2017 届硕士专业学位研究生学位论文（全日制研究生）

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51141500087

華東師範大學

East China Normal University

硕士专业学位论文

MASTRER'S DEGREE THESIS (PROFESSIONAL)

论文题目：OceanBase 分布式数据库中聚合运算性能优化

院 系 名 称：计算机科学与软件工程学院

专业学位类别：工程硕士

专业学位领域：软件工程

指 导 教 师：宫学庆 教授

学 位 申 请 人：熊 辉

2016 年 10 月 20 日

Dissertation for professional master's degree in 2017

University Code: 10269

Student ID: 51141500087

East China Normal University

Title: **Aggregation Optimization In Distributed Database OceanBase**

Department:	**School of Computer Science and Software Engineering**
Professional degree category:	**Master of Engineering**
Professional degree field:	**Software Engineering**
Supervisor:	**Prof. Gong XueQin**
Candidate:	**Xiong Hui**

October, 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《OceanBase 分布式数据库中聚合运算性能优化》，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

《OceanBase 分布式数据库中聚合运算性能优化》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）。

熊辉 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
翁楚良	教授	华东师范大学	
张蓉	副教授	华东师范大学	
蔡鹏	副教授	华东师范大学	
沙朝锋	副教授	复旦大学	

摘　要

OceanBase 由于增量数据和基线数据分离的特点，聚合运算如 sum，count，avg，min，max 的性能非常低下，主要有以下两个原因，首先，系统要满足实时查询的需求，必须先将增量数据和基线数据融合再做聚合运算，这个过程涉及到节点的数据传输，占用大量的响应时间。其次，聚合运算涉及排序，聚合等操作，当读取的基线数据量很大时，磁盘开销就是一个很大的瓶颈。

为了优化 OceanBase 的聚合运算，本文设计了一种数据结构 LocalCube，缓存基线数据的聚合结果，避免基线数据端的重计算，加快读取流程。

本文的主要贡献：

(1) 提出基于 LocalCube 的聚合运算方案，将传统物化视图的思想应用于 OceanBase 架构中。该方案在基线数据大，更新少的场景中能极大地提高系统的实时处理能力。

(2) 提出基于 LocalCube 聚合运算的改进方案，在新行分组选择率低的情况下，能进一步优化聚合查询性能，实现了 LocalCube 批量更新方案。

(3) 提出根据增量数据增量修正基线数据 LocalCube 的方案，从而高效探测基线表行数的变化，实时计算表的行数。

(4) 实现并度量了本文提出的算法，给出大量实验证明方案的可行性和高效性。

关键词：聚合运算，分布式数据库，预计算，查询优化，分布式缓存；

Abstract

Since the incremental data and baseline data is stored separately in OceanBase, aggregation operation (sum, count, avg, min, max) performance is low. There are mainly two reasons, first of all, In order to meet the requriments of real time query, incremental data and baseline data must be merged before aggregation in oceanbase database, this process involves the data transmission between incremental server and baseline server, which take a lot of response time. Then, the aggregation operation process involves sorting and other time costly operations. when the database`s baseline data is large, the disk overhead is bottleneck.

In order to optimize the aggregation operation in OceanBase, this paper designs a data structure named LocalCube, which cache the aggregation result of the baseline data to avoid repeatedly computation, thus improving aggregation performance in OceanBase.

The main contributions of this paper are as follows:

(1) Design an optimization scheme based on LocalCube. The idea of traditional materialized view is applied to the OceanBase architecture. The database real-time processing ability can be greatly improved using this scheme in the scene with large baseline data and little modifies.

(2) Propose an improved aggregation alogrithm based on LocalCube, which can further optimize the aggregation query when the new row group selectivity is low. Implement LocalCube batch update strategy in OceanBase architecture.

(3) Propose LocalCube incremental update scheme based on incremental data.The changes of the baseline data row and the table row numbers can be

calculated in the shortest time.

(4) Implement and measure the aggregation algorithm proposed in this paper. A large amounts of experiments has been carried out to demonstrate the validity and the efficiency of the scheme.

Keywords：Aggregation，Distributed Database，Pre-Computation，Query Optimization，Distributed Cache；

目录

插图

表格

第一章 绪　论

1.1 研究背景

在过去几十年的时间里，数据库系统经历了层次数据库[1]、网状数据库[2]、关系数据库和对象数据库[3]几个发展阶段。当下，关系型数据库成为应用最广泛的数据库，应用于我们生活的方方面面，如银行，电信，教育，电子商务等。

关系模型是在 1970 年由 IBM 的研究员 E.F.Codd 博士首先提出[4]，在之后的几十年中，关系模型的概念得到了充分的发展并逐渐成为主流数据库结构的主流模型。传统的关系型数据库很好地支持事务，满足 ACID 属性，即原子性（Atomicy）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）[5]。

随着信息技术的不断发展，社交网络，电子商务，移动互联网的发展和壮大，很多大型企业的数据规模达到 TB(Terabyte)甚至 PB（Petabyte）级别，许多互联网企业，如 Google、Facebook、腾讯、阿里巴巴、百度等，它们对数据库的需求也变的更加的高，主要体现在以下四个方面[6]：

(1) 这些互联网应用要求数据库有更高的并发处理能力。如今的购物网站和购票网站，要求数据库几万和几十万的并发，特殊情况下，比如在春运期间或者在淘宝双 11 促销期间，甚至要求数据库达到几百万的并发。

(2) 这些互联网应用要求数据库有更好的伸缩性，要求数据库能在若干天甚至若干小时之内有几倍、几十倍的伸缩能力，来满足突发的热点访问，比如，春运购票，淘宝促销等。

(3) 这些互联网应用要求数据库有更好的性价比，与传统的高端服务器、高端处理器相比，互联网公司的数据库系统由许多低成本，高性价比的普通 PC 服务器构成。互联网的业务发展迅速，不能依靠传统的纵向扩展，

即通过提高单台服务器的处理能力来满足业务的需求。而是需要通过增加普通 PC 服务器来横向扩展，提高系统的处理能力。

(4) 这些互联网应用要求数据库有更高的可用性，普通 PC 服务器性价比高，但故障率也明显高于高端服务器，需要在软件层来实现故障处理，保证系统的高可用。

因此，传统的关系型数据库在大数据时代的应用中显的力不从心，首先，传统的关系型数据库无法处理海量数据，它的存储和查询能力无法满足互联网应用的需求；其次，在可扩展性上面临着严峻的挑战，它的事务以及二维关系模型很难高效的扩展到多个存储节点上；最后，不能很好的支持高并发应用，目前许多网站的并发量高达每秒数万次，传统关系型数据库中的磁盘 I/O 就是一个很大的瓶颈。

为了解决关系数据库面临的可扩展性、高并发以及性能方面的问题，许多 NoSQL[7]系统应用而生，NoSQL 开始理解为 no sql，发展到 no relational，也就是指非关系型数据库。它们一般使用键值对存储数据，数据耦合性较低，有良好的扩展性，一般不支持 ACID 特性，它们根据不同的需要可以分为以下几类：

(1) 满足高性能并发读写的键值对（Key-Value）数据库，典型的代表数据库有 Redis[8]、Tokyo Cabinet[9]和 Flare[9]。Redis 本质上是一个 Key-Value 类型的内存数据库，数据全部加载到内存中进行操作，因此它的性能非常出色，每秒可以处理超过 10 万次的读写请求。

(2) 满足海量存储需求和访问的面向文档的数据库，典型的代表数据库有 MongoDB[11]和 CouchDB[12]。MongoDB 是一个介于关系型数据库和非关系型数据库的产品，拥有强大的数据查询功能，它自带的分布式文件系统 GridFS[13]可以很好的支持海量数据存储。

(3) 面向可扩展的分布式数据库，典型的数据库有 Cassandra[14]和 Voldemort[15]。Cassandra 是一个分布式的数据库，数据分布在各个节点上，可以在不停机的情况下添加和删除数据节点，具有很好的可扩展

能力。

上述的非关系型数据库确实能解决一部分应用的需求，它们的读写效率较高，但是数据的一致性不能得到很好的保证。但是如今的互联网公司业务有很多涉及结构化数据的业务，例如淘宝、天猫的商品搜索广告计费，支付宝的移动支付等等，都是商务交易和金融交易，因此数据库的事务（Transaction）[16]功能不可或缺。因此，如今的互联网应用既需要关系型数据库的事务特性，也需要它可扩展，支持海量数据的存储和访问。

经过调研发现，在目前的许多互联网应用中，应用涉及的数据量虽然很大，但是在一段时间（比如一天）内增加，删除，修改的数据总量却不大，人口数据库，在人口数据库中保存了每个人的姓名，身份证号，户籍，家庭住址等信息，全国一共 14 亿人口，如果每人对应一条记录，那么人口数据库总共有 14 亿条记录。但每天修改的记录并不多，例如新增人口、死亡人口、户口迁移的数量大概在百万级别，和总记录数相比，只是一个很小的比例。

由于数据库一天的增删改的量相对于数据库的总量比例很小，可以用单台服务器来保存一段时间的更新数据即增量数据，以前的数据保持不变，叫做基线数据，基线数据按主键排序划分到多台基线服务器上，当更新服务器上的增量数据积累到一定程度，系统将增量数据转储到基线服务器端。根据这类应用的特征，阿里巴巴集团设计并实现了 OceanBase[17]分布式数据库，可以良好的支持该类应用。

OceanBase 不仅满足关系型数据库 ACID 的特性，支持跨行跨表事务，又有很好的可扩展，支持海量数据的存储和访问，并成功的应用于支付宝，淘宝等交易业务。

1.2 研究内容

数据库的聚合运算有 sum，count，avg，max，min 等，他们是关系型数据库重要的数据分析运算，一般和分组操作 group by 一起使用，基本形式如下：

select [group by attributes] aggregates from {relations}

[where {predicates}]

group by {attributes}

having {predicates}

由于 OceanBase 增量数据和基线数据分离的特点，聚合函数的性能比较低下，有以下几个原因：

(1) 系统要满足实时查询的需求，必须先将增量数据和基线数据融合再做聚合运算，这个过程涉及到节点的数据传输，会产生较长的响应时间。

(2) 聚合运算涉及排序，聚合等操作，当读取的基线数据量很大时，磁盘开销就是一个很大的瓶颈。

在传统的关系型数据库中，聚合运算具有多种优化方案，常用的方法是采用物化视图(Materialized View)[18]的方式，即预先计算比较耗时的聚合操作，并把结果保存在数据库中，在查询执行时，直接使用该物化视图，避免了耗时的聚合运算，从而快速得到结果。在分布式的场景下，数据分散在多个物理节点上，聚合运算的效率受到磁盘性能、网络传输性能和 CPU 性能的影响，在 OceanBase 的架构下还涉及到增量数据和基线数据的合并，传统数据库的物化视图并没有考虑这些因素，因此，如何将传统数据库的物化视图和自身架构结合成了迫切研究的课题。

1.3 主要贡献

本文以 OceanBase 数据库为研究对象，针对动态数据，静态数据分离聚合运算性能低下的问题，设计和实现了新的算法，提高聚合函数计算性能。贡献点总结如下：

(1) 提出基于 LocalCube 的聚合运算方案，将传统物化视图的思想应用于 OceanBase 架构中。该方案在基线数据大，更新少的场景中能极大地提高系统的实时处理能力。

(2) 提出基于 LocalCube 聚合运算的改进方案，在新行分组选择率低的情况下，能进一步优化聚合查询性能，实现了 LocalCube 批量更新方案。

(3) 提出根据增量数据增量修正基线数据 LocalCube 的方案，从而高效探测基线表行数的变化，实时计算表的行数。

(4) 实现并度量了本文提出的算法，给出大量实验证明方案的可行性和高效性。

1.4 本文结构

本文结构组织如下：

第一章，绪论部分。包括研究背景，主要介绍数据库的发展以及为了解决当今互联网应用的需要产生的数据库 OceanBase。以及研究内容，介绍聚合运算的相关知识。接着是主要贡献，提出基于 LocalCube 的聚合运算方案以及在其基础上的优化，最后介绍本文的组织结构。

第二章，介绍本文研究内容的相关工作。主要介绍聚合运算的分类，有全局性聚合函数和分布性聚合函数，并介绍了聚合运算的相关技术，有预计算，查询转换，Hash 聚合，抽样，并行处理这几种技术。

第三章，OceanBase 简介与问题描述。介绍 OceanBase 的整体架构，读取流程，更新流程，数据存储等方面的内容，最后介绍 OceanBase 上聚合运算的不足，提出问题和优化方向。

第四章，基于 LocalCube 的聚合运算实现。首先给出了 LocalCube 的数据结构，接着讨论了 LocalCube 的存储，创建，维护和管理，最后介绍 LocalCube 在 OceanBase 上的实现。

第五章，基于 LocalCube 的聚合运算优化。从优化网络传输作为切入点，在 LocalCube 的基础上，提出了一个改进方案。

第六章，实验。设计了大量的对比实验，比较了优化前和优化后聚合运算性能的变化。

第七章，总结和展望。总结本文工作以及对未来的展望。

第二章 聚合运算分类和相关技术

聚合运算是数据库操作中的常用操作，如 sum、count、avg、max、min 等等。聚合运算分为两类，一种为分布性聚合运算，一种为全局性聚合运算。而聚合运算技术主要有以下几种，包括预计算，查询转换技术，Hash 聚合，抽样，并行处理等方法。

2.1 聚合运算分类

常见的聚合运算有 sum、count、avg、max、min 等，它们大致可以分为两类[19]，一类是分布的，另一类是全局的。

定义 2.1.1：分布性聚合运算：如果聚合运算的最新结果仅由它存在的值和操作(插入/删除)的值算出（插入操作为新值，删除操作为旧值），那么该聚合运算就是在该操作上（插入或删除）分布的。

其中，sum 和 count 对插入操作和删除操作都是可分布的，它们的最新结果能由现存值和当前操作的值算出。比如，当前维护了某列的 sum 和 count 值，对应插入操作来说，只需将 sum 值加上插入的值，count 值对应加 1 则获得最新的结果；对于删除操作只需将 sum 值减去删除的值，count 值对应减 1 则获得最新的结果。Avg 的值能由 sum 和 count 算出，因此 Avg 对插入操作和删除操作也是可分布的。

min 和 max 对插入操作是可分布的，若当前维护了某列上的 min 值或 max 值，对于插入操作来说，只需要比较当前插入的值和维护的 min 值或者 max 值，则可以计算出最新的值。

定义 2.1.2：全局性聚合运算：如果计算聚合运算的最新结果所需要的存储空间没有固定的界，那么该聚合运算就是全局的。

min 和 max 对删除操作是全局的，若当前维护了某列上的 min 值或 max 值，对于删除操作来说，若最大值或者最小值删除了，只能通过扫描全表得到结果。

定义 2.1.3：增量可计算：数据库对已有表的操作通常有三种，insert，delete 和 update，其中，update 操作可以被看成一个删除操作(删除旧值)和一个插入操作（插入新值），则数据库可以看成插入和删除操作的集合，如果对于插入和删除操作，聚合运算的新值能由它的原值和增量计算出，那么这个聚合运算就是增量可计算的。

其中，sum，count 和 avg 在插入和删除操作上都是分布的，因此 sum，count，avg 是增量可计算的；min 和 max 在插入操作上是分布的，在删除操作上是全局的，因此 min 和 max 不是增量可计算的[20]。

2.2 聚合运算技术

2.2.1 预计算

预计算是聚合运算通常采用的优化方案，所谓预计算，就是在初次运行的时候系统维护一部分的统计值，在查询的时候直接使用该统计值，减少数据的扫描量，提高运算速度。

物化视图常常应用于比较成熟的集中式数据库中，如 Oracle 10g[21]，Mysql[22]，DB2[23]。它和普通视图不同，普通视图是一张虚拟表，它不存储任何数据，只存放数据的定义，对视图的查询会转换成对基本表的查询，而物化视图存放的是预计算好的数据，可以把它看成一张物理表。

物化视图一般由用户显示建立，常用的 sql 命令为：

```
create materialized view mv as select col1,sum(col2) from table1 group by col1;
```

系统会执行对应的查询语句并将聚合后的结果存放在物化视图中，之后的查询可以直接访问物化视图，这样可以加快查询。

为了保持数据的一致性，物化视图必须与原始数据的变化保持同步，物化视

图的维护的过程就是和原始数据保存同步的过程，按维护方法来说可以分为两种[24]：

(1) 重计算法：当基表更新时，通过直接访问基表，对视图重新计算，这种方法实现简单，使用也比较广泛，适合基础表更新比例比较大的场景，但是重新计算涉及大量的计算开销，对于实时性要求较高的场景不太实用[25]。

(2) 增量计算法：物化视图的增量维护过程一般如图 2.1 所示[26]，当基本表更新时，系统首先判断哪些物化视图会受影响，然后这些更新通过物化视图的定义转换成对物化视图的更新，最后系统把这些更新应用到对应的物化视图中，将原表的修改转换成物化视图的修改。增量计算的特点是计算开销小，适合基础表更新比较小的场景。有很多关于物化视图增量维护的工作，文献[27][28]中提及的算法主要应用在集中式数据库的场景下，文献[29][30]提及了一种基于增量表达式的算法，通过查找最优的增量表达式来缩短增量维护的时间。

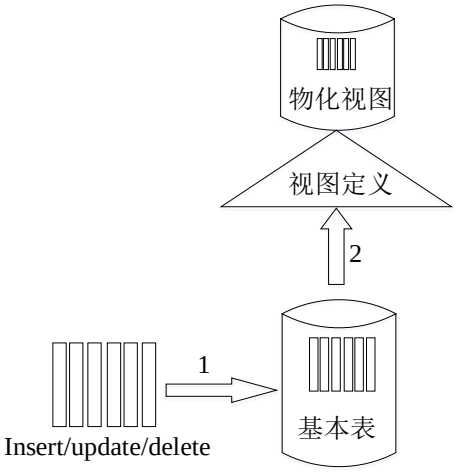

图 2.1 物化视图更新

物化视图会占用一定的硬件资源，它需要和原数据保存一致，系统的维护代价会随着视图个数的增加而增加。因此，存在一个物化视图的选择问题，如何从众多的视图中选择一组存储，维护以及查询代价最小的物化视图是研究的关键。

这些算法分为两种[31]：

(一)**静态算法**：由于事先不能确定系统的查询集合，先假设查询在数据上是均匀分布的，即用户的查询对每份数据访问的概率是相等的。基于这样的假设，文献[32]提出了基于多维数据表格(立方块)的 BPUS 算法，文献[33]提出了一种以物化视图的尺寸为选择标准的 PBS 算法。

(二)**动态算法**：动态算法能克服静态算法不能响应查询请求动态变化，导致物化视图总收益下降的缺点，它能根据查询类型的分布动态的选择物化视图。文献[34][35]提出了基于单位空间上的查询频率的视图选择方法 FPUS，能很好的反映查询的需求。文献[36][37]提出了基于粗糙集聚类的物化视图的动态调整算法 RSCDMV，能解决用户查询多样性的问题。

2.2.2 查询转换

WP. Yan 等人研究了连接和聚合的先后关系对查询性能的影响，在文献[38][39]中作者提出了一类查询转换的方法，通过将查询树上的分组操作上移和下移，产生了两种策略，Lazy aggregation 和 Eager aggregation。

Eager Aggregation 是将部分的分组操作放到连接操作之前，先分组再连接，在分组操作部分下移后，我们依然需要在上层的查询中执行原来的分组操作，Eager Aggregation 减少了连接操作的输入行数，因此可能会产生一个相对较好的总的执行计划。

Lazy Aggregation 与前一种策略不同，它是先执行连接操作再统一执行分组操作。它的主要优点是如果连接很有选择性，那么分组的输入行数就会减少。基于 TPC-D 基准测试证明了这种转换在聚合查询包含分组时非常有用[41]。

2.2.3 Hash 聚合

在 sql server[42]数据库中，为了解决流聚合(Stream Aggregation)[43]的不足，应对大数据的要求，Hash 聚合被提出。哈希聚合的实现方法和哈希连接一样，需要哈希函数的内部运算，形成不同的哈希值，依次并行扫描数据形成聚合值。

sql server 系统会动态的根据查询的情况选择合适的聚合方式，当要聚合的列包含大量重复值时，系统会选择哈希聚合[44]。

以下两个相似查询在系统具有不同的执行计划，如图 2.2 所示。对于 SQL1，系统会推荐流聚合，对于 SQL2，系统会推荐 hash 聚合。

```
SQL1：
Select ClassID,count(*) from Scores  group by ClassID;

SQL2：
Select StudentID,count(*) from Scores group by StudentID;
```

图 2.2 Sql-Server 哈希聚合示例

StudentID 和 ClassID 的分组查询看上去很类似，但是执行计划却不同，因为 ClassID 包含了大量的重复值，StudentID 重复值非常少，所以 Sql server 系统给 ClassID 推送的哈希聚合，而 StudentID 推送的是流聚合。

2.2.4 基于抽样的聚合

在数据库中有一部分的 SQL 需要长时间运行，延时高。为了解决 SQL 运行时间长的问题，有人提出了带时间约束的 SQL 查询，即让一个长时间运行的 SQL 在指定的时间内返回结果。

Oracle 实现了带时间约束的查询[45]，它根据用户对返回结果准确性的接受程度实现两种策略。

(1) 当部分查询结果可以接受时，系统通过减少结果集的基数来满足时间约束。用户可以指定让查询尽快的返回前几行，或者排序结果的前几行。

(2) 当近似的查询结果可以接受时，系统通过抽样来满足时间约束。

```
SELECT … FROM … WHERE …
GROUP BY … HAVING …
ORDER BY …
[[SOFT | HARD] TIME CONSTRAINT (T)
[WITH {APPROXIMATE | PARTIAL}  RESULT]];
```

图 2.3 Oracle 带时间约束查询的语法

图 2.3 为 Oracle 带时间约束查询的语法[46]，上述语法是对 select 语法的扩展，TIME CONSTRAINT 是该 SQL 的时间约束，T 为时间，单位是秒；WITH 语句为结果集返回的类型，是近似结果还是部分结果。如果没有指明 soft 和 hard，默认是 soft，如果没有指明 approximate 和 partial，默认是 approximate。

如果用户指明结果集返回的类型为 approximate，那么该 SQL 会被系统转换成带 SAMPLE 语句的 SQL，如果用户指明结果集返回的类型为 partial，那么该 SQL 会被系统转换成带 ROWNUM 语句的 SQL，示例如下：

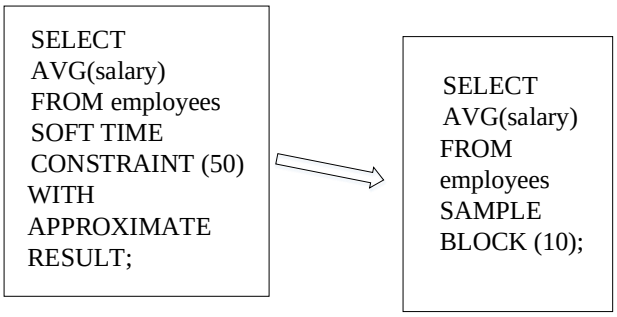

图 2.4 基于抽样的查询转换

图 2.4 的 SQL 是用户的显示输入，它要求查询大约在 50s 完成并接受近似结果，上述的 SQL 会被系统重写（重写是系统内部的优化，对用户不可见），右图的 SAMPLE 语句指明了抽样比例，即在计算工资平均值时只有 10%的数据块被访问到（10%是每一行被访问到的概率）。

2.2.5 并行处理

随着信息技术的不断发展，社交网络，电子商务，移动互联网的广泛使用，很多大型企业的数据规模达到 TB(Terabyte)甚至 PB（Petabyte）级别，分布式系

统成为了解决数据存储和访问的有效途径，聚合运算也充分的利用了分布式系统的特点，通过并行处理的方法，来提高聚合运算的性能。聚合运算并行处理算法通常有两种：

(1) 集中的两阶段算法[47]

Phase1：多处理器的每个节点先在本地关系的分区上做一次聚合运算得到部分的结果。

Phase 2：各个节点再把这些部分的结果发到一个集中的协调节点，协调节点把这些部分的结果融合成最终的结果。

(2) 重新分区算法[48]

首先，基于分组列将表重新分布到不同的节点，然后每个节点在不同的分组上做聚合，并行产生最终的结果。

上述的算法在处理不同的分组选择率时效果不佳，第一种算法只在结果集行数较小时性能不错，第二种算法只在分组后的组数很大时表现很好。上述的两种算法有各自的适应范围，如何将这两种算法融为一体，让它们发挥各自的优点，基于这种想法，Ambuj Shatdal 等人[49][50]提出了基于抽样的自适应算法，分别是自适应集中两阶段算法和自适应重新分区算法，如图 2.5 所示。

```
对关系抽样，找到抽样中组的数目；
如果 组的数目小于临界值
   使用两阶段算法；
否则
   使用重新分区算法；
```

图 2.5 基于抽样的自适应算法

2.3 本章小结

本章主要介绍了聚合运算的分类以及常见的聚合运算技术。

聚合运算可以分为两类，一类为分布性聚合运算，另一类为全局性聚合运算，其中 sum，count，avg 是分布性的，max 和 min 对插入操作是分布性的，对删除

操作是全局的。

集中式的数据库采用物化视图(Materialized View)的方式保存预计算结果，物化视图采用实时或延迟的方式和基本表同步，查询直接访问物化视图，避免了重计算的开销，在许多传统的数据库中广泛应用。在分布式的场景下，聚合运算利用了数据多节点分布的特点，将聚合查询分为多个子查询到各个节点并行处理，再由统一节点汇总部分结果。查询转换技术是通过变换聚合和连接的次序来减少中间结果集的大小，从而优化聚合性能。若查询能接受不精确的聚合结果，那么基于抽样的查询就是一个很好的方法，它能在用户指定的时间内返回结果。

上述的聚合方案都有各自适应的场景，但是都没有考虑 OceanBase 的架构特点，即增量数据和基线数据分离以及基线数据按主键排序划分到多台基线服务器，因此，本文结合了上述物化视图和并行处理思想，给出了自己的解决方案，将其应用到 OceanBase 的架构中，这是本文的创新和贡献。

第三章 OceanBase 简介与问题描述

3.1 背景介绍

如今社交网络，电子商务，移动互联网的发展产生了海量的数据，它们对数据存储和管理的要求也更高。阿里巴巴集团为了满足自身应用的需求，在 2010 年，开发了一款可扩展的分布式关系型数据库 OceanBase，它很好的满足了当今互联网应用的需求：

(1) 支持事务的 ACID 特性。这些特性能保证天猫和淘宝等商业交易业务的可用性和正确性。

(2) 低成本、高可靠。OceanBase 使用一组廉价的 PC 服务器，通过软件层的自动容错和故障恢复机制，对外提供高可靠服务。它可以替代传统昂贵的高端服务数据库，极大的降低运营，生产成本。

(3) 易扩展，高性能。OceanBase 采用基于内存计算技术的动态数据、静态数据分离架构的分布式数据库，支持动态的添加/删除服务器，能自动进行负载均衡和数据迁移。

OceanBase 在设计之初就考虑了上层应用的特点，比如，淘宝的收藏夹，虽然它的数据量特别庞大，有几百亿条数据，但是每天的更新却很少，一般只有几千万条到几亿条数据。因此，OceanBase 采用单台更新服务器来记录最近一段时间的更新，把基线数据分布在多个节点上，这种方案避免复杂的分布式事务，获得数据的高一致性以及集群的高可扩展性。OceanBase 从 2010 年设计开始截止到 2016 年，它已经支撑了淘宝、天猫和支付宝的上线业务，并经历了双十一的考验，成为国内最成熟的，支持线上海量数据的分布式关系型内存数据库。

3.2 OceanBase 介绍

3.2.1 OceanBase 数据模型

如图 3.1 所示，OceanBase 的数据由两部分构成，基线数据和增量数据。其中，增量数据又称动态数据，是一段时间（每天）的更新数据，包括增(insert)，删(delete)，改(update)，保存在更新服务器（UpdateServer）的内存中，称为 Memtable。基线数据又称静态数据，是数据库在 Memtable 开始时刻的快照，分割后存放在基线服务器（ChunkServer）的磁盘中，称为 SSTable。

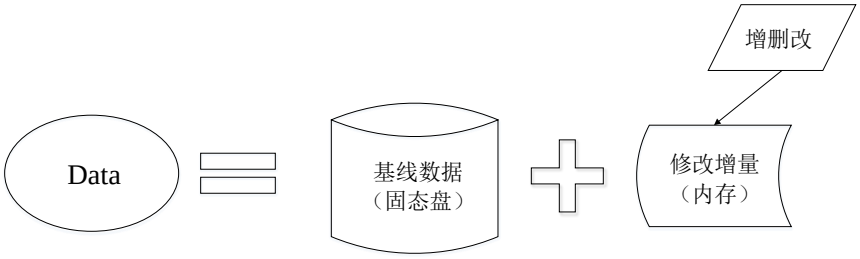

图 3.1 OceanBase 数据模型

3.2.2 OceanBase 架构介绍

OceanBase 以增量方式把当天增删改的数据保存在更新服务器的内存中（Redo log 保存在磁盘），称为 MemTable；对应的基线数据（即数据库在 MemTable 开始时刻的快照）则分割后保存在基线服务器的磁盘（通常是固态盘）上，称为 SStable。增量数据集中的放在一个节点上，而基线数据则分开存放，可以既可以避免复杂的分布式事务，如两阶段提交[51]，同时又有良好的扩展性。

OceanBase 集群中 ChunkServer 和 MergeServer 可以扩展能动态的添加和删除，但是 UpdateServer 和 RootServer 不能，若 UpdateServer 或 RootServer 宕机了，那么系统处于不可服务的状态，这是不能接受的，因此，OceanBase 采用主备的机制提供更可靠的服务，当主机宕机，备机能自动当主并对外提供服务，

UpdateServer 的可靠性是通过 RootServer 来保证的，Rootserver 通过发放租约来选择主 UpdateServer，当主 UpdateServer 发生故障时，RootServer 在租约有效期结束后，从备 UpdateServer 中选择一台当主。

RootServer 采用一主一备，主备之间数据强同步，所有的操作都通过操作日志的方式，先同步到备机才能返回成功。它们的切换是通过 linux 系统自带的 HA[52]服务，主备 RootServer 之间共享 Virtual IP，当主 RootServer 发生故障时，Virtual IP 能自动的漂移到备 RootServer 所在的机器上，备 RootServer 的后台线程检查到 Virtual IP 漂移自身后切换为主，对外提供服务。

图 3.2 是一个 OceanBase 集群的示意图，主 RootServer 和主 UpdateServer 部署在一台服务器上，备 RootServer 和备 UpdateServer 部署在另一台机器上，主备 UpdateServer 采用强同步策略，更新操作日志同步到备机后才返回客户端成功。ChunkServer 和 MergeServer 可以动态的扩展，满足海量数据的需求。

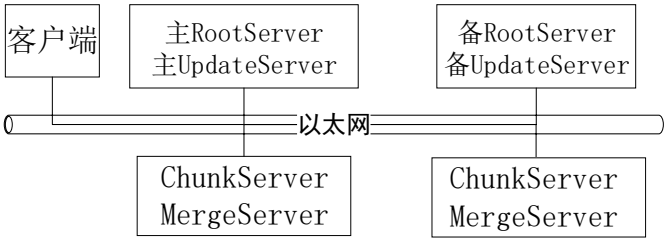

图 3.2 OceanBase 单集群架构

一个 OceanBase 集群由以下四种角色构成：

(1) RootServer：实现数据分布、副本管理、集群管理等功能。RootServer 维护了所有子表的分布信息，为查询提供数据所在节点的地址信息。RootServer 管理数据副本，保证每台 ChunkServer 上存放大致相同的数据量，当某台 ChunkServer 负载过重，触发副本迁移命令。另外，RootServer 与 MergeServer 和 ChunkServer 之间保持心跳（heartbeat），感知 server 的上下线。

(2) UpdateServer：接受系统的写事务，存储系统的增量数据。UpdateServer 为一主一备或一主多备，只有主 UpdateServer 上能接受更新，备 UpdateServer 通过日志和主保存同步。

(3) ChunkServer：存储系统的基线数据，提供读取服务，执行每日合并以及数据分发。ChunkServer 可以横向扩展，支持动态添加/删除服务器，

有较强的扩展性。

(4) MergeServer：接受客户端查询或更新请求，解析 SQL 生成执行计划，提供请求分发和结果合并功能。MergeServer 兼容 MySQL 协议，最大限度的方便用户使用。

3.2.3 查询流程

OceanBase 在处理查询时，必须先将基线数据和增量数据融合，这个过程涉及增量服务器和基线服务器的网络交互。OceanBase 的查询流程如图 3.3 所示：

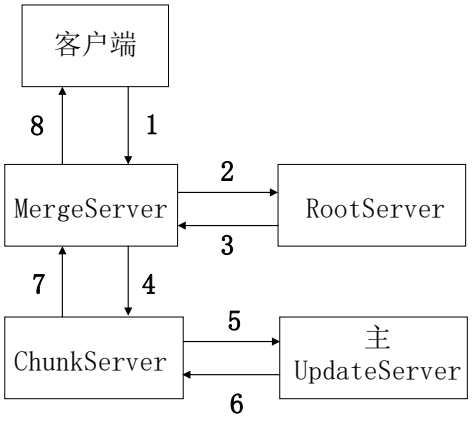

图 3.3 OceanBase 查询流程

首先，MergeServer 接受到客户端的查询请求，解析查询语句并生成查询计划。 在第 2，3 步，MergeServer 去 RootServer 请求数据所在的 ChunkServer 信息。当 MergeServer 获取到对应的 ChunkServer 位置后，把物理计划发送到对应的 ChunkServer 上。

然后，在第 5，6 步，ChunkServer 向 UpdateServer 请求对应的增量数据，Updateserver 返回 ChunkServer 增量数据后，ChunkServer 将本地基线数据和增量数据融合，执行物理计划生成部分结果，将部分结果返回给 MergeServer。

最后，MergeServer 融合各个 ChunkServer 发来的结果，生成最终结果并将结果返回客户端。

3.2.4 写事务流程

OceanBase 的写事务涉及执行计划解析，基线数据获取，操作日志同步备机等几个过程，具体的流程如图 3.4 所示：

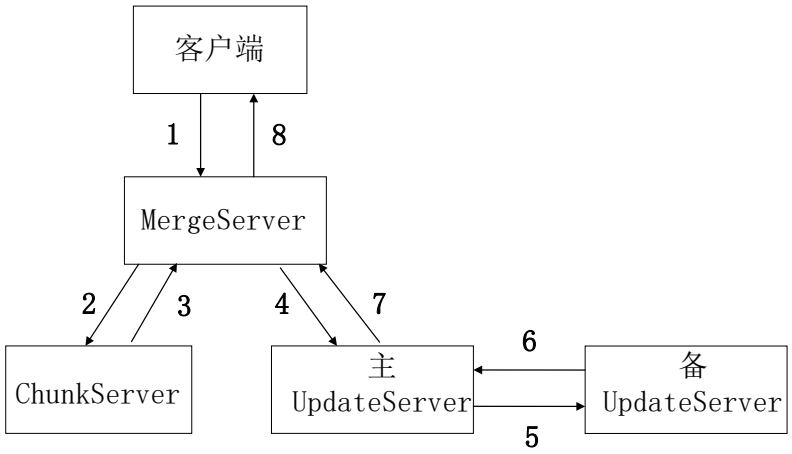

图 3.4 OceanBase 写事务流程

首先，如第 1-4 步所示，MergeServer 接受到客户端的写事务请求，解析 sql，经过词法分析、语法分析，生成逻辑计划，生成物理计划。MergeServer 查询 ChunkServer 上事务所需的基线数据，并将物理计划和基线数据一起发给主 UpdateServer。

然后，步骤 5-7，主 UpdateServer 先写操作日志，然后同步给备 UpdateServer，最后修改本地 Memtable 并返回 MergeServer 写事务成功或者失败。

最后，MergeServer 返回客户端成功或失败。

3.2.5 定期合并

UpdateServer 是唯一能接收写入的模块而且是单点，由于内存有限，随着增量数据的不断增加，需要将内存表中的数据写到 ChunkServer 的固态盘中，这个过程叫做每日合并[53]。

UpdateServer 收到 RootServer 的合并命令后，首先，冻结当前活跃的内存表，生成冻结内存表（Frozen Memtable），并开启新的活跃内存表，接受后续的写入操作。然后，通知 RootServer 数据版本发生变化，RootServer 收到消息后通知

ChunkServer 执行定期合并的操作，ChunkServer 将本地的基线数据与冻结内存表中的增量数据融合生成新的基线数据。

在每日合并的过程中，系统的内存，带宽，CPU 等资源被大量占用，会影响系统的服务能力，因此，每日合并过程往往选在系统负载很小的时间进行，一般选在凌晨 1 点左右。

3.2.6 OceanBase 基线数据索引结构

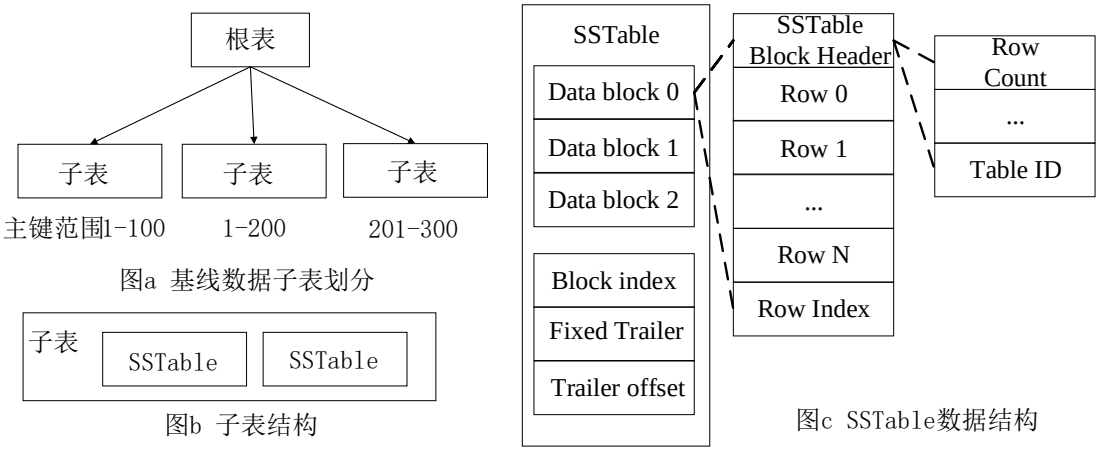

图 3.5 OceanBase 索引结构示意图

在 OceanBase 中，在 ChunkServer 上基线数据按主键排序划分为数据量大致相等的数据范围，称为子表，每个子表由一个或者多个 SSTable 构成，每个 SSTable 存放多个数据块, 如图 3.5 所示。

可以看出，OceanBase 为一个三级的索引结构，分为子表索引，块索引以及行索引，下面举例示意 OceanBase 使用主键索引读取数据的流程：

select * from table1 where id = 1；

首先，MergeServer 会根据解析 SQL 语句，提供主键值去根表查找子表位置信息，找到对应的 ChunkServer。然后，在 ChunkServer 上根据主键找到对应的子表以及对应的 SSTable。在查找 SSTable 时，首先读取 SSTable 中 Trailer Offset 的信息，将 Block index 加载到内存，接着根据 Block Index 定位到主键行所在的

Block，并将该 Block 加载到内存，然后再根据 Row Index 找到主键对应的行。

ChunkServer 支持两种读取方式，一种为 Get 方法，一种为 Scan 方法，使用 Get 方法读取数据时根据主键读取对应的行，使用 Scan 方法则根据主键范围获取数据行， Get 方法虽然每次只获取一行，但是每次读取一个 Block 的数据，OceanBase 在内存中维护了块缓存，在查询的过程中先查询块缓存，如果没有找到则进行一次磁盘 IO 获取数据，块缓存的使用减少查询开销。

3.3 OceanBase 聚合运算的流程

OceanBase 聚合运算的流程大致如图 3.6 所示：

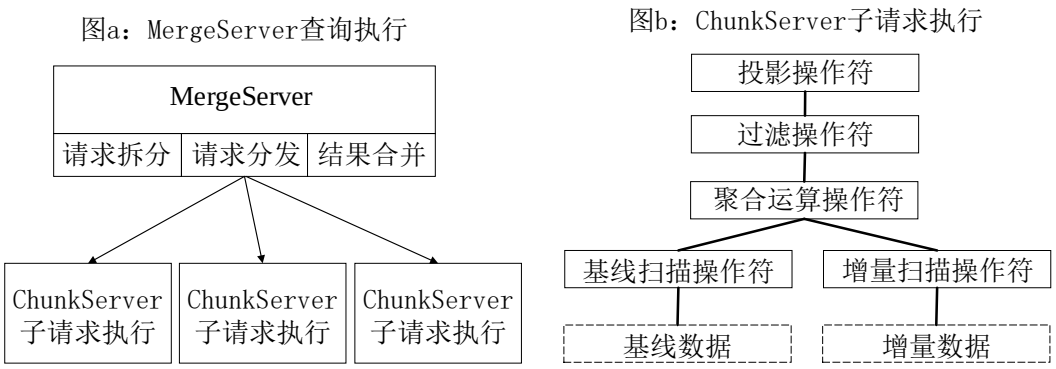

图 3.6 聚合运算执行流程

(1) 如图 3.6-a 所示，MergeServer 收到客户端查询请求，解析 SQL 语句，生成查询请求，如果请求的数据只分布在一个 ChunkServer 上，则将该请求发到该 ChunkServer 去执行，如果请求的数据分布在多台 ChunkServer 上，则将请求拆分成多个子请求发到对应的 ChunkServer 上。

(2) 如图 3.6-b 所示，ChunkServer 在收到查询请求时，构造物理计划，依次生成投影操作符，聚合运算操作符，融合操作符，以及基线扫描操作符。当操作符打开时，基线扫描操作符会扫描本地磁盘的基线数据，融合操作符则会向 UpdateServer 请求增量数据并将增量数据和本地的基线数据融合，聚合运算操作则将融合的结果排序，分组并计算聚合值，最后投影操作符向上返回执行结果。

(3) 最后，MergeServer 则将 ChunkServer 发来的部分结果合并，然后返回给客户端。

从上述的例子可以看出，OceanBase 由于增量数据和基线数据分离的特点，聚合函数的性能比较低下，首先，系统要满足实时查询的需求，必须先将增量数据和基线数据融合再做聚合运算，这个过程涉及到节点的网络交互，会占用大量的响应时间。其次，聚合运算涉及全表扫描，排序，聚合等操作，磁盘开销就是一个很大的瓶颈。因此，聚合运算的优化应该从减少磁盘开销和网络开销两个方面进行。

3.4 本章小结

本章首先介绍了 OceanBase 产生的应用背景，接着详细介绍了 OceanBase 的架构，各个模块的功能，读写流程以及基线数据的索引结构。最后介绍了 OceanBase 聚合运算的流程，指出聚合运算的缺陷并给出优化方向。

第四章 基于 LocalCube 的聚合运算实现

4.1 优化方案

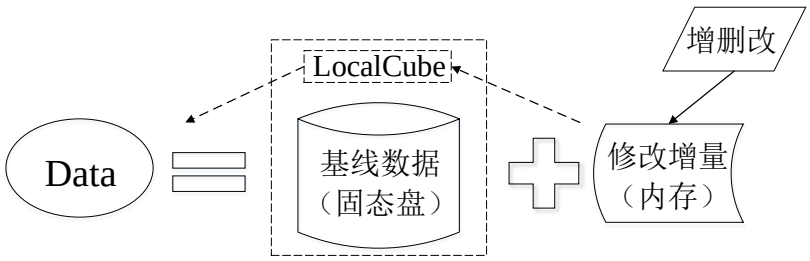

图 4.1　聚合优化实现示意图

在 OceanBase 独特的架构下，基线数据和增量数据存储分离，在读取的过程中需要融合这两部分数据，涉及大量网络传输、磁盘操作，开销很大。由于基线数据更新缓慢(一般周期性)，因此可以在基线数据端先将聚合结果缓存在基线服务器上，在读取的过程中直接融合增量数据和基线聚合结果。这样做的目的是避免基线数据端的重计算，加快读取流程，见图 4.1 所示。

4.2 LocalCube 的定义

LocalCube 实现示意见图 4.1。在 Cache 中它存储了基线数据聚合运算结果。Cache 数据结构本质上是一张二维表，支持表的基本运算，如投影，排序，连接等。

4.2.1 LocalCube 的结构

表 4.1　LocalCube 数据结构

分组属性	聚合属性 1	…	聚合属性 n	主键范围
Group 1				
…				
Group n				

表 4.1 为 LocalCube 的数据结构示意图：第一列为分组属性（Dimension），其余的列为聚合属性（Measurement），主键范围记录对应分组的主键范围。

下面通过一个例子详述 LocalCube 的结构。表 4.2 为某小学一年级学生的成绩表 st_grade，记录一年级学生的语文和数学考试成绩。

表 4.2 某小学一年级学生成绩表 st_grade

Student_no	Chinese	Math	Class
100010	82	80	1
100011	84	90	2
100012	86	97	2
100013	87	92	3
100014	81	91	3

现查询一年级每个班的语文平均分，数学最高分以及每个班的人数。

Select class,count(*),avg(chinese),max(math) from st_grade group by class;

表 4.3 LocalCube 存放结果

Class	Count(*)	Avg(chinese)	Max(math)	Rowkey_Range
1	1	82	90	100010-10010
2	2	85	92	100011-10012
3	2	84	94	100013-10014

LocalCube 保存了上述查询的结果，如表 4.3 所示，Rowkey_Range 为每组对应的主键范围。

4.2.2 LocalCube 的创建

LocalCube 为基线数据的聚合结果，OceanBase 原有流程中的查询不仅读取增量数据而且读取基线数据。在创建 LocalCube 的过程中，要让系统只读取基线数据。因此，我们必须对 OceanBase 原有的读操作流程进行修改。

在 sql 语句中使用 hint，强制让系统只读基线数据并将计算结果存储到 ChunkServer 的 LocalCube 中，hint 命令为/*+read_static_and_create_localcube*/。

以上述查询为例，创建 LocalCube 的命令为：

Select/*+read_static_and_create_localcube*/class,count(*),avg(chinese),max(math) from st_grade group by class;

LocalCube 的创建流程如下，见图 4.2：

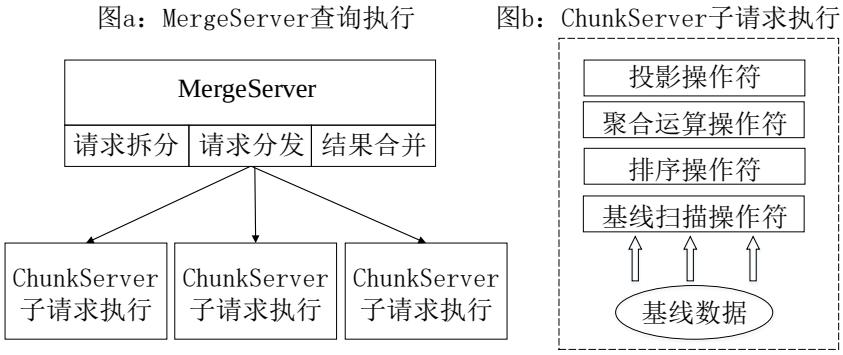

图 4.2 创建 LocalCube 流程

(1) 构建查询计划：MergeServer 在接受到客户端的请求后经过词法解析，语法解析，生成逻辑，物理计划。

(2) 查询计划分发：MergeServer 根据数据分布将查询计划分为多个，发到对应的 ChunkServer 上。

(3) ChunkServer 构建操作符：ChunkServer 会构造上述操作符，如图 4.2 所示，在构建操作符的过程中，会根据 read_static_and_add_localcube 语义只生成基线扫描操作符，读取基线数据。

(4) 操作符打开：最上层的操作符打开，接着它的孩子操作符依次打开。然后，投影操作符调用 get_next_row 接口，驱动底层的操作符调用 get_next_row 去获取数据。

(5) 获取数据：基线扫描操作符在 get_next_row 的过程中读取本地的基线数据；排序操作符则将基线操作符传来的数据根据分组属性排序；聚合操作符将排序操作符传来的数据分组、聚合，并统计各个分组的主键范围；

(6) 投影操作符将聚合运算操作符的数据存放到本地 ChunkServer 的 LocalCube 中，完成 LocalCube 的创建。

在一台 ChunkServer 上可以缓存多个 LocalCube。为了方便查找， LocalCube 存放在 LocalCubeManager 中。LocalCubeManager 维护了一个 Map 数据结构，其中 key 为 LocalCube 的元信息，记录了它的 Id，字段的名称，个数和类型。为了方便查找，LocalCube 的 Id 和对应原始表的 TableId 保持一致。其中 value 为 LocalCube 对象，存放基线数据聚合结果。

4.2.3 LocalCube 的维护

(一) LocalCube 的批量重建

LocalCube 是基线数据的统计值，如果基线数据不发生变化，那么 LocalCube 也不发生变化。基线数据只有在每日合并完成后会更新，因此 LocalCube 的更新选取在每日合并之后。

具体流程如下：

(1) 在 RootServer 上有一个定时任务 check_tablet_version，会不断检测数据版本的变化判断每日合并是否完成。如果合并完成，RootServer 会向 MergeServer 发送一个心跳包 Ob_LocalCube_BatchCreation_Command。

(2) MergeServer 收到 RootServer 发来的心跳包后，会触发 LocalCube 批量重建命令，将用户创建 LocalCube 时输入的 Sql 重新执行，完成重建。

(二) LocalCube 的释放

LocalCube 的释放发生在两个过程中，第一，每日合并开始前，LocalCube 被释放，它会在每日合并完成后再重新建立；第二，删除表时，系统会自动释放对应表的 LocalCube。

每日合并前释放 LocalCube 的流程如下：

(1) RootServer 在开始每日合并前，发送一个心跳包 Release_LocalCube_Command 给 ChunkServer。

(2) ChunkServer 收到心跳包后，将本地 LocalCubeManager 中的 LocalCube 全部释放。

LocalCube 为基础表的缓存，当基础表被删除时，对应的 LocalCube 也需要

被释放，具体流程如下：

(1) MergeServer 接受到用户的删表请求，将物理计划发给 ChunkServer，

(2) ChunkServer 在接受到数据包后，先删除物理表的 schema 信息，然后通过表名查找到对应的 LocalCube，将 LocalCube 释放，然后返回给客户端。

4.2.4 LocalCube 的选择

LocalCube 为基线数据的聚合结果，若一张大表的基线数据分布在多台 ChunkServer 上，则该表的 LocalCube 也分布在对应的多台 ChunkServer 上。每台 ChunkServer 会先查看本地的缓存，然后再选择对应的方案。LocalCube 选择算法见算法 1。

算法 1 ChunkServer 上 LocalCube 的选择算法

输入：查询语句

1: ChunkServer 从 MergeServer 发来的数据包（查询语句）中解析出查询参数（SqlParam），包括 TableId，分组列和聚合列的集合；
2: **if** 在 LocalCubeManager 中找到了 id 为 TableID 的 LocalCube **and**
3: LocalCube 中的分组列和聚合列集合包含了查询的分组列和聚合列集合 **then**
使用 LocalCube；
4: **else**
5: 走 OceanBase 原有的读取流程；

可以看出，如果 LocalCube 失效，系统则会走 OceanBase 原有的流程，见算法第 5 步，忽视聚合查询优化设计，但是不会影响结果的正确性。

4.2.5 LocalCube 存储

(1) LocalCube 存储数据

在 OceanBase 架构中，UpdateServer 记录了一段时间的更新数据，基线数据按主键排序划分存储在多台 ChunkServer 上，它在每日合并之前保持不变。OceanBase 中的 ChunkServer 相当于一个分布式的数据仓库，LocalCube 存放基

线数据的聚合运算结果。

(2) LocalCube 存储代价

LocalCube 保存了基线数据聚合运算的结果，它的大小为基线数据聚合运算结果集的大小。Measurement 为聚合列的个数，通常由用户的分析需求决定。分组选择率是在经过聚合运算后结果集记录数目和原表记录数目的比值，它由负载特征决定。

分组选择率越大表明 group by 后的分组越多，在极端情况下和原表的个数相等；反之则分组越少，在极端情况下为一个分组。以 TPC-D 测试基准的数据集为例，100GB 的数据，聚合运算的结果集数量从 2 个元组到 1400000 个元组不等，可见不同负载分组选择率的差距。

LocalCube 所占用的空间跟选取的聚合列(Measurement)的个数和分组选择率正相关，聚合列(Measurement)选取的越多，分组选择率越高，则 LocalCube 所占用的空间越大。

若 st_grade 表的大小为 100GB，需要分析每班人数，语文成绩平均分，数学最高分。聚合结果集数目和 TPC-D 测试基准保存一致，则 LocalCube 的存储代价为 120Byte 到 84MB 之间，其中，class 字段占 4 Byte，count 字段占 4 Byte，avg 字段占 8 Byte，max 字段占 4 Byte，主键范围约占 40Byte。

通过计算得知，100GB 的数据，存放 5 列值，LocalCube 的占用空间最大为 84MB，存储空间占用对单台服务器来说压力很小。若分组的数量维持在一个较小的水平，比如 1000 组，则存储代价会非常小。而且在 OceanBase 数据库中 ChunkServer 的数目可以横向扩展，因此，LocalCube 常驻 ChunkServer 内存是可以接受的。

4.3 基于 LocalCube 的查询

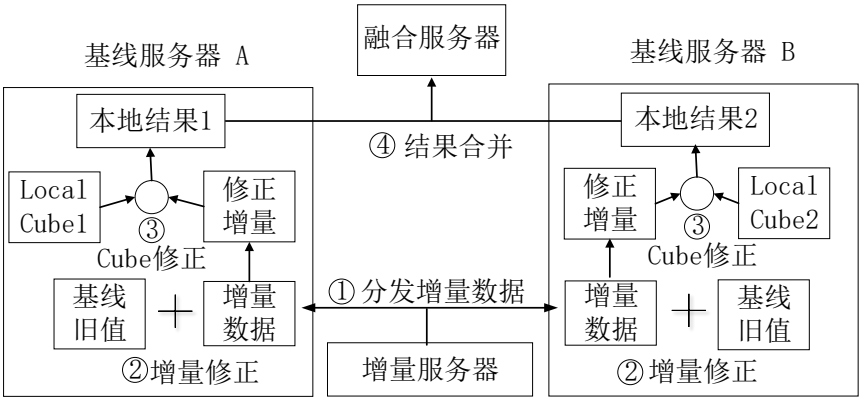

图 4.3 基于 LocalCube 的查询流程

图 4.3 为基于 LocalCube 的查询，分为四个步骤：

1. UpdateServer 增量数据区分
2. 获取本地基线原值
3. 本地 LocalCube 和修正增量归并融合
4. 全局结果合并

4.3.1 UpdateServer 增量数据区分

OceanBase 中对 Memtable 的修改采用批量更新的方式，即将一段时间的写操作记录在 UpdateServer 的内存表中，不区分对基线数据的更新和新插入的数据。

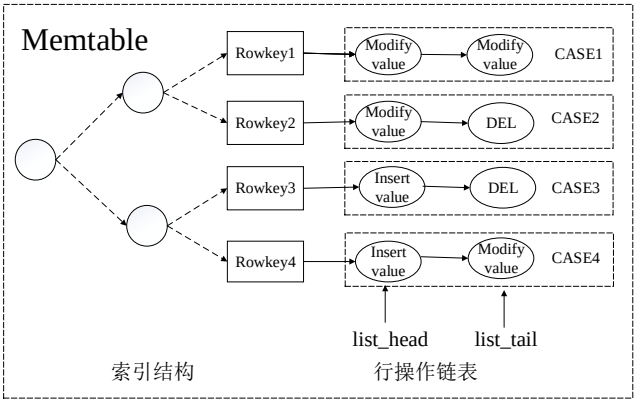

图 4.4 Memtable 内存结构

如图 4.4 所示，Memtable 包括索引结构及行操作链表两部分，索引结构为一颗高性能的 B+树，叶子节点对应 Memtable 的一行数据，Rowkey 为行主键，value 为行操作链表的指针，指向行操作链表。行操作链表记录了对每行的操作，它的每一个元素都是一个 cell，记录对该行的修改。更新某行只在行操作链表的尾部加入一个 cell，记录修改后的值，并不记录原值，删除某行只是在行操作链表的末尾加一个 DEL 的标记，并不是真的删除索引结构和行操作链表里的内容。

为了在查询中使用 LocalCube，通过增量数据修正 LocalCube 来获得最终的结果，必须将增量数据区分为基线修改数据和新行数据。如果是对基线数据的更新操作，则需要找到基线数据的原值，LocalCube 减去原值，加上新值；如果是对基线数据的删除操作，LocalCube 减去原值。如果是新插入的数据，则直接融合到 LocalCube 中。因此，我们自定义了以下的数据类型和六种操作：

定义 4.3.1 新行：新插入的数据，该主键在基线数据中并不存在，保存在 UpdateServer 的 Memtable 中，是完整的一行数据。

定义 4.3.2 基线更新行：基线数据的 update 操作，保存了修改后的值，存在 UpdateServer 的 Memtable 中。

定义 4.3.3 基线删除行：对基线数据删除，在 OceanBase 中为一个 DEL 标记，保存在 UpdateServer 的 Memtable 中。

定义 4.3.4 六种操作：四种针对基线数据的操作，包括基线基线更新操作（static_update），删除操作（static_delete），空操作（null）和基线插入操作（static_insert）。两种针对 LocalCube 的操作，Op_Insert 和 Op_Delete，如图 4.5 所示。

```
enum  RowType{
  STATIC_UPDATE = 0;
  STATIC_DELETE = 1;
  STATIC_NULL = 2;
  STATIC_INSERT = 3;
}
```

```
enum OperationType{
  OP_INSERT = 0;
  OP_DELETE = 1;
}
```

图 4.5 自定义的六种操作

例如，表 st_grade 的数据存放在 ChunkServer 上，有四行修改，学号为 100010 的记录语文成绩更新为 88 分，数学成绩更新为 90 分，则行操作链表中将保存 2 个 cell，为别<Chinese，88>，<math，90>；学号为 100011 的记录被删除，则行操作链表中将保存一个 cell，为<del，*>；学号为 100016 的记录先插入接着被删除，则行操作链表中保存 2 个 cell，分别为<Chinese,80，math ,90，class ,4>，<del，*>；学号为 100017 的记录被插入，则行操作链表保存一个 cell，为<Chinese,82，math,93，class,4>。

上述四位同学的数据更新情况分别对应 Memtable 中的四种情况，见表 4.4。

表 4.4 基线操作对应表

Memtable 中类别	是否为新行	尾部操作类型	操作效果	基线操作
CASE 1	否	值	基线数据行（原行）被修改	Static_Update
CASE 2	否	DEL	基线数据行（原行）被删除	Static_Delete
CASE 3	是	DEL	新增一行最后被删除	Static_Null
CASE 4	是	值	新增一行被修改	Static_Insert

4.3.2 OceanBase 中的实现

OceanBase 中，TEValue 为管理 B+树中的行操作链表的一个数据结构，它和 B+中的 Rowkey 一一对应，头指针 list_head 指向链表的第一个修改操作，尾指针 list_tail 指向链表的最后一个修改操作。Memtable 中对应的行操作链表中的 cell 分为两类，一类是值，记录修改后的结果；一类是删除标记 DEL。

修改如下：

(1) 为了区分 Memtable 中的新行和旧行，在 TEValue 中加入一个布尔值 is_new_row，默认为 false。在修改 Memtable 构建 TEvalue 的时候，判断本次修改的语义，如果是 insert 操作，表明是新增一行，则把 is_new_row 置为 true。

(2) 在查询 Memtable 阶段，首先根据 Rowkey 找到对应的 TEValue，然后判断 TEValue 中 is_new_row 的类型，再判断行操作链尾部的 cell 信息，即可区分自定义的四种语义，将对应的语义记录在对应的行中。

4.3.3 增量数据转换

算法 2 介绍了增量数据转换为 LocalCube 修正数据的过程，为下一步融合 LocalCube 做准备。在 UpdateServer 上传过来的数据中，基线修改操作只记录了新值，基线删除操作只记录 DEL 标记，因此，我们需要根据增量数据的主键去 ChunkServer 找到对应的原行。这个过程需要调用 OceanBase 的 GET 方法，该操作要读取磁盘，是本算法的主要开销。使用 GET 方法获取行，每次读取一个 Block 的数据，并缓存在内存中，若缓存命中，则直接使用块缓存，如果读取的数据存放比较集中，则 GET 方法的效率会很高。

算法 2 增量数据转换为修正数据算法

输入： 增量数据 IncrementalData

输出： 修正增量数据 AmendData

```
1:  遍历增量数据的每一行，获取每行的 RowType；
2:      if 该行的 RowType = = STATIC_UPDATE then
3:          将该行的 OperationType 标记为 OP_INSERT；
4:          根据主键获取基线数据原行 GET(Rowkey)，将原行的 OperationType 标记为 OP_DELETE；
5:          将上述两行添加到 AmendData 中；
6:      else if 该行的 RowType= = STATIC_DELETE then
7:          根据主键获取基线数据原行 GET(Rowkey)，将原行的 OperationType 标记为 OP_DELETE，并添加到 AmendData 中；
8:      else if 该行的 RowType= = STATIC_INSERT then
9:          将该行的 OperationType 标记为 OP_INSERT 并添加到 AmendData 中；
10:     else
11:         do nothing;
```

4.3.4 本地数据融合

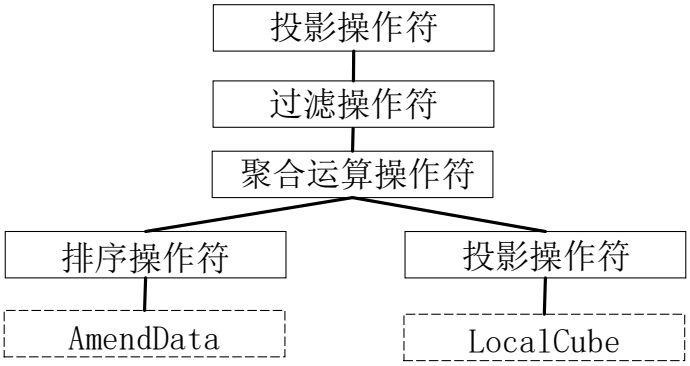

表 4.5 本地数据融合操作符示意图

如图 4.5 所示，ChunkServer 在判断能使用本地 LocalCube 后，会构造上述操作符，执行采用流水线的方式，具体过程如下：

(1) 顶层操作符打开，驱动下层操作符依次打开。

(2) 顶层操作符打开后调用 get_next_row 接口获取一行，依次驱动下层操作符调用 get_next_row。

(3) 排序操作符将 AmendData 按分组列由小到大排序，并向上返回一组的数据行。

(4) 投影操作符从 LocalCube 中取出需要的数据列，向上返回一行数据。

(5) 聚合操作符将排序操作符和投影操作符传上来的行数据融合，生成融合结果，并返回上一层的操作符。融合算法见下节。

(6) 过滤操作符会按过滤条件对聚合操作符传来的数据过滤。

(7) 投影操作符将过滤操作符传来的行投影，计算表达式，并发送给 MergeServer。

在相关工作中介绍了两种聚合运算类型，即分布性聚合运算，如 Sum，Count，Avg，和全局性聚合运算，如 Max 和 Min。两种聚合运算在融合阶段的算法稍有不同，下面分别介绍。

（一）分布性聚合运算融合算法

分布性聚合运算采用二路归并的方式进行融合，见算法 3，AmendData 中的

数据按照组排好序，LocalCube 本身有序。算法中的变量含义如下：result_row 和 result_group 分别代表 LocalResult 中当前行和当前行的组号，amend_row 和 amemd_group 分别代表 AmendData 中的一行和对应组号，cube_row 和 cube_group 分别代表 LocalCube 中的一行和对应组号。

算法 3 分布性聚合运算的二路归并融合算法----sum，count

输入：AmendData（增量修正数据），LocalCube（本地缓存）

输出：LocalResult（本地结果）

```
1:  取 AmendData 中的一组数据 AmendGroupRows，组为 amemd_group；
2:  取 LocalCube 中的一行数据 cube_row，组为 cube_group；
3:  while AmendData 不为空并且 LocalCube 不为空 do
4:    if amemd_group < cube_group Then
5:       将 AmendGroupRows 中所有记录按列对应相加，count 为记录条数，并添
     加到 LocalResult，取 AmendData 中的下一组数据；
6:    else if amemd_group > cube_group Then
7:       将 cube_row 添加到 LocalResult，取 LocalCube 中的下一条数据；
8:    else amemd_group = = cube_group Then
9:       遍历 AmendGroupRows 中所有记录；
10:          if 当前记录为 delete 操作 then
11:             把该条记录的列值从 cube_row 的 sum 中删除，count 值--；
12:          else
13:             将该条记录的列值加入到 cube_row 的 sum 中，count 值++；
14:       将 cube_row 添加到 LocalResult；
15: while AmendData 不为空 do
16:       取 AmendGroupRows 中的第一行记录添加到 LocalResult，取 AmendData 中
     的下一组数据；
17: while LocalCube 不为空 do
18:       将 cube_row 添加到 LocalResult，取 LocalCube 中的下一条数据；
```

由于 avg(col) = sum(col)/count(col)，因此在数据库中维护 sum(col)的值和 count(col)的值，将 avg 的查询转换为 sum 和 count 的查询。

（二）全局性聚合运算融合算法

全局性聚合运算的处理流程和分布性聚合运算的处理流程类似，全局性聚合运算在调用排序操作符时需要对 AmendData 进行三次排序：

首先，按分组属性由小到大排序。然后，在同一分组内按行的 OperationType 分为 insert 组和 delete 组。最后，分别在 insert 组和 delete 组内按聚合列排序。

Max 融合流程如算法 4 所示：

算法 4 全局性聚合运算的二路归并融合算法----max

输入：AmendData（增量数据），LocalCube（本地缓存）

输出：LocalResult（本地结果）

```
1:  取 AmendData 中的一组数据 AmendGroupRows，对应的组为 amemd_group；
2:  取 LocalCube 中的一行数据 cube_row，对应的组为 cube_group；
3:  while AmendData 不为空并且 LocalCube 不为空 do
4:    if amemd_group < cube_group Then
5:       取 AmendGroupRows 中的第一行记录添加到 LocalResult；
6:       取 AmendData 中的下一组数据；
7:    else if amemd_group > cube_group Then
8:       将 cube_row 添加到 LocalResult，取 LocalCube 中的下一条数据；
9:    else amemd_group 等于 cube_group Then
10:      if max_current < = max_insert Then
11:         将 cube_row 的 max_current 变更为 max_insert，添加到 LocalResult；
12:         取 AmendData 中的一组数据和 LocalCube 中的一行数据；
13:      else Then
14:         if max_current > max_delete Then //说明最大值没有被删除
15:            将 cube_row 添加到 LocalResult；
16:         else if max_current = max_delete   Then
17:            根据主键的范围，去扫描基线数据，重新计算最大值并添加到
       LocalResult；
18:         else return ERROR；
19: while AmendData 不为空 do
20:      取 AmendGroupRows 中的第一行记录添加到 LocalResult，取 AmendData 中
21: 的下一组数据；
```

22: **while** LocalCube 不为空 **do**

23: 将 cube_row 添加到 LocalResult，取 LocalCube 中的下一条数据；

在计算 Max 时分别在 insert 组和 delete 组内按聚合列由大到小排序，并记录 insert 组内聚合列的最大值 max_insert 和 delete 组内的最大值 max_delete。max_current 为 LocalCube 中维护的基线数据的最大值。

Min 融合流程和 Max 融合流程相似，在计算 Min 时则分别在 insert 组和 delete 组内按聚合列由小到大排序，并记录 insert 组内聚合列的最小值 min_insert 和 delete 组内的最小值 min_delete。min_current 为 LocalCube 中维护的基线数据的最小值。

4.3.5 算法正确性证明

算法将增量数据转换化为对 LocalCube 的删除和插入(AmendData)。其中分布性聚合运算(Sum、Count、Avg)对插入操作和删除操作是增量可计算的，因此算法可行。全局性聚合运算(Max、Min)对删除操作不是增量可计算的，全局性聚合运算的正确性证明如下：

(1) 数据都是按组排序的，在同一组内，删除是对基线数据的删除，而基线数据的最大/小值就在 LocalCube 中，因此，删除操作的最大/小值肯定小于等于/大于等于 LocalCube 中的最大/小值。

(2) 在同一个分组内，如果新插入数据的最大/小值比 LocalCube 中的最大/小值大/小，则更新该最大值，否则，判断最大/小值有没有被删除，如果没有，则结果不变，如果有，则根据主键范围扫描重新计算。

4.3.6 RowkeyRange 对算法的影响

LocalCube 的最后一列存储的是每个 group 对应的主键范围 RowkeyRange。在计算全局性聚合运算时(Max 和 Min)，当最大值/最小值被删除时，能根据这个主键范围查找该组内基线数据的值，防止对全表进行扫描，当每个组的 RowkeyRang 划分的较开时，扫描的数据量会比较小，否则，Max 和 Min 的算法处理效果不好。

4.3.7 基于 LocalCube 聚合运算示例

下面结合一个例子来说明基于 LocalCube 的查询流程。若 St_grade 表按主键范围存储在两台 ChunkServer 上，ChunkServer1 存储的主键范围为[Min,100012]，ChunkServer2 为[100013,Max]，见表 4.6，4.7。

表 4.6 ChunkServer1 存储的基线数据

Student_No	Chinese	Math	Class
100010	82	80	1
100011	84	90	2
100012	86	97	2

表 4.7 ChunkServer2 存储的基线数据

Student_No	Chinese	Math	Class
100013	87	92	3
100014	81	91	3

(1) 生成 LocalCube

select/*+read_static_and_create_localcube*/class,count(*),sum(chinese),avg(chines) from st_grade group by class;

ChunkServer1 上的 LocalCube 为 LocalCube1，ChunkServer1 上的 LocalCube 为 LocalCube2，如表 4.8，4.9 所示：

表 4.8 LocalCube1 结构示意

Class	Count(*)	Sum(chinese)	Avg(chinese)	Rowkey Range
1	1	82	82	100010-10010
2	2	170	85	100011-100012

表 4.9 LocalCube2 结构示意

Class	Count(*)	Sum(chinese)	Avg(chinese)	Rowkey Range
3	2	168	84	100013-100014

(2) UpdateServer 增量数据区分

UpdateServer 有以下更新，主键 110010 被删除 <del，*>， 主键 110011 被修改<chinese，88>，<math,90>，<class,1>，主键 110015 先被插入紧接着被删除<del,*>，主键 110016 被插入<Chinese,82，math,93，class,4>。则 UpdateServer 的增量数据如表 4.10 所示，按主键有序排列。

表 4.10 UpdateServer 增量数据

Student_No	Chinese	Math	Class	**RowType**
110010	NULL	NULL	NULL	**Static_delete**
110011	88	90	1	**Static_update**
110015	NULL	NULL	NULL	**null**
110016	82	93	4	**Static_insert**

(3) 查询一年级每个班的语文平均分

select class,avg (chinese) from st_grade group by class;

MergeServer 将解析的物理计划发送到 ChunkServer1 和 ChunkServer2，ChunkServer1 和 ChunkServer2 向 UpdateServer 请求对应主键范围的增量数据，UpdateServer 收到消息后将增量数据发送到两台 ChunkServer 上。ChunkServer1 和 ChunkServer2 收到的更新数据如表 4.11 和 4.12 所示：

表 4.11 ChunkServer1 增量数据范围

Student_No	Chinese	Class	RowType
110010	NULL	NULL	Static_delete
110011	88	1	Static_update

表 4.12 ChunkServer2 增量数据范围

Student_No	Chinese	Class	RowType
110015	NULL	NULL	null
110016	82	4	Static_insert

(4) 增量数据转换为 AmendData

ChunkServer1 根据主键调用基线数据的 GET（110010，110011）方法找到每个主键对应的原始行，将增量数据转化为 AmendData，并按 class 属性排序，如表 4.13 所示。

表 4.13 CS1 的增量修正数据 AmendData

Student_No	Class	Chinese	OperationType
110010	1	82	Delete
110011	1	88	Insert
110011	2	84	Delete

表 4.14 CS1 本地聚合结果

Class	Sum(Chinese)	Count(*)
1	88	1
2	84	1

(5) 本地数据融合

ChunkServer1 将本地的 AmendData 和 LocalCube 二路归并融合，生成本地结果，如表 4.14 所示。ChunkServer2 执行相同的流程，将本地的增量修正数据 AmendData 和 LocalCube 融合，生成本地结果，如表 4.15 所示。

表 4.15 ChunkServer2 本地聚合结果

Class	Sum(Chinese)	Count(*)
3	168	2
4	82	1

表 4.16 全局结果

Class	avg(Chinese)
1	88
2	84
3	84
4	82

(6) 全局结果合并

MergeServer 将 ChunkServer1 和 ChunkServer2 的结果合并，如表 4.16 所示。

4.4 LocalCube 的效果分析

4.4.1 创建所需时间

创建 LocalCube 所需的时间为基线数据生成聚合结果的时间，该过程在 ChunkServer 上执行，只有本地开销，没有节点数据传输。创建时间与基表的大小和 sql 的复杂度有关，若涉及的基本表越大，聚合的 sql 越复杂，则生成 LocalCube 的时间越长。LocalCube 重建请求一般在每日合并之后，每日合并一般在凌晨，因此，重建的时间不会影响用户的查询。

4.4.2 缓存失效

每日合并期间缓存失效，理由如下：

首先，每日合并期间基线数据版本发生变化。LocalCube 不能对应当前的基线数据，LocalCube 失效，只有当每日合并完成后，系统会生成新版本的基线数据。其次，为了减少对每日合并的影响。每日合并期间，ChunkServer 会拉取 UpdateServer 上的增量数据到本地，生成新的基线数据，这个过程会有大量的磁盘，内存，网络的开销。为了减少对每日合并的影响，在触发每日合并时，系统首先会释放 ChunkServer 上的缓存 LocalCube，减少内存占用。

4.5 基于 LocalCube 算法效率分析

OceanBase 原有流程中，聚合运算的时间代价主要包含四个部分：UpdateServer 扫描 B+树获取增量数据的代价，增量数据传输给 ChunkServer 的代价，ChunkServer 上基线数据和增量数据融合代价以及排序，分组计算的代价。

- 设扫描每条增量数据的代价为 m_{incre}，增量数据为 m 条，增量数据 m 分为 m_{NewRow} 和 $m_{StaticModifyRow}$，则扫描代价为 m_{incre}。
- 设传输每条增量的数据花费 T_{trans}，网络传输代价为 mT_{trans};
- 设扫描每条基线数据的代价为 T_{static}；融合一条基线数据和增量数据的代价为 T_{fuse}；设基线数据的数目为 n，则融合代价为 $nT_{static} + (m+n)T_{fuse}$；
- 设每条数据排序分组计算的代价为 $T_{sort\text{-}group}$，则排序分组计算的代价为 $(m+n)\ T_{sort\text{-}group}$。
- 设 LocalCube 中组的个数为 G_{LC}。

则 OceanBase 原有聚合运算的代价 OBCost 为 $m(T_{incre}+T_{trans}) + nT_{static} + (m+n)T_{fuse} + (m+n)\ T_{sort\text{-}group}$；

基于 LocalCube 中聚合运算的的代价主要包含五个部分，UpdateServer 扫描 B+树获取增量数据的代价，增量数据传输给 ChunkServer 的代价，增量数据转换的代价(增量数据获取原值代价)，AmendData 排序分组的代价以及修正

LocalCube 的代价。

- 其中前两部分代价和 OceanBase 原有代价一样，为 $m(T_{incre}+T_{trans})$；
- 只有基线修改数据才需要从磁盘扫描，则增量数据获取原值的代价为 $m_{StaticModifyRow}\ T_{static}$。
- 如果增量数据中 Static_Update 会转换两条数据，会使 AmendData 数据行增多，如果增量数据中没有 Static_Update，则 AmendData 的数据量为 m；如果增量数据的旧行操作全部为 Static_Update 操作，则 AmendData 的数据量为 $m+m_{StaticModifyRow}$。则 AmendData 排序分组的代价为 $mT_{sort-group}$ 到$(m+m_{StaticModifyRow})T_{sort-group}$ 之间。
- AmendData 和 LocalCube 采用二路归并的方式融合，LocalCube 中组的行数和 AmendData 中行数融合。则修正 LocalCube 的代价为$(G_{LC}+[m, m+m_{StaticModifyRow}])T_{fuse}$。

基于 LocalCube 方案最小的代价为 LowerCost：$m(T_{incre}+T_{trans}) + m_{StaticModifyRow}\ T_{static} + m\ T_{sort-group} + (G_{LC}+m)T_{fuse}$

基于 LocalCube 方案最大的代价 UpperCost：$m(T_{incre}+T_{trans}) + m_{StaticModifyRow}\ T_{static} + (m+m_{StaticModifyRow})T_{sort-group} + (G_{LC}+m+m_{oldRow})T_{fuse}$

则基于 LocalCube 的方案性能提升的时间至少为：$OBCost–UpperCost = (n–m_{StaticModifyRow})T_{static} + (n–G–m–m_{StaticModifyRow})T_{fuse} + (n–m–m_{StaticModifyRow})T_{sort-group}$。

其中 n 为基线数据的行数，m 为增量数据的行数，可以看出如果 n>>m 时，性能提高非常明显，这也符合 OceanBase 的架构设计，即修改较少，基线数据很大。而且，当基线数据修改越少，即 $m_{StaticModifyRow}$ 越小，修正缓存的操作越少，性能越高，这符合增量维护算法的特点。

4.6 本章小结

本章首先提出优化聚合运算的方案，接着介绍了 LocalCube 的数据结构，它在本质上是一张表，缓存了基线数据的聚合结果。然后详细的讨论了 LocalCube

的存储代价，LocalCube 的创建，LocalCube 的重建，LocalCube 的释放以及 LocalCube 的选择。

在第三节讨论了基于 LocalCube 的查询，分为四步，第一步为 UpdateServer 增量数据区分，第二步为获取基线数据原值，第三步为本地数据融合，第四部为全局数据融合。接着证明了算法的正确性，并讨论了 RowKeyRange 对算法的影响。最后通过一个示例来演示基于 LocalCube 算法的查询流程。在第四节分析了 LocalCube 的使用代价，包括创建时间以及有效时间。在第五节给出了基于 LocalCube 算法的效率分析，从理论上证明了该方法的有效性。特别适合增量数据小，基线数据很大的场景，这和 OceanBase 的架构设计是一致的。

第五章 基于 LocalCube 的聚合运算优化

5.1 优化思想

在第四章的方案中，通过构造 LocalCube 来优化查询，在查询的过程中，UpdateServer 将增量数据先发送到 ChunkServer，ChunkServer 融合增量数据和 LocalCube，这种方案能极大的减少 ChunkServer 的计算开销，但是如果 UpdateServer 上的增量数据比较大，那么网络开销就会比较大。

为了进一步减少网络传输代价，优化聚合运算的查询性能，在 UpdateServer 发送增量数据前，先将增量数据中的新行数据先聚合，减少数据传输，再将新行聚合结果和基线修改数据发送到 ChunkServer 上和 LocalCube 融合，如图 5.1 所示：

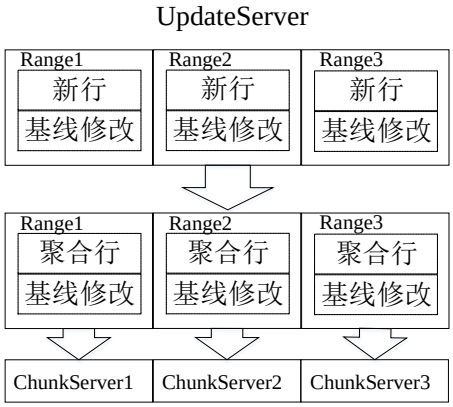

图 5.1　UpdateServer 聚合示意图

5.2 基于 LocalCube 查询改进算法

算法的第一阶段：增量聚合

ChunkServer 将本地生成的物理计划发到 UpdateServer，UpdateServer 对新行聚合。

算法的第二阶段：增量转换

ChunkServer 获取基线修改的原值，并将基线修改转换成 insert 操作和 delete 操作，其中 update 可以看成 delete 原值和 insert 新值。

算法的第三阶段：修正 LocalCube

insert 操作和 delete 操作对 LocalCube 进行修正。

算法的第四阶段：结果合并

MergeServer 合并 ChunkServer 发来的部分结果，并返回给客户端。

5.2.1 新行聚合

查询流程还是按照 OceanBase 原有的流程。首先，MergeServer 将查询请求发到对应的 ChunkServer；然后，ChunkServer 向 UpdateServer 请求增量数据。在这个过程中，将本地生成的物理计划携带，UpdateServer 在收到 ChunkServer 的请求后，会扫描 B+树上的增量数据，将新行数据聚合形成聚合行，连同基线修改数据一起发到对应的 ChunkServer 上，大致分为两个步骤：

(1) 数据区分

在扫描 B ＋树的过程中，判读 TEvalue 中 is_new_row 的值，并判断行操作链表的尾部类型，如果为表 5.1 中的 Case4，则表明新增一行，并把该行数据加入到 NewRowData，否则，将该行放到 StaticModifyData 中。

(2) 物理计划执行

在数据全部扫描完毕之后，Updateserver 执行物理计划，将 NewRowData 作为底层的基本表，生成聚合结果存放到 UPSAggregationData，连同 StaticModifyData 发送到对应的 Chunkserver 上，如图 5.2 所示：

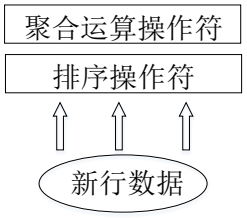

图 5.2 Updateserver 执行计划操作符

5.2.2 融合 LocalCube

该过程和第四章的融合过程一样，将 AmendData(增量修正数据)经过分组，排序等操作后和 LocalCube 融合，如图 5.3 所示：

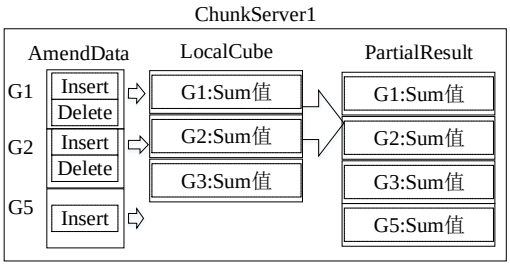

图 5.3 ChunkServer 融合过程示意图

5.2.3 算法效率分析

定义 5.2.3 UPS 选择率 $UPS_{Selectivity}$：经过 UpdateServer 聚合后，结果集元组数目和输入元组数目的比值。

改进流程中聚合运算的代价主要包含六个部分，UpdateServer 扫描 B+树获取增量数据的代价，新行聚合代价，增量数据传输给 ChunkServer 的代价，增量数据获取原值的代价，AmendData 排序分组的代价以及修正 LocalCube 的代价。

设聚合每条数据的代价为 T_{Agg}。增量数据 m 分为 m_{newRow} 和 m_{oldRow}。

- 则 UpdateServer 扫描代价为 mT_{incre}。
- UpdateServer 聚合代价为 $m_{newRow}T_{Agg}$。
- 传输代价为$(m_{newRow}UPS_{Selectivity}+m_{oldRow})T_{trans}$。
- 增量数据获取原值的代价为 $m_{oldRow}\ T_{static}$。
- AmendData 排序分组的代价为$(m_{oldRow}+m_{newRow}UPS_{Selectivity})T_{sort\text{-}group}$ 到$(2m_{oldRow}+m_{newRow}UPS_{Selectivity})T_{sort\text{-}group}$之间。
- 修正 LocalCube 的代价为$(G_{LC}+[m_{oldRow}+m_{newRow}UPS_{Selectivity},2m_{oldRow}+m_{newRow}UPS_{Selectivity}])T_{fuse}$。

则 $Imp_{LowCost} = mT_{incre}+m_{newRow}T_{Agg}+(m_{newRow}UPS_{Selectivity}+m_{oldRow})T_{trans}+m_{oldRow}\ T_{static}+(m_{oldRow}+m_{newRow}UPS_{Selectivity})T_{sort\text{-}group}+(G_{LC}+m_{oldRow}+m_{newRow}UPS_{Selectivity})T_{fuse}$。

$Imp_{UpperCost}=mT_{incre}+m_{newRow}T_{Agg}+(m_{newRow}UPS_{Selectivity}+m_{oldRow})T_{trans}+m_{oldRow}T_{stati}$
$_{c}+(2m_{oldRow}+m_{newRow}UPS_{Selectivity})T_{sort\text{-}group}+(G_{LC}+2m_{oldRow}+ m_{newRow}UPS_{Selectivity})T_{fuse}$。

则和第四章的方案相比，性能提高为 $UpperCost - Imp_{UpperCost} = m_{newRow}(1-UPS_{Selectivity}) T_{trans} + m_{newRow}(1-UPS_{Selectivity})(T_{sort\text{-}group}+T_{fuse}) - m_{newRow}T_{Agg}$

可以看出如果新行的分组选择率越低，聚合开销越小，性能提高越多，这也符合我们的假设，即新增数据聚合后能极大的减少数据，最极端的例子为计算 count，最后只返回一个值。

5.3 LocalCube 增量更新

在前面的设计方案中，LocalCube 在每日合并开始时释放，每日合并完成后重新生成，这导致的结果是每日合并开始到 LocalCube 重新生成的期间里 LocalCube 是失效的，这会影响用户在每日合并期间的查询，而且重新计算的方式涉及大量的计算，系统的查询性能会受到影响。

因此我们提出 LocalCube 增量更新的方案。从第四章可知，查询的过程就是增量数据修正缓存的过程，得到的结果是查询开始时系统最新数据的聚合结果，而 LocalCube 是每日合并前基线数据的最新结果，如果在每日合并开始的时候，我们调用预先定义的查询(不带 read_static)，并将增量更新融合到 LocalCube 中，那么 LocalCube 就自动更新了。

具体流程为：每日合并开始时，触发 select 语句读取基线数据(使用 LocalCube)和增量数据，并用聚合结果替代 LocalCube。则完成了对 LocalCube 的增量更新，接下来的查询可以使用 LocalCube。

5.4 UpdateServer 全量数据

5.4.1 UpdateServer 全量数据定义

定义 5.4.1 查询基线修改比例：一条 sql 查询语句所涉及的列的基线修改比例。若表 t1 一共有 k 列，分别为 c1,c2,…ck,一条 sql 语句中出现的列为 m，分别

为 c1,c2,…,cm，该表基线数据的行数为 n，所请求列的基线数据在 UPS 上更新的行数分别为 c1_n,c2_n,…,cm_n。Modify_percent= (c1_n+c2_n+…+cm_n)/n*m。

```
select sum(col1) from table1 group by col2;
```

该句 sql 中所请求列为 col1,col2,假设 table1 基线数据的行数为 1000 行，其中 col1,col2 的基线数据被更新的行数分别为 1000 行，999 行，则查询基线修改比例为（1000 + 999）/ 1000*2 = 99.95%

定义 5.4.2 UPS 全量数据：满足查询的所有数据全部在 UpdateServer 上，包括两种情况，一种情况为查询基线修改比例为 100%，查询所需的基线数据都被修改了；另一种情况为基线数据为 0，这表明所有数据刚插入，数据全部在 UPS 上。

在全量数据的情况下，查询所需的数据全部在 UpdateServer 上，因此，可以直接计算好结果然后返回，最大程度的减少网络传输，ChunkServer 的磁盘扫描，分组，排序，数据融合等操作。比较典型的场景是计算导入表的数目，具体内容如下节所示。

5.4.2 UPS 全量数据的应用

(1) 计算表的行数

在实际生产上有一个很强烈的需求，在刚导入完一批数据后，需要验证导入的数目是否正确，要查询导入表的行数，在优化前，导入一亿条记录后，执行 select count(*) from table1 语句需要 20 分钟左右，由于数据刚刚导入，数据全部在 UPS 上，按照 OceanBase 原有的流程，UpdateServer 会传给 ChunkServer 全表的主键信息，ChunkServer 将基线数据的主键(此时为零)和增量数据的主键采用二路归并融合，计算表的行数并返回给客户端。

此时的场景为 UpdateServer 全量数据的场景，数据刚刚导入，表的数据全部在 UpdateServer 上，UpdateServer 直接计算出 count 结果发给对应的 ChunkServer，完全没有主键信息的传输。

具体流程如下：首先，MergeServer 将查询请求发送给对应的 ChunkServer，ChunkServer 向 UpdateServer 请求增量数据，并将 LocalCube 中的 count 值发到 UpdateServer，此时 LocalCube 中的 count 值为 0。然后，UpdateServer 接受到查询请求后，扫描 B+树中行操作链表的标记，如果 flag 为 Static_Insert 则 count++，如果 flag 为 Static_Delete 则 count--，如果 flag 为 Static_Update 或 NULL 则 count 不变。扫描完毕后将计算好的 count 值发送给对应的 ChunkServer，ChunkServer 直接将 count 值返回给 MergeServer，MergeServer 融合 count 值并返回客户端。

可以看出计算表行数的方法，不仅适用于全量数据的场景，也适用于一般的场景。它的计算表达式为：$Count_{Static} + Count_{Static\_Delete} + Count_{Static\_Insert}$。其中 $Count_{Static}$ 为基线数据的行数，$Count_{Static\_Delete}$ 为基线数据删除的行数，$Count_{Static\_Insert}$ 为新增数据的行数。

(2) 方案效率分析

OceanBase 原有流程中，count(*)的时间代价主要包含三个部分，UpdateServer 扫描 B+树的代价，增量数据传输给 ChunkServer 的代价以及基线数据和增量数据融合代价。

设扫描每条增量数据的代价为 T_{incre}，增量数据为 m 条，则扫描代价为 mT_{incre}。

设传输每条增量的数据花费 T_{trans}，网络传输代价为 mT_{trans}；

设扫描每条基线数据的代价为 T_{static}；融合一条基线数据和增量数据的代价为 T_{fuse}；设基线数据的数目为 n，则融合代价为 $nT_{static} + (m+n)T_{fuse}$；

则 OceanBase 原有流程 count(*)的代价为 $m(T_{incre}+T_{trans}) + nT_{static} + (m+n)T_{fuse}$；

而本方案的时间代价为：mT_{incre}。其中，网络传输代价基本忽略不计，UpdateServer 传输的是 count 值，而不是主键信息；融合代价为零，LocalCube 中缓存了当前基线数据的表行数，无需扫描基线数据，无需融合基线数据。

本方案 count(*)的代价只有扫描 UpdateSever 上 B+树的代价，扫描操作是在 UpdateServer 的内存中进行，全内存操作，而且 B+树的性能经过优化的，支持并发查询，因此 count 的性能极大提高。

5.5 本章小结

本章首先介绍了 LocalCube 方案的不足，即当增量数据的量比较大时，网络传输会产生较长的响应时间，为了进一步优化，提出了增量数据先聚合再传输的方案，当新行选择率较低时能大大的减少传输量。接着给出了优化方案的设计和实现，并从理论上验证了算法的有效性。

第六章 实验分析

本文设计了一系列的实验来验证 LocalCube 方案在 OceanBase 的架构下，能极大的提高聚合运算的性能。本章将给出这些实验过程，并针对不同的场景来对实验结果做细致的分析和总结。

6.1 实验环境介绍

（1）硬件设备

CPU：Intel(R) Xeon(R) CPU E5-2620 * 2，2 * 6 *2 个线程，主频 2000MHz，2500MHz（超频），L3 缓存 15MB

内存：168GB、152GB、158 GB、168GB、80GB、144GB、128 GB、112 GB、144 GB、128 GB

网络带宽：1000Mb/s（有少数部分机器之间的网络带宽为 100Mb/s）

磁盘 IOPS：76*4=304，磁盘带宽 400MB/s、6500MB/s（读缓存）

（2）系统环境

服务器上部署的开源的 OceanBase0.4.2 版本，实验搭建了一个 OceanBase 集群，由 5 个节点构成，节点 1 上运行着主 RootServer 和主 UpdateServer，节点 2 上运行着备 RootServer 和备 UpdateServer，节点 3,4,5 上运行着 ChunkServer 和 MergeServer。

（3）实验数据

实验所用数据为数据生成器随机生成数据。数据中组的个数，增量数据和基线数据的比例等可变因子由程序控制。表 6.1 显示了测试数据表的 schema 信息，一共为三列，其中 col1 为主键，col2 为分组属性。

表 6.1 测试表的 schema 信息

属性名称	是否为主键	数据类型	数据长度（Byte）
Col1	是	int	4
Col2	否	int	4
Col3	否	int	4

6.2 LocalCube 创建性能

select sum(col2),count(col2),avg(col2),max(col3),min(col3) from table1 group by col1;

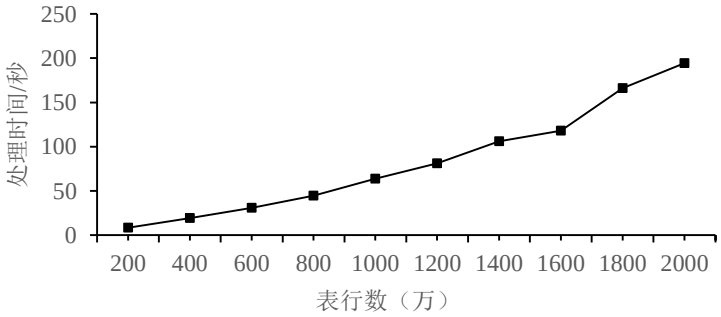

图 6.1 表的行数对 LocalCube 创建的影响

图 6.1 为创建单表的 LocalCube 所需的时间，可以看出，当表的行数越大时，创建 LocalCube 的时间也越长。创建 LocaLCube 的开销为对基线数据做聚合运算所用的开销，该方案中不存在增量数据和基线数据的融合，因此没有网络开销。

6.3 基于 LocalCube 聚合运算性能分析

从算法效率分析中可知，基于 LocalCube 聚合运算的查询不仅与组的个数，增量数据和基线数据的比例，增量数据中新行数据和基线修改数据的比例有关，而且在优化方案中还与新行的分组选择率有关。通过以下的实验来证明我们的结论。

6.3.1 增量数据和基线数据的比例对查询的影响

实验条件：数据总量为 2000 万行，增量数据中新行数据和基线修改数据的比例为 1:1，组的个数为 100。

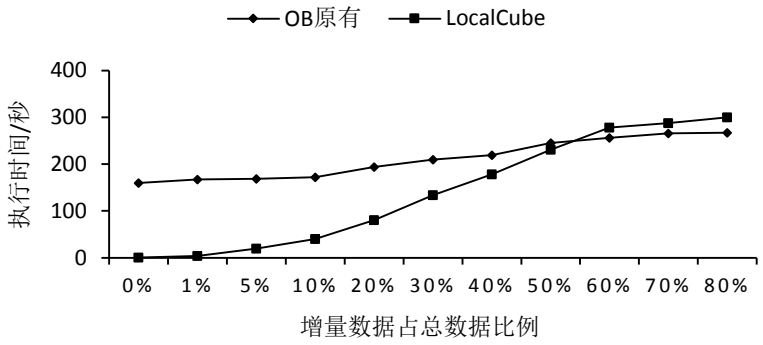

图 6.2 增量数据比例对聚合性能的影响

结果分析：如图 6.2 所示，随着增量数据占总数据比例的增加，OB 原有方案和基于 LocalCube 方案的查询时间都会增加，当增量数据比例小于 50%时，LocalCube 的方案明显优于 OB 原有的方案，当增量数据比例大于 50%时，LocalCube 的方案不如 OB 原有的方案，如图 6.2 所示。

原因分析：在 OB 原有的方案中，系统的开销主要为网络开销和磁盘开销，当增量数据占比越来越大，虽然磁盘开销越来越小，但是网络开销增长更大，因此，整体查询时间呈现增长的趋势。

在基于 LocalCube 的方案中，优化的是 OB 原有开销中磁盘的部分，因此，当增量数据增多，基线数据减小时，磁盘的开销不是查询的主要瓶颈时，优化的效果会逐渐降低，下面做具体的分析：

当增量数据为零时，所有的数据都分布在基线服务器上，查询的所有开销为聚合基线数据花费的开销，而基线数据的聚合值已经缓存在 LocalCube 中，在查询时，系统直接返回 ChunkServer 上的 LocalCube，无需任何的融合修正操作，因此该场景的性能最高，查询基本不耗时。

随着增量数据的增加，一方面网络的开销也在增大，另一方面增量数据融合 LocalCube 的开销不断的增加，主要集中在增量数据中基线修改部分通过主键查

找原值的过程，涉及磁盘开销，根据第四章算法效率分析中的表达式，当增量数据越多，基线修改数据占增量数据的比例越大时，该表达式可能为负值，表示基于 LocalCube 的查询性能在某一时刻会不如 OB 原有的性能。

从理论上分析，如果 n=m，且 m 全为基线修改行，即 $m=m_{oldRow}$，那么表达式则为$(n-m_{oldRow})\ T_{static} + (n-G-m-m_{oldRow})\ T_{fuse} + (n-m-m_{oldRow})\ T_{sort-group} = -G\ T_{fuse} - (T_{fuse}+T_{sort-group})m < 0$。

因此，从该实验可知，增量数据占总数据的比例大约在 55%时，LocalCube 的性能和 OB 原有性能一样，当大于 55%时，OB 原有的性能反而更高。

OceanBase 在设计之初就考了上层应用的特点，更新数据占数据总量比例非常小，通常情况下更新数据只占总数据的 1%。因此，在这样的架构下，LocalCube 的方案是有现实意义的。

6.3.2 组的个数对查询的影响

实验条件：数据总量为 2000 万行，增量数据占总数据的 1%。增量数据中新行数据和基线修改数据的比例为 1:1。

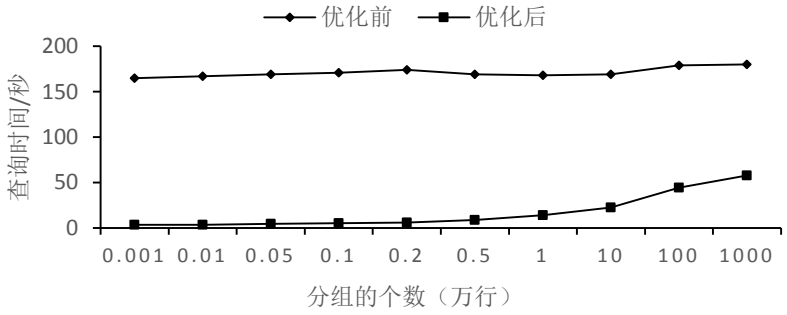

图 6.3 分组个数对聚合性能的影响

结果分析：如图 6.3 所示，OB 原有的查询流程对组的变化不明显，而在基于 LocalCube 的方案中，组对查询的影响很大。

原因分析：在 OB 原有的查询流程中，基线服务器先将增量数据和基线数据融合，然后再按分组属性排序，无论分多少的组，这个代价是固定的，因此组对

原有查询流程的影响不大。在 LocalCube 的方案中，查询流程会涉及增量数据和 LocalCube 的融合，如果组的数目越大，则需要融合的次数就越多，那么查询越耗时，从上图中可以看到，当 LocalCube 中组的数目达到万级，百万级时，查询延时上升的非常快，虽然延迟上升了，但是二路融合过程是无需排序分组等操作，且是内存操作总体性能还是要远远高于 OB 原有的性能。

组的数目越多，不仅会影响查询时间，而且需要更多的存储空间，在极端条件下，如果选取主键作为分组属性，那么 LocalCube 的数目和原表数目相等，这种情况是要避免的，因此，要合理的选择分组属性。

6.3.3 增量数据中新行数据和基线修改数据的比例对查询的影响

实验条件：数据总量为 2000 万行，其中基线数据为 1980 万，增量数据为 20 万，增量占总数据的 1%，组的个数为 100。其中增量数据分为新行数据即新插入的数据和基线修改数据。

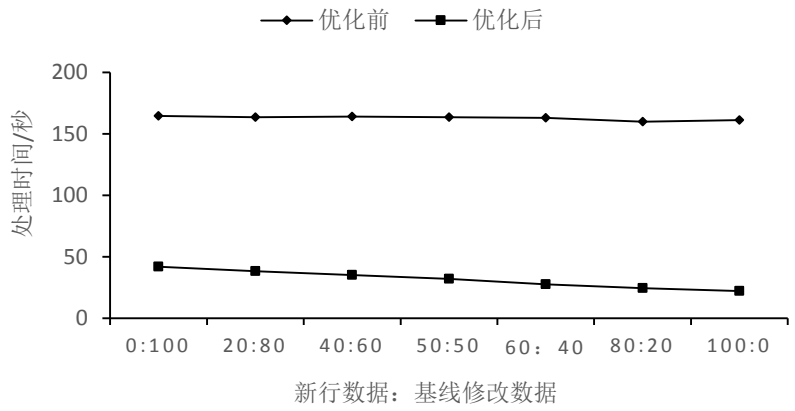

图 6.4 新行比例对聚合性能的影响

结果分析：如图 6.4 所示，新行数据和基线修改数据的比例对 OB 原有查询的性能几乎没有影响，而对基于 LocalCube 的查询性能影响较大，当增量数据中新行比例越大时 LocalCube 的性能越好，反之性能越差。

原因分析：在 OB 原有的查询流程中，增量数据会和基线数据根据主键顺序采用二路归并融合，融合的开销为$(m+n)T_{fuse}$，其中 $m=m_{oldRow}+m_{newRow}$，无论新

行数据和基线修改数据的比例为多少，融合代价不变，因此 OB 原有的查询性能不受影响。在 LocalCube 的方案中，在融合增量数据和 LocalCube 之前会先获得基线修改行的原值，它的开销和基线修改数据量成正比。当增量数据中的基线修改操作较多时，则去磁盘取原值的开销就越大，查询性能降低。反之，如果增量数据中的新行越多，则去磁盘获取原值的开销就越小，极端情况下，如果全为新行，则没有取原值的过程，直接将新行和 LocalCube 融合，查询性能会大大提高。

6.3.4 数据量对 LocalCube 方案的影响

实验条件：增量数据占总数据的 1%。增量数据中新行数据和基线修改数据的比例为 1:1，组的个数为 100。

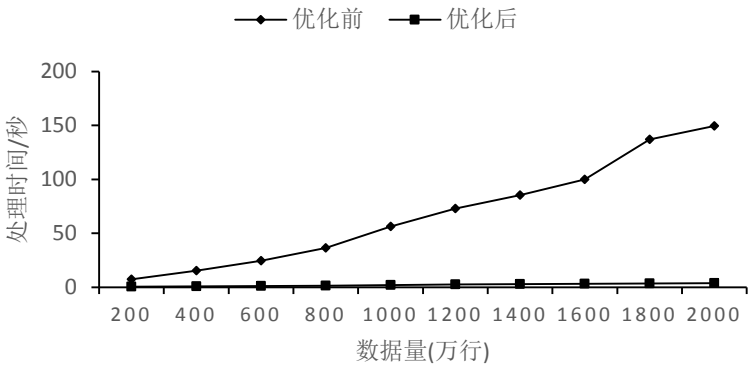

图 6.5 数据量对聚合性能的影响

结果分析：如图 6.5 所示，随着数据量的增加 OB 原有查询时间会越来长，而且增长很快，而基于 LocalCube 的查询性能虽然也在增长，但是增长很缓慢。

原因分析：在增量数据占总数据的 1%的情况下，系统的查询瓶颈主要为基线服务器的磁盘开销，而 LocalCube 已经缓存了预计算的聚合结果，因此能将查询时间维持在一个很小的水平。

6.4 LocalCube 改进方案性能分析

6.4.1 新行分组选择率对查询性能的影响

实验条件：数据总量为 2000 万行，增量数据占总数据的 30%，增量数据中

新行数据和基线修改数据的比例为 1:1，组的个数为 100。

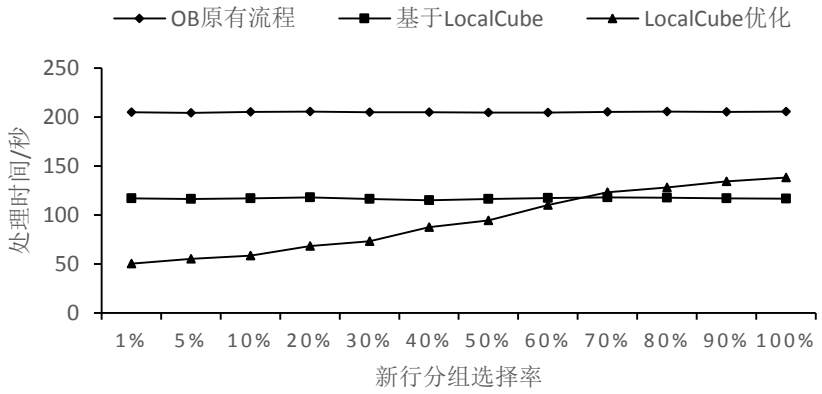

图 6.6 新行分组选择率对聚合运算的影响

结果分析：如图 6.6 所示，新行分组选择率对 OB 原有查询的性能几乎没有影响，对 LocalCube 的查询性能也没有影响，而对 LocalCube 改进方案的影响较大，当新行分组选择率越大，该方案的查询性能越差，反之性能越好。

原因分析：在 OB 原有的流程和 LocalCube 的方案中，并没有对网络传输这块做优化，而在 LocalCube 的改进方案中采用对增量数据的新行先聚合再发送的方式来减少增量数据的传输量，从而使网络的开销降低。

LocalCube 改进版的方案性能提高的程度为：$m_{newRow}(1–UPS_{Selectivity})\ T_{trans} + m_{newRow}(1–UPS_{Selectivity})(\ T_{sort\text{-}group}+T_{fuse}\ ) – m_{newRow}T_{Agg}$

从上述表达式可知，它的查询代价与新行分组选择率负相关，新行分组选择率越低，即经过聚合后的结果集元组数越少，则性能提高越多。例如：新行分组选择率为万分之一，那么 300 万新行经过聚合后为 300 行，则传输的代价大大降低。当新行选择率为 100%时，即经过聚合后结果集元组数和新行数相等，上述的表达式化简为$–m_{newRow}T_{Agg} < 0$，说明 LocalCube 改进版的方案反而较改进前有所下降，下降的开销为新行聚合的开销。

因此，从该实验可知，新行分组选择率大约在 70%左右，LocalCube 改进版的性能和改进前一样，当大于这个数时，改进前的性能反而更高。

在实际的数据集中，如果新行的绝对量比较大，比如有 300 万数据，而 LocalCube 的组数为 100，那么新行的分组选择率很有可能为 100/3000000，较少的可能为 100%（极端情况也可能）。一般情况下如果，LocalCube 的组个数维持在 100 到 1 万左右，采用 LocalCube 的改进版方案是能降低网络开销的。

6.4.2 统计表行数 count(*)

实验条件：数据量从 100 万到 2000 万不等，数据刚导入，全部在 UpdateServer 上。

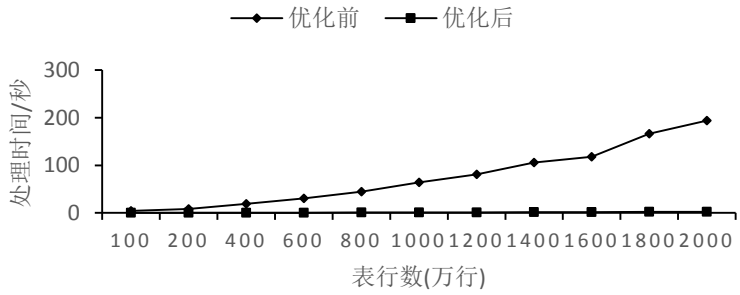

图 6.7 表行数统计

结果分析：如图 6.7 所示，随着数据量的增加 OB 原有查询时间会越来长，而且增长很快，而 count(*)查询时间虽然也在增长，但是增长非常缓慢。

原因分析：统计某张表的行数，select count(*) from table1;该查询没有分组操作，只需扫描 B+树中行操作链表的标记即可，该过程为全内存操作，查询性能非常好。在该实验中 2000 万的数据大约 346MB，而内存的速度量级为 GB/S，查询 2000 万表的行数开销不到 1 秒。

6.5 本章小结

本文通过设计一系列的对比实验来验证了基于 LocalCube 算法的有效性，这种方案在 OceanBase 的架构下能极大的提高聚合运算的性能。

第七章 总结和展望

7.1 本文总结

OceanBase 分布式关系型数据库针对了上层应用更新数据少，基线数据多的特点，将增量数据和基线数据分开存放，这种架构很好的满足了事务的 ACID 特性和高可扩展性，目前 OceanBase 已经全面的支撑了阿里巴巴集团的各种应用，如支付宝，淘宝，聚划算等，成为国内最具特色的分布式数据库。

由于 OceanBase 增量数据和基线数据分离的特点，聚合运算如 sum，count，avg，min，max 的性能非常低下，主要有以下两个原因，首先，系统要满足实时查询的需求，必须先将增量数据和基线数据融合再做聚合运算，这个过程涉及到节点的数据传输，占用大量的响应时间。其次，聚合运算涉及排序，聚合等操作，当读取的基线数据量很大时，磁盘开销就是一个很大的瓶颈。目前学术界对 OceanBase 这种独特架构的研究很少，因此本文以聚合运算作为切入点，提出了基于 LocalCube 的聚合运算。

本文首先探讨了 LocalCube 的定义和结构，接着简述了它的构造，更新和管理的流程，并验证了 LocalCube 的适用性，接着给出了基于 LocalCube 聚合查询的方案。

基于 LocalCube 的方案主要针对更新数据少，基线数据多的场景，这和 OceanBase 的架构设计是保持一致的。通过设计 LocalCube 的结构，将基线数据聚合结果存放到 LocalCube 中，查询过程只需融合增量数据和 LocalCube，最大限度的减少了重复计算带来的磁盘开销和排序开销。

在前一种方案的基础上，提出了一种更有效的优化方案，通过将增量数据先聚合再传输，在新行分组选择率低的情况下，能极大的减少数据的传输量，从而进一步优化查询。

最后，本文通过理论和实验比较了基于 LocalCube 查询和 OceanBase 原有查询的效率，验证了 LocalCube 的有效性。

7.2 未来工作

基于 LocalCube 的查询对全局性聚合运算的效果在最大值/最小值没有被删除或者插入的值比最大值大或比最小值小的情况下，效果很好。否则，性能退化到原有 OceanBase 的性能，因此如何解决上述场景的问题是未来研究的方向。

本文提出了 LocalCube 这种结构，但是如何选取合适的分组属性，来减少总体的存储代价是未来的研究方向。

目前 LocalCube 采用批量更新的方式进行更新，LocalCube 只是存储了基线数据的聚合结果，每次查询还是要融合增量数据和 LocalCube，如果让 LocalCube 实时和最新数据同步，那么查询的性能会更高。

参考文献

[1] Bachman C W. The Programmer as Navigator.[J]. Communications of the Acm, 1973, 16(11):635-658.

[2] Kamfonas M. Recursive hierarchies: the relational taboo[J]. The Relational Journal, 1992, 27(10).

[3] Cattell R G G, Barry D K, Bartels D, et al. The object database standar d: ODMG 2.0[C]// Morgan Kaufmann Publishers Inc. 1997:62-63.

[4] Codd E F. A relational model of data for large shared data banks[J]. Co mmunications of the Acm, 1983, 26(1):64-69.

[5] The Open Group. Distributed TP: The XA Specification. Free PDF-https:// www2.opengroup.org/ogsys/catalog/C193, 1992.

[6] 阳振坤. OceanBase 关系数据库架构[J]. 华东师范大学学报(自然科学版), 2 014(5): 141-148.

[7] Stonebraker M. SQL databases v. NoSQL databases[J]. Communications of the ACM, 2010, 53(4): 10-11.

[8] Redis. http://redis.io/. [Online; accessed 13-October-2016].

[9] Tokyo Cabinet. http://fallabs.com/tokyocabinet/.[Online; accessed 13-October -2016].

[10] Flare. http://labs.gree.jp/Top/OpenSource/Flare-en.html. [Online; accessed 13 -October-2016].

[11] Chodorow K. MongoDB: The Definitive Guide[M]. O'Reilly Media, Inc. 2 013.

[12] Anderson J C, Lehnardt J, Slater N. CouchDB: The Definitive Guide[J]. Andre, 2010, 215(1):pages. 76-80.

[13] Chodorow K. 50 tips and tricks for MongoDB developers[J]. European Ur-ology Supplements, 2011, 38(2):348.

[14] Lakshman A, Malik P. Cassandra: a decentralized structured storage syste-m[J]. Acm Sigops Operating Systems Review, 2010, 44(2):35-40.

[15] Sumbaly R, Kreps J, Gao L, et al. Serving large-scale batch computed da ta with project Voldemort[C]// Usenix Conference on File and Storage Tec hnologies. 2012:18-18.

[16] Gray J. The Transaction Concept: Virtues and Limitations[J]. Proc of Vld b, 1981:144-154.

[17] Alibaba. OceanBase. https://github.com/alibaba/oceanbase. [Online; accessed 13-August-2016].

[18] Gupta A, Mumick I. Maintenance of Materialized Views: Problems, Techn iques, and Applications[C]// MIT Press, 1999:145 - 157.

[19] Gray J, Chaudhuri S, Bosworth A, et al. Data Cube: A Relational Aggreg ation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals[J]. Data Mining & Knowledge Discovery, 1996, 1(1):152-159.

[20] Palpanas T, Sidle R, Cochrane R, et al. Chapter 69 – Incremental Mainten ance for Non-Distributive Aggregate Functions[C]// International Conferenc e on Very Large Data Bases. 2002:802-813.

[21] Oracle Corporation. Oracle. https://www.oracle.com/database/index.html. [On line; accessed 12-August-2016].

[22] Oracle Corporation and/or its affiliates. MySQL. https://www.mysql.com/. [Online; accessed 13-August-2016].

[23] IBM. DB2. https://www.ibm.com/analytics/cn/zh/technology/db2/. [Online; accessed 12-August-2016].

[24] Blakeley J A. Efficiently updating materialized views[J]. Acm Sigmod Record, 1970, 15(2):61-71.

[25] Gupta A, Mumick I S, Rao J, et al. Adapting materialized views after redefinitions: techniques and a performance study[J]. Information Systems, 2001, 26(5):323-362.

[26] Mumick I S, Quass D, Mumick B S. Maintenance of data cubes and summary tables in a warehouse[J]. Acm Sigmod Record, 1997, 26(2):100-111.

[27] Ceri S, Widom J. Deriving production rules for incremental view maintenance[C]// MIT Press, 1996:441-463.

[28] Griffin T, Libkin L. Incremental maintenance of views with duplicates[M]// Materialized views. MIT Press, 1999:328-339.

[29] Abiteboul S, Mchugh J, Rys M, et al. Incremental Maintenance for Materialized Views over Semistructured Data[C]// Int Conf on Very Large Databases, New-York. 1998.

[30] Yue Z, Garcia-Molina H. Graph structured views and their incremental maintenance[J]. 1999, 20(3):116-125.

[31] 张东站, 黄宗毅, 薛永生. NDSMMV 一种多维数据集物化视图动态选择新策略[J]. 计算机研究与发展, 2008, 45(5):901-908.

[32] Harinarayan V, Rajaraman A, Ullman J D. Implementing data cubes efficiently[M]// Materialized Views:Techniques, Implementations, and Applications. MIT Press, 1999:343-360.

[33] Shukla A. Materialized view selection for multidimensional datasets[C]// V

ldb'98, Proceedings of, International Conference on Very Large Data Base s, August 24-27, 1998, New York City, New York, Usa. 1998:488-499.

[34] Zhang C, Yao X, Yang J. An evolutionary approach to materialized views selection in a data warehouse environment[J]. IEEE Transactions on Syste ms Man & Cybernetics Part C, 2001, 31(3):282-294.

[35] Yu J X, Yao X, Choi C H, et al. Materialized view selection as constrai ned evolutionary optimization[J]. IEEE Transactions on Systems Man & C ybernetics Part C, 2003, 33(4):458-467.

[36] Harinarayan V. Implementing data cubes efficiently[M]// Materialized View s:Techniques, Implementations, and Applications. MIT Press, 1999:343-360.

[37] Kalnis P, Mamoulis N, Papadias D. View selection using randomized sear ch[J]. Data & Knowledge Engineering, 2002, 42(1):89-111.

[38] Yan W P. Interchanging group-by and join in distributed query processing [C]// Conference of the Centre for Advanced Studies on Collaborative Res earch, October 24-28, 1993, Toronto, Ontario, Canada, 2 Volumes. 1993:8 23-831.

[39] Yan W P, Larson P A. Eager Aggregation and Lazy Aggregation[J]. Vldb, 1970:345--357.

[40] Hu Y, Sundara S, Srinivasan J. Supporting Time-Constrained SQL Queries in Oracle.[C]// International Conference on Very Large Data Bases, Univer sity of Vienna, Austria, September. 2007:1207-1218.

[41] TPC-D. http://www.tpc.org/. [Online; accessed 13-October-2016].

[42] Microsoft. Microsoft SQL Server. http://www.microsoft.com/zh-cn/server-clo ud/products/sql-server/. [Online; accessed 12-August-2016].

[43] Venkatsubra V, Jain V. Method, system and article for dynamic real-time

stream aggregation in a network: WO, US7519724[P]. 2009.

[44] Bellamkonda S, Dageville B. Efficient data aggregation operations using h ash tables: US, US 7469241 B2[P]. 2008.

[45] Hu Y, Sundara S, Srinivasan J. Estimating aggregates in time-constrained approximate queries in Oracle[C]// EDBT 2009, International Conference o n Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings. 2009:1104-1107.

[46] Hu Y, Sundara S, Srinivasan J. Supporting Time-Constrained SQL Queries in Oracle.[C]// International Conference on Very Large Data Bases, Univer sity of Vienna, Austria, September. 2007:1207-1218.

[47] Widom J. Review - Query Evaluation Techniques for Large Databases.[J]. 1999, 1.

[48] Lohman G M, Pirahesh M H, Shekita E J, et al. Optimization of data re partitioning during parallel query optimization: US, US 6112198 A[P]. 20 00.

[49] Shatdal A, Naughton J F. Adaptive parallel aggregation algorithms[J]. Ac m Sigmod Record, 1995, 24(2):104-114.

[50] Shatdal A. Random sampling of rows in a parallel processing database sy stem: US, US 6564221 B1[P]. 2003.

[51] Al-Houmaily Y. J., Samaras G. Two-phase commit[M]//Encyclopedia of D atabase Systems. Springer US, 2009: 3204-3209.

[52] Robertson A. Linux-HA heartbeat system design[C]// Linux Showcase & Conference. 2000:20-20.

[53] 杨传辉.大规模分布式存储系统原理解析与架构实现[M].机械工业出版社, 2 013:12-17.

致谢

时光荏苒，岁月如梭，又到了毕业的时刻，回想这两年半的时光，充满了感动和温暖。初到海量所这个心仪已久的团队，遇到很多优秀的老师和同学，自己由于基础比较薄弱，在项目和学习上遇到了很多的困难，自身的能力不能迎接项目和学习的挑战，在那段日子里各位老师和同学对我不离不弃，鼓励我帮助我，让我从一点一滴做起。我从中学到了很多也收获了很多，我的学习能力、思维方式和科研精神都得到了提升。研究生二年级，在交通银行实习，在工作中也遇到了很多困难，是老师和同学们给予了很多的帮助，让我渡过难关。在此，我要向那些关心我，鼓励我，帮助我的老师和同学们表示最衷心的感谢和祝福。

首先，我要感谢我的导师宫学庆教授。在研究生的几年中，宫老师给予我无微不至的关怀和照顾。每当我在生活或是学习上遇到困难时，宫老师总是能帮我分析问题所在，然后结合我的情况提供最有效的建议，鼓励我做最好的自己。在交通银行实习的时间里，宫老师在工作中也体现了很强的工作能力，总是能在最短的时间里将问题简化，这给我很大的启发。宫老师在论文方面也给予我很大的帮助，本篇论文从选题到设计，再到实现都有宫老师的悉心指导，每次和宫老师交谈都能有新的启发和感受。

其次，我要感谢我的指导老师张蓉教授。初到海量所，张老师带着我做研究，张老师对工作认真负责，做事雷厉风行，总能高效的完成各项工作，我在张老师指导下一步步的成长。张老师对学生非常的用心，事无巨细，我发给张老师的所有报告，张老师总是认真的评阅，并给予宝贵的指导意见。本篇论文的撰写，张老师作为我的指导老师，每个章节都仔细的修改，从格式到标点符号，到语言的组织，到论文思想的表达，都倾注了她大量的心血，在此，我祝福张老师永远年轻漂亮，心想事成。

我还要感谢实验室的其它老师们。感谢钱卫宁老师对我们本届专硕论文的统一指导，感谢张召老师对我学习和生活的关注，感谢周傲英老师每次振聋发聩的演讲，感谢金澈清、高明、蔡鹏、何晓丰、王晓玲等各位老师们对我学习的帮助。

另外，我要感谢我身边的同学们。感谢郭进伟博士，李宇明博士，朱涛博士，储佳佳博士，朱燕超博士，段惠超博士，周欢博士，感谢他们对我平时学习的帮助以及论文的帮助和指点；感谢刘伯众，祝君，余晟隽，王雷，钱招明小伙伴对我的帮助，有了你们我的生活非常的开心；还要感谢实验室 OceanBase 项目组的所有成员。

最后，我要感谢我的父母和家人，他们对我无微不至的关怀和谆谆教诲使我不断的成长，没有你们的鼓励和帮助就没有今天的我，谢谢你们。

谨以此文感谢所有关心和帮助过我的人们，再次谢谢你们。

熊辉

二零一六年十月九日

发表论文和科研情况

■ 已发表或录用的论文

[1] 软件著作权：数据库事务流量抓取和重放软件

■ 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