键范围向内存表拉取增量数据。而本文算法在实现的时候，是为每个节点分配一个连接列上的范围，根据这个连接列范围向内存表拉取该范围内的增量数据。这里就出现了问题：当连接列不是主键的时候，无法通过连接列的范围向内存表拉取该范围内的增量数据。

解决这个问题的关键就是：通过连接列的范围生成一个近似的主键范围，然后通过这个近似的主键范围向内存表拉取增量数据，并且保证连接列范围内的基线数据所对应的增量全部都在该近似的主键范围内。上文提出的最大范围算法就是用来求出这个近似的主键范围的。

5.4.2 数据修改操作

增量数据的类型有很多种，OceanBase支持四种对数据修改的操作：insert，replace，update，delete。这四种操作在memtable上的增量数据类型不同。所以在算法的第一阶段，UpdateServer给ChunkServer发送遍历memtable 的结果时，需要对不同类型的增量数据进行特殊处理：

1. 因为delete操作在memtable的表示是只在该行的行操作链表中增加一个删除的标识，所以即使该行的行操作链表中没有连接属性上的值，也要把该行对应的bitmap里的值置为1，并且连接属性的值为空。

2. 对于insert操作来说，基线数据是没有新插入的行的，所以ChunkServer在根据UpdateServer的遍历结果修改基线数据的时候，需要在基线数据中添加那些新插入的行。

3. 对于replace操作来说，基线数据中可能有该行，也可能没有。所以ChunkSer-ver在根据UpdateServer的遍历结果修改基线数据的时候，如果某一个主键对应的bitmap里面的值为1，但是在基线数据中找不到该主键，同样也需要在基线数据中添加一行。

5.5 实验设计与结果分析

为了验证改进后的分布式排序归并算法的正确性与效率，本章设计了一系列实验。实验的思路是通过对比OceanBase开源版本与改进后版本对相同sql的处理时间与处理结果，并根据对比结果得出结论。

5.5.1 实验环境介绍

本文实验都是在HP DL360服务器上进行的，该服务器配有192GB内存和1TB硬盘，装有CentOS-6.4 操作系统。每台服务器有两个4核处理器，每个核的频率为2GHz，有两层私有高速缓存，容量分别为16KB和64KB，同一处理器

内的4核共享一层2MB 大小的L3 高速缓存。不同的节点间以千兆以太网连接，两个节点间实际网络传输速率在110MB/s 到118MB/s 之间。

服务器上部署开源OceanBase0.4.2版本，该数据库源码使用C++编写，总代码量为30万行左右。本文在该源码的基础上实现了分布式排序归并连接算法，算法所占代码量为8475 行。系统采用的gcc和g++版本均为4.4.7。

实验搭建了2个OceanBase集群，每个集群的节点个数不同。以集群O1为例，该集群由7台服务器构成：节点1上运行主RootServer和主UpdateServer，节点2上运行备RootServer 和备UpdateServer，节点3,4,5,6,7上运行ChunkServer和MergeServer。

实验采用的数据是使用数据生成器随机生成的数据，数据集的大小从1G到6G不等。为了保证两个集群使用同样的数据，我们首先把随机生成的数据存到文件中，然后使用工具ObImport将该文件导入两个集群中。

5.5.2　实验结果分析

在集群中ChunkServer个数为5个，基线与增量数据比例为1:1时，不同的大表数据量对连接结果的影响如图5.8所示：

结果分析：该图的横坐标表示两张表不同数据量，分别从500万到3000万不等。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位。通过对该图的分析可以得出以下结论：随着表数据量的不断增加，使用了优化算法的连接响应时间近线性增加，未使用优化算法的连接响应时间近指数级增加。表数据量越大，算法的优化效果越好。

具体原因分析：当大表数据量线性增加时，对于优化后的算法来说，每个节点网络传输的数据量呈线性增加，故整个连接的响应时间也呈近线性增长。对于未使用优化算法的连接来说，当大表数据量超过1500万行时，出现了单点瓶颈：一个节点的内存不足以存下大表的所有数据。这导致了连接的响应时间呈近指数

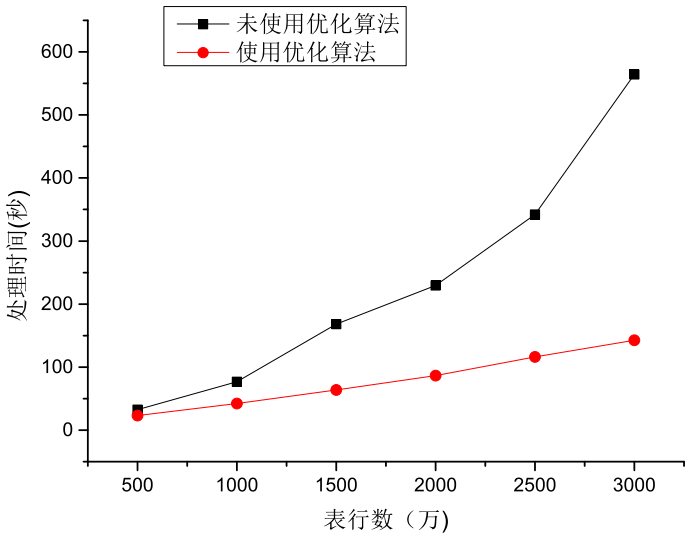

图 5.8: 5台CS下的连接时间比较

增长。

在集群中ChunkServer个数为10个，基线与增量数据比例为1:1时，不同的大表数据量对连接结果的影响如图5.9所示：

结果分析：该图的横坐标表示两张表不同数据量，分别从500万到3000万不等。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位。通过对该图的分析可以得出以下结论：使用了优化后的算法后，ChunkServer节点个数的增加会大幅减少连接的响应时间；而未使用优化算法的连接，ChunkServer节点个数的增加对响应时间几乎没有影响。

具体原因分析：该图与图5.8相比，只是增加了ChunkServer节点个数，但是优化后的算法在相同数据量下的响应时间呈现大幅减少趋势。这是因为增加集群中节点个数会导致算法的并行性增加，每个节点的网络传输量相对减少，故整体响应时间随之减少。而未使用优化算法的连接不存在并行计算，故增加ChunkServer节点个数不会影响到查询的响应时间。

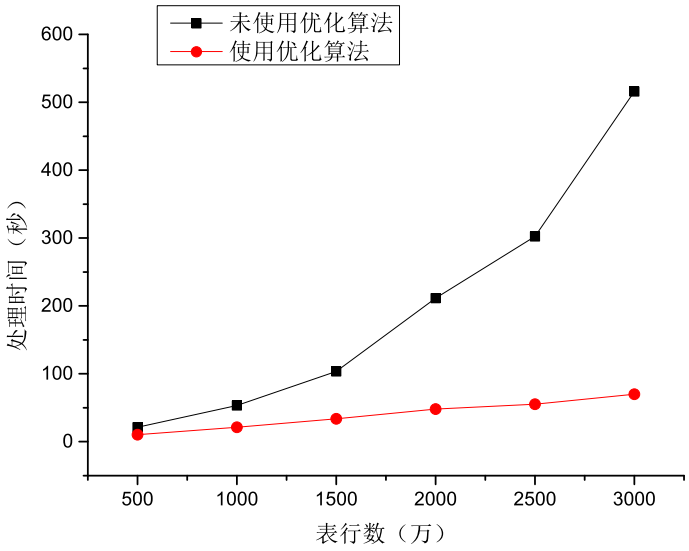

图 5.9: 10台CS下的连接时间比较

在集群中ChunkServer个数为10个，大表数据量为1000万行，基线与增量数据比例变化对连接结果的影响如图5.10所示：

结果分析：该图的横坐标表示增量数据占总数据量的比例，分别从0.1到0.7不等。纵坐标代表每个连接的查询响应时间，具体为从用户输入连接语句到数据库返回连接结果的时间间隔。查询响应时间以秒为单位。通过对该图的分析可以得出以下结论：随着增量数据占总数据量的比例增加，未使用优化算法的处理时间呈近线性增长，使用了优化算法的处理时间呈先下降后上升的趋势。

具体原因分析：当增量数据占总数据量的比例越来越大时，未使用优化算法的连接因为要做合并操作的增量数据越来越多，导致网络传输量增加，故整体响应时间呈近线性增长。而对于使用了优化算法的连接：当基线数据减少时，ChunkServer之间数据交互的数量变少，整体的处理时间下降。但是当增量数据越来越多的时候，增量数据的网络传输时间增加，导致整体的处理时间上升。

除了通过对比实验来测试算法的优化性能，本文还参考了TPC-H基准测试对算法优化后的OceanBase进行了进一步测试。TPC-H基准测试是由TPC-D发展而

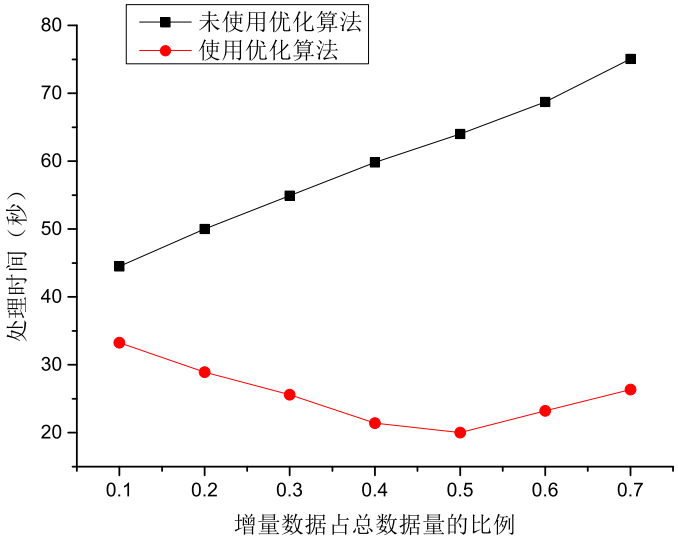

图 5.10: 基线与增量数据比例变化对响应时间的影响

来，包括22个查询，其主要评价指标是各个查询的相应时间，其度量单位是每小时执行的查询数。本文使用了TPC-H不同大小的数据集，修改了查询语句。最终生成的测试结果如图5.11：

结果分析：该图的横坐标表示做连接的表的大小，分别从1GB到30GB不等。纵坐标代表每小时处理的查询数，纵坐标越大，表示查询的性能越好。通过对该图的分析可以得出以下结论：随着表数据量的增加，优化后的算法和未使用优化的算法都呈下降趋势，但是优化后的算法性能仍然明显高于未优化的算法。

通过以上实验和测试可以看出，改进后的算法的确极大地提高了OceanBase对连接操作的处理效率。并且ChunkServer个数越多，改进后的算法效率越高。

5.6　本章小结

本章首先介绍了分布式排序归并连接算法的设计思路与五个主要流程。其次详细分析了算法的正确性与高效率。在对增量数据的处理中，提出了“最大范围”算法用来对增量数据和基线数据分别处理，避免了对两张表的全部数据做合

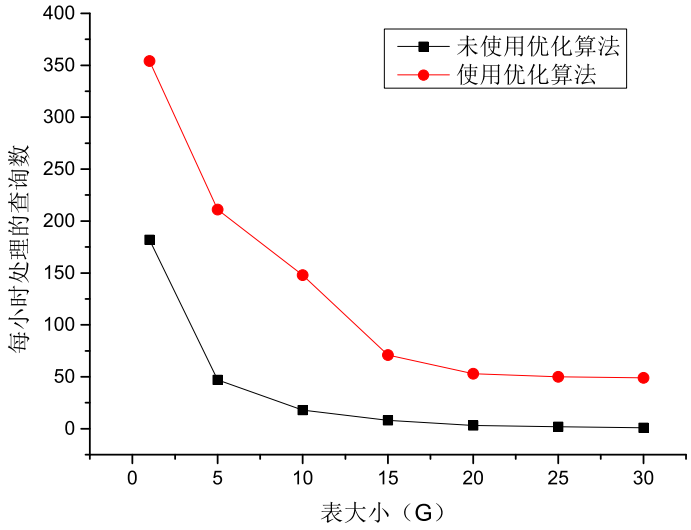

图 5.11: TPC-H对比结果

并。最后通过一系列的对比实验验证了算法的高效率。

第六章　总结与展望

6.1　本文总结

读写分离架构的分布式数据库是近几年的研究热点，该架构在保持高可扩展性的同时也支持事务的ACID 特性，对一些现象级的互联网应用来说，该架构的数据库是一个很好的选择。目前国内最具代表性的读写分离架构的数据库是阿里巴巴开发的OceanBase数据库。OceanBase 已经全面支撑了淘宝，支付宝，聚划算等应用，在双十一活动，秒杀活动中的表现也非常显眼。

没有任何数据库是完美的，读写分离架构的数据库的最大缺点就是增加了查询的复杂度。每次查询都要先将基线数据和增量数据合并，并返回合并后的结果。目前学术界对该架构查询优化的研究较少，故本文以大表连接为切入点，提出了不先合并数据，而是对基线和增量数据分别处理的优化思路，并设计和实现两种大表连接算法：SemiJoin和分布式排序归并连接。

SemiJoin优化算法主要针对大表和小表的连接。该算法首先读取小表的全部数据，构造小表在连接属性上的取值集合。其次根据该集合构造对大表的过滤条件，并将该过滤条件分发到所有包含大表数据的节点上，使得每个节点并行地过滤大表数据，并将过滤后的数据通过网络传输发送到一台主节点上。最后，算法在该主节点的内存中对小表的全部数据和大表过滤后的数据做排序归并连接。可以看出，该算法避免了在网络上传输大表的全部数据，通过过滤条件极大地降低了大表要做连接的记录数。过滤条件的构造十分重要，直接关系到大表在网络上的传输量。故本文在SemiJoin算法中提出了一种综合构造过滤条件的思路。该思

路的输入是小表的取值集合，输出是多个between表达式和in 表达式。大量的对比实验验证了改进后的SemiJoin算法极大减少了查询的响应时间。

分布式排序归并连接算法针对的是两张大表的连接。算法的优化思路是并行计算，首先将两张表的数据shuffle 到不同的连接属性范围内，其次每个范围内并行地做排序归并连接，最后把所有的连接结果汇总到主节点上。在数据shuffle 的过程中，本文提出了最大范围算法，该算法对基线数据和增量数据分别处理，避免了合并数据操作，提高了查询效率。最后，本文通过大量的对比实验，验证了该算法的高效率。

6.2　未来工作

OceanBase作为一个开源数据库，目前还有很多不是很完善的地方。未来工作可以以连接算法优化为切入点，深入研究OceanBase的查询优化机制，为OceanBase增加统计信息、代价计算模型、多个物理操作符等，构成一个基本的查询优化框架，弥补OceanBase目前在查询优化方面的不足。

对于SemiJoin优化算法，本文只是优化了过滤条件的构造过程，但是没有避免掉基线与增量数据的合并操作。未来工作可以重点考虑如何在保证正确性的前提下，避免大量数据合并。具体实现可以参考分布式排序归并连接算法中的最大范围算法，对基线数据和增量数据分别处理。

对于分布式排序归并连接算法，算法流程中第一步构造统计信息阶段占用了很大一部分时间，未来工作可以集中在如何避免统计信息的计算上。具体实现是引入统计信息机制，每次连接前直接查询统计信息，这种实现的好处是避免了计算统计信息的时间，缺点是增加了维护统计信息的开销。另一种改进方式是引入缓存机制，缓存上一次计算统计信息的结果。在没有缓存缺失的情况下，算法第一阶段花费的时间可以忽略不计。

本文重点讨论了两张大表连接的情况，对于多表连接，如何根据统计信息和中间结果集的大小来确定连接的顺序，以及如何对多表的数据同时进

行MapReduce操作等优化方法，都是以后对连接优化算法研究的重点。

参考文献

[1] Rick Cattell. Scalable SQL and nosql data stores. SIGMOD Record, 39(4):12–27, 2010.

[2] Lars George. HBase - The Definitive Guide: Random Access to Your Planet-Size Data. O'Reilly, 2011.

[3] Kristina Chodorow and Michael Dirolf. MongoDB - The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly, 2010.

[4] Michael A. Cusumano. In defense of IBM. Commun. ACM, 58(10):27–28, 2015.

[5] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. IACR Cryptology ePrint Archive, 2016:116, 2016.

[6] Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP theorem. IEEE Computer, 45(2):30–36, 2012.

[7] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. ACM Trans. Comput. Syst., 31(3):8, 2013.

[8] Nikita Shamgunov. The memsql in-memory database system. In IMDM@ VLDB, 2014.

[9] 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, 2013.

[10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2):1626–1629, 2009.

[11] Tom White. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.

[12] Kosuke Aritake, Yuji Kado, Tsuyoshi Inoue, Masashi Miyano, and Yoshihiro Urade. Structural , functional characterization of hql-79, an orally selective inhibitor of human hematopoietic prostaglandin d synthase. Journal of Biological Chemistry, 281(22):15277–15286, 2006.

[13] Dhruba Borthakur. Hdfs architecture guide. HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf, 2008.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, 2008.

[15] 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.

[16] David B Enfield and JS Allen. On the structure and dynamics of monthly mean sea level anomalies along the pacific coast of north and south america. 1980.

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

[18] Patrick Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. ACM Transactions on Database Systems (TODS), 9(1):133–161, 1984.

[19] 刘敏，王意洁. 并行i/o 技术研究. 计算机应用研究, 20(8):29–31, 2003.

[20] 谭怀亮，王燕，孙建华，陈浩. 分布式系统卷重构过程的改写块预取方法. 湖南大学学报: 自然科学版, 36(1):77–80, 2009.

[21] Hans P Zima, Heinz-J Bast, and Michael Gerndt. Superb: A tool for semi-automatic mimd/simd parallelization. Parallel computing, 6(1):1–18, 1988.

[22] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pages 145–156. ACM, 2002.

[23] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. Cache conscious algorithms for relational query processing. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pages 1–10. IEEE, 2010.

[25] Yi Luo, Wei Wang, and Xuemin Lin. Spark: A keyword search engine on relational databases. In ICDE, pages 1552–1555, 2008.

[26] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In CIDR, volume 5, pages 225–237, 2005.

[27] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In Proceedings of the 2011 ACM

SIGMOD International Conference on Management of data, pages 37–48. ACM, 2011.

[28] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pages 362–373. IEEE, 2013.

[29] Lior Aronovich and Israel Spiegler. Cm-tree: A dynamic clustered index for similarity search in metric databases. Data & Knowledge Engineering, 63(3):919–946, 2007.

[30] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. Cloudramsort: fast and efficient large-scale distributed ram sort on shared-nothing cluster. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pages 841–850. ACM, 2012.

[31] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. Proceedings of the VLDB Endowment, 5(10):1064–1075, 2012.

[32] Peter MG Apers, Alan R Hevner, and S Bing Yao. Optimization algorithms for distributed queries. Software Engineering, IEEE Transactions on, (1):57–68, 1983.

[33] 石小艳，李秀华. Sdd-1 查询优化算法的研究与改进. 科技信息(学术研究), 28:064, 2008.

[34] Deniz Yuret and Michael De La Maza. Dynamic hill climbing: Overcoming the limitations of optimization techniques. In The Second Turkish Symposium on Artificial Intelligence and Neural Networks, pages 208–212. Citeseer, 1993.

[35] 张薇，马建峰. Lpca——分布式存储中的数据分离算法. 系统工程与电子技术, 29(3):453–458, 2007.

[36] 刘放美，王猛. 分布式查询优化算法及对sdd-1 算法的改进. 科技广场, 2:84–88, 2005.

[37] 闫保权. 蚁群聚类lf 算法在matlab 中的实现. 信息技术, (3):143–145, 2013.

[38] 陈和平，张前哨. A 算法在游戏地图寻径中的应用与实现. 计算机应用与软件, 22(12):118–120, 2005.

[39] 田立中，付宜利，马玉林，谢龙. 装配路径规划中基于动态坐标的a^* 搜索算法. 计算机集成制造系统, 8(4):316–319, 2002.

[40] 孟军，李建强，张大鲲. 应用聚簇索引的多连接查询优化方法. 大连理工大学学报, 1, 2003.

[41] 徐银如. 线性最优设计中w－算法收敛性的一般证明. 华东师范大学学报: 自然科学版, (2):1–7, 1990.

[42] Zhe Li and Kenneth A Ross. Perf join: An alternative to two-way semijoin and bloomjoin. In Proceedings of the fourth international conference on Information and knowledge management, pages 137–144. ACM, 1995.

[43] 杨旭超. 基于半连接的分布式数据库查询优化算法探讨. 计算机时代, (2):16–19, 2012.

[44] 钱磊，于洪涛. 改进的半连接查询优化算法. 燕山大学学报, 36(2):178–182, 2012.

[45] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In Proceedings of the 2013 ACM

SIGMOD International Conference on Management of data, pages 13–24. ACM, 2013.

[46] 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.

[47] Ilya Grigorik. Sstable and log structured storage: Leveldb, 2012.

[48] Kevin R Fall and W Richard Stevens. TCP/IP illustrated, volume 1: The protocols. addison-Wesley, 2011.

[49] 张树刚，张遂南，黄士坦. Crc 校验码并行计算的fpga 实现. 计算机技术与发展, 17(2):56–58, 2007.

[50] Jim Gray and Andreas Reuter. Transaction processing: concepts and techniques. Elsevier, 1992.

致　　谢

时光匆匆，一眨眼三年的研究生生活即将结束。在华东师范大学计算机科学与软件工程学院的这三年时间里，我的导师、实验室师兄师妹、同学和家人等都给予我很多关心和帮助，使我在研究生阶段得到了很大的成长。在此，我深深地感谢所有关心我，帮助过我的人们。

首先，我要感谢周傲英教授领导的海量计算研究所。这是一个令人温暖的大家庭。它让我们有机会接触数据库领域国际一流的研究人员，在国内领先的设备环境下进行科学研究。周老师学识渊博，讲解问题深入浅出，每次向周老师请教问题都能让我受益良多。

我还要感谢周敏奇老师。周老师一直以高标准严格要求自己，对科研一丝不苟又充满激情。这些都深深地感染了他的学生们。生活上，周老师平易近人，给予我很大帮助。

我特别感谢高明老师，高老师极为认真负责地指导了我的科研论文，让我在一次次的修改和更新中总结并改进自己的不足，教会了我对科研问题思考的方式和认真做好事情的态度。

我非常感谢海量所的钱卫宁老师，宫学庆老师，张蓉老师，蔡鹏老师，张召老师，王晓玲老师等，他们在课堂上或者讨论班上指导过我的学习，扩大了我的知识面。钱卫宁老师指导我对OceanBase 的研究方向，向我提了很多学术上的建议，让我受益良多。宫学庆老师指导我在交通银行的实习，在生活上和学术上给予我很大的帮助，让我得到了极大的成长。

我由衷感谢OB组的兄弟姐妹们（周欢、张晨东、庞天泽、翁海星、刘骁等）近三年的陪伴，在OB组大家一起奋斗的日子让人怀念。我还要感谢王佳豪、王东

惠、龙飞、张春熙、李捷莹等学弟学妹对我论文的修改和建议。我更要感谢我的家人对我的理解和支持，让我能够更加集中精力参与科研学习。

最后希望所有的老师、同学、家人和朋友能够身体健康，工作顺利。

樊秋实

二零一六年四月六日

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] **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，计算机学报. 2016. (EI)

■ 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管理系统研究与开发，2015－2017，参加人