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：51141500090

華東師範大學

East China Normal University

硕士学位论文

MASTRER'S DISSERTATION

论文题目 ： **面向可扩展数据库管理系统的应用迁移与改造**

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 ：	钱卫宁 教授
学 位 申 请 人 ：	余晟隽

2016 年 9 月 30 日

Dissertation for professional master's degree in 2017

Student ID: 51141500090

University Code: 10269

East China Normal University

Title: **THE MIGRATION AND OPTIMIZATION FOR APPLICATION IN SCALABLE DATABASE MANAGEMENT SYSTEM**

Department:	**School of Computer Science and Software Engineering**
Professional degree category:	**Master of Engineering**
Professional degree field:	**Software Engineering**
Supervisor:	**Prof. QIAN Weining**
Candidate:	**YU Shengjun**

September, 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《面向可扩展数据库管理系统的应用迁移与改造》，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

《面向可扩展数据库管理系统的应用迁移与改造》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，本论文的研究成果归华东师范大学所有。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，并向主管部门和相关机构如国家图书馆、中信所和“知网”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）。

余晟隽 硕士专业学位论文答辩委员会成员名单

姓名	职称	单位	备注
翁楚良	教授	华东师范大学	**主席**
张蓉	副教授	华东师范大学	
沙朝锋	副教授	复旦大学	
蔡鹏	副教授	华东师范大学	
张召	副教授	华东师范大学	

摘　要

关系型数据库历经数十年的发展，在各行各业后台电子信息系统中都有着广泛的应用。随着互联网飞速发展，许多传统关系型数据库系统正面临着的巨大挑战。这些挑战主要来自两个方面：一方面，随着信息沟通的加快，企业为了存储数据越来越多的业务历史数据，需要付出越来越高的成本；另一方面，企业机房进行硬件升级的繁琐流程难以适应爆发性增长的业务特征。

可扩展数据库管理系统，由于它可扩展性强，搭建成本低，容灾方案完备等特点，受到了广大企业的关注。越来越多的企业开始尝试将运行在传统关系型数据库上的业务迁移至可扩展数据库管理系统。但由于这类系统查询执行逻辑简单、网络传输效率较低等原因，可扩展数据库管理系统的查询性能往往较差，使得业务迁移过程中企业需要面临许多改造的难题。本文从这类难题出发，有如下贡献：

1. 通过应用改造的方式，解决了运行在传统关系型数据库上的业务系统向可扩展数据库系统进行业务迁移后，系统整体查询性能降低，无法满足业务需求的问题。

2. 解释了不同负载特点场景下，查询性能降低的原因，提出了通过应用改造提升查询性能的五方面策略，分别涉及可扩展数据库管理系统的库表结构设计、二次索引构建、定期合并优化、函数调用优化以及连接算法选择。

3. 验证了应用改造策略的有效性。本文通过 CEDAR 数据库系统，验证了企业将真实应用从传统关系型数据库系统迁移至 CEDAR 数据库系统时，应用改造对查询性能提升的有效性。

本文提出的应用改造策略在 CEDAR 系统的测试结果表明，通过应用改造，可以使业务系统的查询满足业务需求。同时，本文提出的应用改造策略对其他不同类型的可扩展数据库管理系统有借鉴意义，也为可扩展数据库管理系统后续的查询优化开发工作提供了参考。

关键词：应用改造，可扩展数据库管理系统，性能调优，数据迁移

Abstract

Relational database has the extensive applications in electronic information system in each industry and field after the development for several decades. With the rapid development of Internet, numerous traditional relational database systems are facing the enormous challenges. And these challenges mainly come from two aspects: on one hand, with the fastening of information communication, in order to store the increasing business historical data, enterprises need to pay increasing costs; on the other hand, the tedious process of hardware upgrading in enterprise machine rooms is difficult to adapt to business characteristic of explosive increase.

Scalable database management system has received the attention of vast enterprises due to its strong scalability, low establishment costs, complete disaster recovery solution and so on. More and more enterprises have started to attempt transferring the business in traditional relational database to the scalable database management system. For this type of systems are simple in query execution and lower network transmission efficiency, the query performance of scalable database management system is worsen, causing numerous transformation difficulties in the business migration process. This paper starts from this difficulty and makes the contributions as below:

1. Through the method of application transformation, it solves the problem that the overall inquiry performance of the system is reduced and the business demand cannot be satisfied after the business system operating in the traditional relational database has transferred its business to scalable database system.

2. It interprets the reasons of inquiring performance decline in the scenarios of different load characteristics, proposes the strategies of improving inquiry performance through application transformation, and separately refers to the library list structure design of scalable database management system, secondary indexing creates, regular

merge optimization, function invoking optimization and join algorithm selection.

3. It demonstrates the effectiveness of application transformation strategy. This paper demonstrates the effectiveness of application transformation on the improvement of inquiry performance while enterprise transfer the realistic applications from traditional relational database system to CEDAR database system through CEDAR database system.

The testing results of application transformation strategies proposed by this paper in CEDAR system demonstrate that through application transformation, the inquiry of business system can satisfy the business demands. In the same time, application transformation strategy proposed by this paper not only equips with reference meaning for other different types of scalable database management systems, but also provides the reference for the subsequent inquiry, optimization and development of scalable database management system.

Key Words: Application transformation, Scalable Database Management System, performance tuning, data migration

目 录

插　图

表　格

第1章 绪论

1.1 研究背景与意义

随着电子信息行业的不断发展与进步，特别是互联网行业的蓬勃发展，越来越多的人通过互联网能够方便地体验购物、娱乐等服务。近年来，许多传统行业积极拥抱互联网，随着业务的不断发展，这些企业的数据量变得越来越大。大型企业存储的数据总量普遍达到 TB（Terabyte）级，有的甚至突破 PB（Petabyte）级别，同时每天还要面临着数百 GB（Gigabyte）的增量数据。以某国有银行的历史库系统为例，数据库中记录最多的十张表，记录数均超过 100 亿，最大的单张表数据量超过 1.3TB。另一方面由于用户的不断增多，业务的爆发性增长成为一种常态，以阿里巴巴公司 2015 年的双十一活动为例，活动期间交易峰值达到 8.59 万笔/秒，仅仅 12 分 28 秒的时间，交易额就突破 100 亿元[1]。伴随着网络购物交易量爆发式增长的是物流行业在双十一活动期间需要处理的运单数量。据统计，在 2015 年双十一活动期间，即 11 月 11 日至 16 日期间，物流行业快件量达到 7.8 亿件，同比增长近 45%，日最高处理快件达 1.6 亿件，是日常处理快件数量的 3 倍[2]。这样具有显著峰值的业务特点也使得物流行业后台信息系统的处理能力面临着巨大的挑战。

为了解决海量数据的存储，处理随时可能爆发性增长的业务，传统上企业往往通过向上扩展（Scale-up）的方式，增强单台服务器的性能，如扩大辅助存储，增加内存容量，使用性能更强劲的处理器等方式来应对增长的业务。但是，采用增强单台服务器性能的扩容方式会存在三个方面的问题。首先，服务器的硬件升级，付出的经济成本不是线性的，性能强劲的服务器也伴随着更高昂的售价。这意味着企业为了获得一倍的性能提升，可能需要付出多倍的经济成本。其次，服务器进行硬件升级周期时间长，这指的是性能强劲的小型机、大型机等硬件设备

需要为企业进行定制化生产，从提交需求订单到获得硬件产品往往需要经历漫长的等待，这与如今飞速发展的业务特点格格不入。最后，单机设备进行软硬件的升级时，对外往往需要中断服务，而这种中断不仅影响了用户的体验，而且可能面临着巨额的经济损失。

相较于增加单台服务器的性能，另一种解决方案是使用分布式集群系统，通过普通商用服务器组成集群，增加服务器的数量就可以使系统的性能近似线性增长。1979 年美国计算机公司（CCA）研制的第一个分布式数据库系统 SDD-1[3]证实了分布式数据库系统的可行性。分布式数据库系统解决了上述系统扩容以及系统但以分布式集群系统为背景的分布式数据库系统也有着自身的不足，这些不足体现在：一方面，分布式场景下的写事务需要通过两阶段提交[4]等协议来保证数据的一致性，但这会使得系统性能降低；另一方面，网络通讯是分布式场景下的一大代价，这也会降低数据库系统的性能。系统的性能问题使得分布式系统开发面临着更大的性能挑战。因此分布式系统被应用到实际生产环境中的进程缓慢。直到近年来相关技术的不断成熟，才渐渐地出现分布式系统的实际用例，如 Google 公司研发的 Megastore 系统[5]、Spanner 系统[6]与 F1 系统[7]。

为了解决高并发、大数据量场景下数据库管理系统的性能问题，一些 NoSQL 系统[8]得到人们的广泛关注。典型的系统有 Amazon 公司研发的 Dynamo[9]，开源的 Redis[10]等键值系统，他们根据主键支持简单的创建，读取，更新和删除服务，具有查找速度快等特点；Facebook 公司研发的 Cassandra[11]，开源的 HBase[12]等列存储系统，区别于传统的按照数据行对数据进行组织和管理，列存储系统每次对一个列的数据进行分组和存储，可扩展性强，查找速度快；还有类似于 MongoDb[13]，CouchDB[14]等文档型数据库。这些 NoSQL 系统一方面不再像传统的数据库管理系统那样支持结构化查询语言（Structured Query Language， SQL）对数据进行一些逻辑复杂的查询，另一方面它们在数据的一致性等方面往往难以达到金融企业对数据强一致性的要求。

在这样的背景下，基于 LSM-Tree（Log-Structured Merge Tree）[15]思想的可扩展数据库管理系统，如可扩展关系型数据库管理系统 CEDAR[16]，开始受到

关注。CEDAR 数据库是一个支持海量数据存储的可扩展数据库系统，提供关系模型以及 SQL 接口。企业可以更加容易地将存储的结构化数据迁移至 CEDAR 数据库系统中。CEDAR 数据库将运行期间新写入的数据维护在一个节点中，将历史数据以数据分片的形式冗余、分布式得存储在其它节点中，仅需要增加存储历史数据的节点，就可以快速的扩展系统的存储性能，体现了可扩展的特性。在系统运行期间通过定期合并的形式将新写入的数据与历史数据进行融合。

如图 1.1 所示，是一个常见的业务系统处理架构模型。来自各地区的用户客户端向业务系统发起请求，完成相应的业务逻辑，业务系统通过数据库连接池与后台的数据库管理系统建立连接，向数据库发起读写请求，完成相应修改，并把结果返回给客户端。

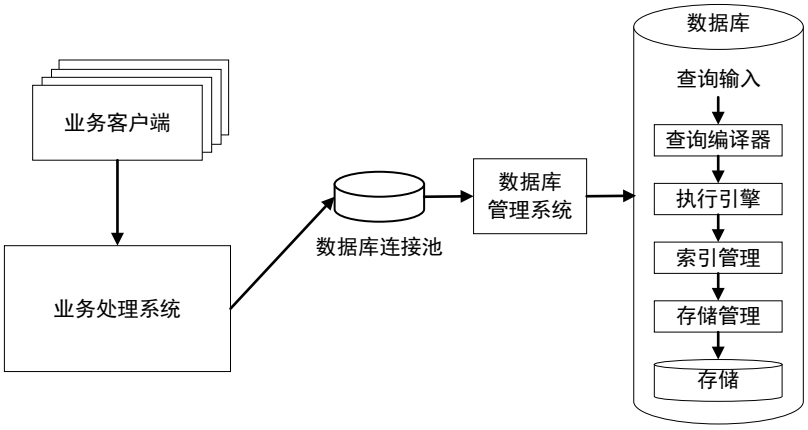

图 1.1 企业常见业务系统架构

当企业使用可扩展数据库管理系统替换原有的数据库管理系统，由于数据库管理系统的系统架构发生了改变，相比于传统的数据库管理系统，数据的读取需要通过远程过程调用（Remote Procedure Call Protocol, RPC）实现，数据在节点间的网络传输成为了一项不可忽视的代价。另一方面，这一类可扩展数据库管理系统发展时间有限，对于查询的处理还仅是简单的实现。将运行在集中式数据库管理系统中的线上业务迁移至这一类系统中往往还伴随查询效率低等问题。因此为了受益于可扩展数据库管理系统采购成本较低，系统性能扩展容易，数据不丢失等优点的同时，还提供较好的查询性能，往往需要对业务进行改造。本文基于对金融和物流行业的线上系统应用改造的经验，分析和总结了

不同的应用负载特点，以应用的迁移与改造为切入点，分析和总结了在可扩展数据库管理系统中应用迁移与改造的可以借鉴的原则。

1.2 本文工作

由于可扩展数据库管理系统在可扩展性以及容灾方面的优势，越来越多的企业开始评估将业务迁移至可扩展数据库系统的可能性。在业务迁移的过程中，往往会遇到两方面的问题。一方面传统数据库系统与可扩展数据库管理系统间的功能不一定兼容，这些不兼容表现在它们提供的 SQL 查询语法不完全一致，以及可扩展数据库管理系统可能不支持如游标、存储过程、触发器等传统数据库系统中常见的功能。另一方面，可扩展数据库管理系统还没有查询优化功能，对用户发起的查询请求只能生成简单且单一的执行计划，无法像传统数据库那样自动的对用户输入的查询进行优化，提升响应时间。

在这样的背景下，为了帮助企业尽快完成业务迁移任务，在得益于可扩展数据库管理系统的扩展性优势的同时，保证业务系统的响应时间能够达到业务部门的真实需求。最有效的方法就是对业务的查询语句进行改造。本文从这一目标出发，完成了以下四个方面的工作：

1) 总结导致系统查询性能较低的负载模型特点。本文从金融行业、物流行业的真实线上应用出发，分析了这些业务的查询语句迁移至可扩展数据库管理系统中响应时延增长原因。针对不同的负载，抽象成不同的负载模型。

2) 如图 1.1 所示，针对数据库系统的存储结构，提出了在可扩展数据库管理系统中三个不同方面的优化方法，分别是表结构优化，索引结构优化以及定期合并优化。阐述了在可扩展数据库管理系统中优化的具体优化策略，通过建立代价模型的方式分析了优化策略的有效性，通过测试验证了优化策略的正确性。

3) 针对图 1.1 中业务处理系统的查询语句，提出了在可扩展数据库管理系

统中两个不同方面的优化方法，分别是表达式优化和连接操作优化。阐述了具体的优化策略，通过建立代价模型，分析了由于底层数据库系统的架构发生了改变，相比于传统数据库系统连接操作的不同。通过实测试面验证了优化策略的正确性。

4) 分别对金融行业和物流行业的某一具体线上应用进行了数据迁移，通过业务改造，测试了应用在原系统，新系统改造前和新系统改造后的三者的性能对比。实验表明相比于改造前的查询性能，改造后业务系统的查询性能得到大幅的提升，对业务系统的关键查询可以近似达到原系统的查询性能。

1.3 本文结构

根据本文所做的研究工作，文章的结构安排如下：

第二章介绍本文研究的相关工作，首先介绍了传统单机场景下，数据库管理系统的一些性能调优方式，之后为了更加清楚的阐述和分析本文提到的各种改造方案的合理性以及有效性，介绍了可扩展数据库管理系统的基本架构、查询执行流程以及基础的运算算子含义，为接下来分析可扩展数据库管理系统中查询改造的有效性分析做好准备。并根据可扩展数据库管理系统的架构特点，简要阐述了改造需要解决的主要问题。

第三章介绍了与存储相关的应用改造技术，阐述了由于可扩展数据库管理系统由于存储扩展容易等特点，它与传统数据库管理系统中一些存储设计的不同之处，解释了这些不同在可扩展数据库管理系统中带来的收益是值得的。

第四章介绍了在可扩展数据库管理系统的场景下，一些查询在系统实现中优化的必要性，针对连接操作等问题，分析了在可扩展数据库管理系统中的连接操作与传统数据库管理系统中执行流程的差异，通过各种不同连接算法的代价对比，给出了不同场景下连接算法的选取原则。

第五章是实际应用改造的效果说明，首先介绍了从传统数据库管理系统中将企业真实数据迁移至可扩展数据库管理系统中的方法，之后结合企业真实 UAT

环境测试，展示了基于三、四章改造原则进行应用改造的有效性。

第六章是对全文工作的总结，对提出的所有改造原则进行回顾，以及对可扩展数据库管理系统未来研究工作的展望，阐述了可扩展数据库管理系统在企业真实应用中还需要解决的问题。

第2章 相关工作

2.1 传统数据库管理系统调优技术

20 世纪 60 年代至今，数据库技术得到了广泛的关注和蓬勃的发展，先后经历了层次模型[17]，网状模型[18]以及如今广泛使用的关系模型[19]。为了避免数据库性能成为业务系统的性能瓶颈，数据库系统调优也成为了研究的热点问题。数据库系统的调优是一个系统的方法，通过修改系统参数、变更系统硬件配置、优化应用组件来提升系统的性能。数据库的调优主要可以分为两个方面的工作，一方面是对数据库自身集成的查询优化器的研究，另一方面是各数据库厂商针对自家发行的数据库产品外围调优工具的研发。

文献[20][21][22]最先开始了关于数据库查询优化问题的研究，随着数据库技术的不断发展，目前不同的数据库产品对系统内部的查询优化器都有着不同的实现。其中，MicroSoft 公司所研发的 SQLServer 数据库[23]对查询优化的相关内容有着深入的研究，文献[24][25]描述了 SQLServer 数据库产品在发展早期使用的基于规则的查询优化框架。甲骨文公司所研发的 Oracle 数据库[26]引入了数据分布直方图来获取数据表中数据的分布特征，并使用了基于代价估计的查询优化方式，在其发布的 Oracle10g 产品开始，已经使用基于代价估计的查询优化方式取代了基于规则的查询优化方式。开源的数据库系统 PostgreSQL[27]则是将遗传算法应用到了查询优化中[28]。

除了数据库系统内部集成的查询优化器意外，许多数据库厂商还提供了一系列的外围工具来帮助用户提升数据库性能。例如 SQLServer 数据库产品提供了查询分析器、索引优化向导等功能。IBM 公司研发的 DB2 数据库[29]则可以通过 Design Advisor 工具[30]来获得对业务表的索引构建建议。Oracle 数据库则提供了 SQL Tuning Advisor 工具[31]对业务的 SQL 语句提供优化建议。

在实际的使用中，对数据库系统进行调优是一个系统性的工程，需要从多方面考虑，文献[32]对于数据库系统的调优问题，提出了 10 个解决步骤。数据库管理系统中的性能调优主要有提高系统吞吐量和缩短用户响应时间两个目标。系统吞吐量指的是单位时间内系统能够完成的 SQL 数量。响应时间指的是用户从客户端发出查询请求到获得查询的第一条记录所需要的时间，常用毫秒或秒表示。各数据库厂商推出的数据产品如 Oracle，SQL Server、DB2 通过降低磁盘 IO、CPU 运行时间等方式尽可能的缩短每个查询的响应时间，同时尽可能的提升数据库系统的吞吐量。

为了避免数据库系统的性能成为业务系统的性能瓶颈，可以从多个层面对数据库管理系统进行调优，主要方式有在数据库设计阶段对库表结构的逻辑、物理模型设计进行调整优化；在应用程序中对 SQL 语言进行调优，如通过查询重写来避免不必要的连接操作，减少对数据的运算等；在数据库管理系统的运行环境中改变配置参数，如调整优化磁盘的 I/O，调整数据库的内存分配等手段。本文主要关注数据库设计与应用程序调优两个方面。

如前所述，数据库管理系统的性能由多种因素决定。在考虑数据库管理系统调优问题时，需要多维度的考虑，分析系统瓶颈。文献[33]综合的阐述了数据库管理系统调优的原则，通过各类数据库的性能实验，分析和比较了不同的调优方式。文献[34]对企业后台电子系统中常见的 Oracle 数据库系统，从 SQL 语言层面对调优方式进行了阐述。近年来 MySQL 数据库系统由于简单、开源、性能良好等优点，受到了越来越多企业的青睐，文献[35]对 MySQL 系统的性能调优做了详细阐述，成为 MySQL 数据库系统管理人员重要的参考手册。

2.2 可扩展数据库管理系统

2.2.1 读写分离类型 SDBMS 架构

在传统数据库管理系统中，记录和索引的随机读写问题往往会成为性能瓶颈。对于记录的随机读写操作，往往可以通过使用固态硬盘（Solid State Drives, SSD）或是实现数据库内置缓存来缓解此类问题。其中，使用固态硬盘得益于它不再需

要磁盘寻道时间的硬件特点，可以良好的解决记录随机读取的问题，但是由于写入放大等问题难以支持记录的随机写入；实现数据库内置缓存机制，如 Oracle 数据库系统中的 Buffer Cache[36]、MySQL 数据库中的 Buffer Pool[37]，基于最近被修改的页面在未来极有可能被再次使用这样的假设，将被修改的页面存储在内存中，直到被换出时才写回磁盘。这样的方式虽然可以缓解随机读写问题，不过缓存系统的设计实现往往伴随着内存管理、数据一致性、数据恢复等复杂的实现问题。此外缓存机制还面临着冷启动的问题，即在数据库刚启动阶段，系统的性能会非常的差，需要将数据库访问请求流量逐步切入系统。对于数据库系统中索引的随机读写操作问题，由于数据库的索引结构往往通过 B+树来实现。在 B+树的场景中，对数据进行搜索、插入、删除操作都需要$O(log_B N)$次的磁盘读写，其中 B 表示一个页面包含的键值数，N 表示键值总数。即使假设 B 树中的非叶子节长期维护在内存中，但叶子节点的读取也至少需要一次磁盘读写操作。普通磁盘每秒仅能支持几百次的读写，如传统的 SAS 盘提供的最大 IOPS 不超过 300，在索引表十分庞大的场景下，每秒仅能对索引进行几百次的操作，无疑降低了系统的性能。

可扩展数据库管理系统是一个以 LSM-Tree（Log Structured Merge Tree）为基础的数据库管理系统。LSM-Tree 是一种高效的索引结构，它通过将数据的读操作和写操作分离的方式，有效的规避了磁盘的随机写入问题，付出的代价是在读取的时候可能会访问较多的磁盘文件。但由于固态硬盘等新型硬件设备的出现，能够更好的支持文件在辅助存储进行读取操作。LSM-Tree 的核心思想是将写操作的修改增量维护在内存中，当增量数据大小达到设定值后将修改增量批量地写入磁盘中，通过批量写操作替换对磁盘的随机写操作，达到优化写性能的目的；对于读操作，需要将磁盘中的历史数据和内存中的修改操作进行合并，得到融合数据后才可以返回给客户端。LSM-Tree 的结构模型以及合并过程如图 2.1 所示，历史数据被存储在 C_1 组件中，通常在磁盘上；对数据的最新修改被记录在 C_0 中，通常维护在内存里。用户的写操作被记录在 C_0 中，而读操作需要将 C_1 与 C_0 的数据进行融合后才可以返回。系统每运行一段时间后，需要发起合并的流程，将

C_0 上的修改与 C_1 数据进行合并，形成新的历史数据，C_0 的大小减小，继续接受后续新的写入。

由于 LSM-Tree 的良好特性，近年来在工程上进行了大量的相关实践与应用，相关系得到了大量关注，如 BigTable[38],LevelDB[39],OceanBase[40],CEDAR 等系统的实现都有借鉴它的思想。

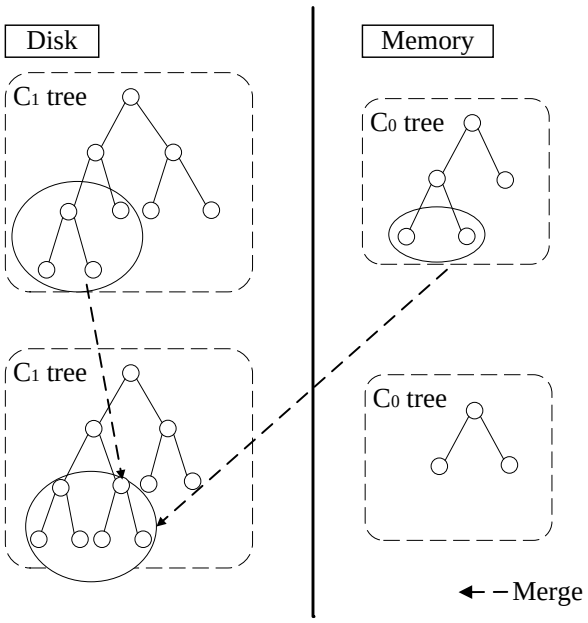

图 2.1 LSM-Tree 结构模型

为了后续讨论的方便，本文将可扩展数据库管理系统简称为 SDBMS (Scalable Database Management System)。SDBMS 将旧的磁盘数据（通常也称作基线数据）按照特定的规则，如按照主键值范围，划分成若干数据分片，冗余地存储在不同的服务节点中；将新增或修改的内存数据（通常也称作增量数据）单独存储在一个内存等硬件配置较高的服务节点中。在数据的更新量远小于总数据量的业务场景中，SDBMS 可以将基线数据以数据分片的形式冗余存储在不同的节点的磁盘中，将更新操作以 B+树的形式存储在一个节点的内存中。SDBMS 运行了一段时间后（通常为一天），启动定时任务将内存中的增量数据与不同节点中的基线数据进行合并。SDBMS 通过这样读写分离的形式，避免了上述的磁盘随机写问题，提供了可扩展的服务以及较好的读写性能，同时由于写操作被集中在一个服务节点上，也避免了分布式事务带来性能的问题。

根据功能的不同，SDBMS 的服务节点可以分为四种类型，如表 2.1 所示。

表 2.1　SDBMS 节点类型

节点名称	节点功能
MNode(Manager Node)	集群管理节点
TNode(Transaction Node)	事务处理节点
SNode(Storage Node)	基线数据存储节点
QNode(Query Node)	查询请求处理节点

各节点通过全连通网络相互协作，为来自客户端的各种请求提供服务。各类型的节点在网络中的结构如图 2.2 所示。

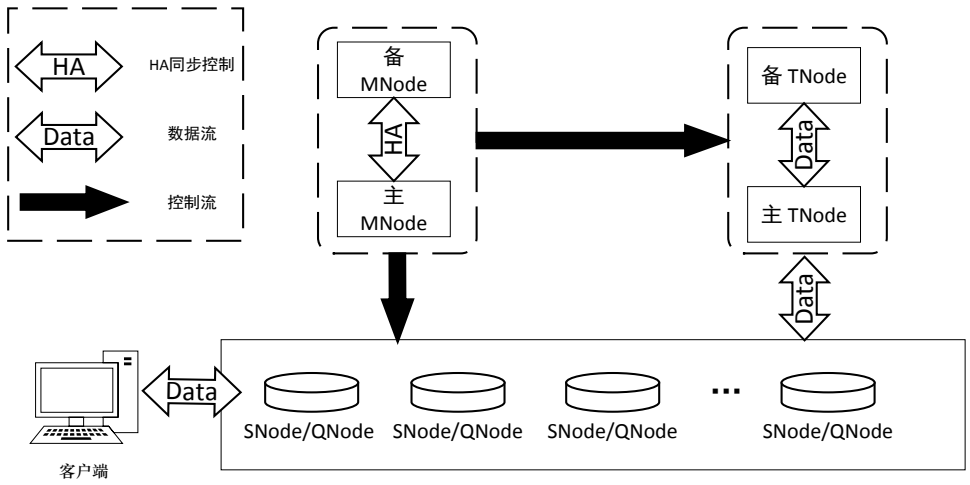

图 2.2　SDBMS 结构模型

MNode 是 SDBMS 的控制节点，维护了其他节点的状态信息，管理了数据分片的副本数量以及分布信息。由于 MNode 作为集群的控制节点，如果该节点发生故障，会导致整个系统处于不可用状态，因此在实际场景中，常常需要通过 HA(High Available)机制，部署一主一备来避免网络故障或节点宕机等意外造成集群不可用的状态。

TNode 是 SDBMS 的写事务处理节点，也就是存储增量数据的节点。如前所述，所有的增量数据都维护在该节点的内存中，由于所有的写事务都需要交由该节点进行处理，所以 TNode 会是集群中负载最为繁重的节点，通常选择部署在硬件配置更强劲的服务器上，同时保证一主一备的配置，根据实际业务对可靠性

和性能之间不同的取舍，主备之间可以选着不同级别的同步策略。

SNode 是 SDBMS 的基线数据存储节点，通常是将基线数据切片后分散冗余的存储在不同的 SNode 中，避免节点的硬盘故障造成的数据丢失或网络故障造成的节点不可访问使集群无法对用户提供服务。一般说来，基线数据有两或三份备份，也可通过参数进行设置。SNode 以自动请求 TNode 上的增量数据与本地的基线数据进行合并的方式对外提供读服务。SDBMS 通过定期的合并功能，将增量数据与基线数据进行融合，从而生成新的基线数据。

QNode 是 SDBMS 的查询请求处理节点，负责接收并处理客户端的读写请求。在系统接收到来自客户端的 SQL 请求后，该节点需要完成词法分析、语法分析、生成执行计划等一系列操作，并转发到相应的一个或多个 SNode 或者 TNode 中。如果需要读取的数据存储在多个 SNode 中，还需要对各个 SNode 返回的结果进行合并操作。

2.2.2 SDBMS 数据存储方式

如前所述，SDBMS 的数据在分为增量数据与静态数据两个部分，其中增量数据以 B+树的形式维护在 TNode 中，静态数据以数据分片的形式冗余的存储在多个 SNode 中。如图 2.3 所示，系统中有 5 个数据分片，每个数据分片以 3 个副本的形式均衡的存储在 4 个不同的 SNode 中，数据分片的位置信息被维护在 MNode 中，TNode 中存储了 5 个数据分片的增量更新数据。

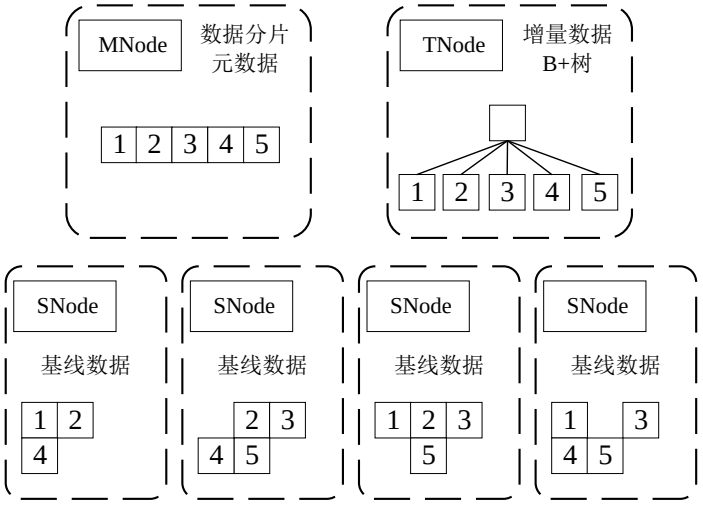

图 2.3 SDBMS 数据结构

SDBMS 需要定期将 TNode 中的增量数据与 SNode 中的基线数据进行合并，形成新的基线数据，具体过程是：首先，TNode 冻结目前活跃状态的内存表，转为冻结状态的内存表，同时开启新的活跃状态内存表，将后续的更新操作全都写入新内存表中。之后，TNode 告知 MNode 数据版本已经发生了改变，MNode 接收到消息后，通过检测 SNode 状态的心跳消息告知 SNode。最后，接收到消息的各个 SNode 启动定期合并流程，从 TNode 中获取各数据分片对应的增量数据。

以上，我们简要介绍了 SDBMS 中不同类型节点的数据结构，以及 SDBMS 发起定期合并的流程。由于上述的 SNode 以及 QNode 都部署在采购相对容易且成本较低的商用 PC 上，通常通过扩展 SNode 的数量就可以方便快速的扩展系统的存储容量，通过扩展 QNode 的数量，可以增加同一时间接入系统的用户数量。由于采用了读写分离的架构，所以对查询请求的处理相比于传统集中式的架构会略有不同，在下一个小节中介绍 SDBMS 查询请求的具体执行流程。

2.2.3 SDBMS 查询执行引擎

SDBMS 由于采用了读写分离的架构，相比于传统的集中式数据库管理系统，查询操作变得更加复杂。因为查询涉及了多节点间的数据传输以及基线数据与增量数据的融合问题。在 SDBMS 中，查询操作分为 4 个步骤：

1) QNode 接收客户端发送的请求，进行词法分析，语法解析，以及预处理后，生成相应的执行计划。

2) QNode 根据数据分片的分布信息，将读请求发送给各个 SNode。

3) SNode 根据查询信息，向 TNode 请求自身数据分片范围内数据的增量信息，并将得到的增量数据和自身的基线数据进行合并后，返回给 QNode。

4) QNode 对各个 SNode 返回的数据进行合并，进一步处理后，返回至客户端。

由于在 SDBMS 中，读写事务都涉及到了不同节点间网络的交互，节点之间的请求发送与数据传输造成的时延时不可忽视的。在之后查询性能的代价分析中，

节点间的网络时延将作为一个重点的分析对象。

2.2.4 SDBMS 查询算子

在 SDBMS 的查询操作中，涉及到许多不同的查询算子，它们各自完成不同的基础功能，通过相互的组合，共同完成用户复杂的查询请求。SDBMS 中常用的查询算子如表 2.2 所示。

表 2.2 SDBMS 常用查询算子

算子名称	算子功能
Get(K,R)	QNode 向 SNode 发起读请求，获取关系表 R 中主键值为 K 的记录。
Scan(Filter,R)	QNode 向 SNode 发起读请求，根据 Filter 过滤条件，获取满足过滤条件的数据。
Sort(ID,R)	QNode 对收取到的关系表 R 数据，根据 ID 列值，进行排序操作。
Join(r,s)	QNode 对关系表 R 的数据 r 与关系表 S 中的数据 s 进行连接操作。

其中，***Join***(R,S)表示的连接操作根据数据库系统内部的连接实现算法不同，又可以分为不同的连接算子，本文将在 4.2.1 小节中介绍这些不同的连接算法，这里仅是介绍相关符号代表的含义。

2.3 SDBMS 查询面临的挑战

可扩展数据库管理系统由于架构上的特殊性，数据分散的存储在不同的节点中，对数据的读取请求需要通过远程过程调用实现。而远程过程调用的返回时间通常在亚毫秒或毫秒级（0.1~1 毫秒），相比于单机环境下微秒级的返回速度，效率上有大约 100 倍的差距。所以如何减少无法产生查询结果集的数据在节点间的传输成为性能调优的一个关键。

另一方面，传统的数据库管理系统经过长期的发展，累积了大量的优化经验，这些经验在实际应用中体现在：首先，数据库管理系统本身集成了查询优化器，

系统可以根据一定的规则自动改写查询语句提升查询语句的性能；其次，经验丰富的系统管理人员通过修改查询语句，修改系统变量设置等手段来提升查询的性能。而可扩展数据库管理系统由于近年来才开始尝试应用在真实的企业环境中，目前可扩展数据库管理系统还不支持查询优化的相关功能，即系统对客户端发送的查询仅能生成简单且固定的执行计划，无法像传统的数据库管理系统那样收集相关的数据统计信息，预估查询的执行代价，并从生成的多个执行计划中选择执行代价最小的查询执行方式。

为了能够让业务系统的响应时延满足具体业务的要求，尽快完成业务的迁移工作，需要采用业务改造的方式，对业务系统中查询性能无法满足要求的查询操作进行改造。但正如前所述，可扩展数据库管理系统架构上具有区别于传统数据库管理系统的特殊性，使得传统数据库系统的查询优化方式无法直接使用到可扩展数据库管理系统中。本文为解决查询的响应时延问题，提取总结了查询性能无法满足要求的负载特点，对不同的负载场景提出了相应的解决方法，在第三、四章的内容中将详细的介绍这些方法。

2.4 本章小结

本章首先简要介绍了传统集中式数据库管理系统中查询的调优方式，接下来介绍了可扩展数据库管理系统的典型系统架构，可扩展数据库管理系统中的读写事务执行流程，以及查询操作涉及到的常用操作算子符号和含义。介绍了读写分离可扩展数据库管理系统通过增加节点的方式对系统的存储能力以及系统处理并发请求性能的可扩展性。同时也指出了由于查询执行处理上缺乏优化，虽然系统可以通过增加节点的方式提升对并发查询的支持，但是针对单个复杂查询的性能往往是低效的。通过分析，说明了读写分离架构的可扩展数据库管理系统在查询方面遇到的难点。在接下来的两章中，将针对这些影响查询性能的问题提出相应的解决策略，并通过代价分析与实验来说明改进性能尝试的有效性。

第3章 SDBMS 的存储结构调优

可扩展数据库管理系统由于它存储扩展能力强的特点，为了优化各类应用的性能，可以通过适当的冗余来达到提升性能的目的。本章首先介绍了为了适应一些分析类应用，可扩展数据库管理系统在表结构设计上的一些优化方式。之后介绍了在提升数据查找性能时发挥重要作用的二次索引相关内容，简介了它与传统数据库管理系统中索引的不同，并给出了可扩展数据库管理系统中构建索引的相关原则。最后，对可扩展数据库管理系统中数据分片的特点，提出了系统运行时需要遵循的两个原则。

3.1 表结构优化策略

由于可扩展数据库管理系统架构与执行流程的改变，使得可扩展数据库管理系统对多表连接的查询较为缓慢。本小节主要从表结构设计相关方面考虑，提出了提成业务性能的两个原则，分别是对实时性不敏感的多表连接查询优化以及连接查询时属性列的设计。这两种方式都是以冗余存储为代价的形式来获得更好的查询性能。

原则 T-1 对于不要求数据实时性的多表连接操作，创建加工表存储查询快照。

在真实的业务场景中，有的业务仅需要对一天之前甚至更早的历史数据进行符合业务逻辑的分析运算，这样的查询操作被称为不要求数据实时性的查询操作。对于这样的查询，可以设计加工表来存储查询快照。查询快照指的是查询产生的结果集，在多表的连接操作中，通过存储查询快照的方式，就可以用简单的加工表单表查询来替换复杂的多表连接操作。在业务系统中根据真实的业务需要来更新加工表中查询快照的内容，通过定时任务对查询快照进行定期更新，确保数据的正确性。

考虑真实系统中常见的一些分析类查询操作，如 SQL3-1 所示，其中 TABLE1 至 TABLE4 的表结构及含义以附录的形式给出，这里所述的查询旨在于分析一段时间内各业务的销售总额以及生产总成本。在这一查询中涉及到了 4 个数据表的连接操作，在查询的执行过程中，需要将多个表分散存储在各个 SNode 中的数据分片传输到处理查询的 QNode 中进行处理。当表的数据规模较大时，可以预见到数据分片在网络间的传输会是一大代价。

SQL 3-1 查询不同类型产品的销售总额

```
SELECT T1.SITE_NAME, T2.ARRIVE_TIME, T4.PRODUCT_ID,
    SUM(T3.EWB_CHARGE), SUM(T4.EWB_COST)
FROM TABLB2 AS T2
    JOIN TABLE1 AS T1 ON T2.SITE_ID = T1.SITE_ID
    JOIN TABLE3 AS T3 ON T2.EWB_NO = T3.EWB_NO
    JOIN TABLE4 AS T4 ON T2.EWB_NO = T4.EWB_NO
WHERE T2.ARRIVE_TIME <= VALUE1 AND T2.ARRIVE_TIME >= VALUE2
    AND T3.EWB_STATE = VALUE3
GROUP BY T4.PRODUCT_ID
```

然而这样的查询业务又有自身显著的特点，即查询对数据的实时性要求不高，通常仅需要查询前一天或是前几天的数据即可满足业务需求。为了能够提升这类查询对客户端的响应时间，可以通过设计加工表，冗余存储查询结果来达到业务的需求。具体地对于 SQL3-1 所示的查询任务，可以通过设计如表 3.1 所示的加工表用来缓存查询的结果集。通过定时任务的方式，定时地执行相关查询，将结果导入到加工表中，当客户端发起查询请求时，不再通过数据表进行多表连接操作来获得结果，而是直接从加工表中获取相关的查询结果。用单表查询替换了多表连接的方式来提升业务查询的性能。

表 3.1　加工表示例

列名	列含义	是否主键
SITE_NAME	站点编号	是
ARRIVE_TIME	日期	是
PRODUCT_ID	站点类型	否
SUM_CHARGE	产品累计营业额	否
SUM_COST	产品累计运输成本	否

原则 T-2：当连接操作在多个属性列上等价时，选择整形等简单数据类型作为连接条件。

类似于 Decimal，Varchar 数据类型的比较操作相较于整型等数据类型通常都比较耗时。所以在设计库表结构时，可以通过冗余列信息等方式尽量避免查询操作在这些数据类型上进行连接。

例如，Table 1 与 Table 2 在站点编号属性列上进行连接操作，即执行 SQL3-2 所示查询，需要得到订单的站点名称，站点所属地区等信息，就可以在站点编号这样 INT 数据类型的属性列上进行连接操作。

SQL 3-2 连接操作中连接列的选择 1

```
SELECT * FROM TABLE1
    JOIN TABLE2 ON TABLE1.SITE_ID = TABLE2.SITE_ID
WHERE ARRIVE_TIME < VALUE1
```

除了上述查询之外，还可以在 TABLE 2 中存储站点名称，使用 SQL3-3 所示的查询语句进行查询得到相关信息。但是由于站点名称属性列是 Varchar 数据类型，相比于整型在比较操作上的计算代价更大，所以在数据量大的场景下使用 Table 2 的表结构设计和 SQL3-2 的查询语句更加合适。

SQL 3-3 连接操作中连接列的选择 2

```
SELECT * FROM TABLE1
    JOIN TABLE2 ON TABLE1.SITE_NAME = TABLE2.SITE_NAME
WHERE ARRIVE_TIME < VALUE1
```

3.2 索引优化

索引是提高非主键查询的有效方式，本小节着重介绍了可扩展数据库管理系统中索引与传统集中式数据库管理系统中索引的区别，阐述了索引中回表与不回表的概念，分析并验证了不同类型查询使用索引的效率，同时给出了在可扩展数据库管理系统的查询中构建索引的建议。通过对存储结构中索引的优化，可以得到的益处是迅速提升查询中读取数据的响应时延，由此付出的代价是为了维护索引的一致性，牺牲了一定的更新性能。

3.2.1 SDBMS 索引接口

在 SDBMS 的存储结构优化中一个重要步骤是设计一个合理有效的数据结构来保证 QNode 能够快速准确的从 SNode 获取需要的数据。在涉及非主键查询的场景下，通常通过构建二次索引来实现数据快速准确的读取。文献[41]描述了 SDBMS 的二次索引的设计与实现，为了满足数据的读写功能，二次索引应该提供如表 3.2 所示的读写接口用于维护索引表与数据表间数据的一致性和提供对索引表的快速读取。

表 3.2　SDBMS 二次索引功能接口

接口名称	接口功能
Insert(Row)	将 Row 的值原子地插入数据表与索引表
Update(Column, Key)	更新数据表中主键为 Key 记录的 Column 列并同步到索引表中
Delete(Key)	删除数据表中主键为 Key 的记录并同步到索引表中
Get(Key, R)	读取数据表 R 的索引表中主键为 Key 的记录
Scan(Filter, R)	根据 Filter 条件读取数据表 R 的索引表中所有主键满足条件的记录

其中 ***Insert***(Row)、***Update***(Column, Key)和 ***Delete***(Key)三种操作是用来维护数据表与索引表一致需要使用到的写操作，不属于本文的讨论范围。在读写分离的可扩展数据库管理系统中，当索引处于可用状态时，针对数据的查询可以分为回表数据查询和不回表数据查询两种。回表数据查询指的是用户根据查询条件对

索引表进行查询后，还需要根据从索引表获得的数据表主键信息，对数据表进行数据查询，才能得到用户想要的数据。不回表数据查询指的是用户发起的查询仅需要访问数据表就可以得到结果，直接返回给用户。无论是回表数据查询还是不回表数据查询，又可以根据提供的条件不同分为带关键字的记录查找以及带过滤条件的范围查询两种方式。带关键字的记录查询是通过 ***Get***(key, R)算子来实现的，含义是查找索引表 R 主键列中值为 key 的记录，返回记录信息或返回不存在。带过滤条件的范围查询被定义为 ***Scan***(Filter, R)，含义是查找索引表 R 主键列中满足 Filter 条件的记录，其中 Filter 条件通常为一个不等式，最终返回所有满足过滤条件的记录或返回空集。

3.2.2 SDBMS 索引性能分析

通过建立索引的方式，避免了 SDBMS 对非主键列进行读取操作时的全表扫描，是从存储结构层面提升查询效率的重要手段，针对二次索引的改造问题，本文提出两个改造原则。

原则 I-1：当查询的过滤条件涉及多个属性列时，在选择区分率最高的列上构建二次索引。

所谓选择区分率指的是属性列上不重复值个数与记录总数的比值，它的取值范围在（0，1）之间，计算的比值越接近于 1 表明该列数据的选择性越强，越接近于 0 表明该列数据的选择性越弱。

例如考虑查询 SQL3-4，它的含义是根据外界输入的 VALUE1，VALUE2，VALUE3,选出寄送时间在 VALUE1，VALUE2 之间，这里假设为一天的时间间隔，运单状态为 VALUE3，这里假设是已签收状态的运单。即需要查询某一天时间内已签收状态的所有物流运单。

SQL 3-4 多条件的单表查询

```
SELECT EWB_NO FROM TABLE3
WHERE EWB_DATE <= VALUE1 AND EWB_DATE >= VALUE2
       AND EWB_STATE = VALUE3;
```

在本例中，如原则中假设的场景所述，查询的过滤条件涉及了多个属性列，在考虑这个查询的索引构建时，既可以选择在 EWB_DATE 属性列上构建索引，也可以选择在 EWB_STATE 属性列上构建索引。具体选择在哪一个属性列上构建索引可以根据 SQL 查询得到的结果进行判断。其中对 EWB_DATE 属性列发起的查询如 SQL3-5 所示

SQL 3-5 EWB_DATE 属性的选择区分率查询

```
SELECT DISTINCT(EWB_DATE)/COUNT(EWB_NO)
FROM TABLE3;
```

得到 SQL3-5 查询返回的结果，同时对 EWB_STATE 属性列发起的查询如 SQL3-6 所示。比较 SQL3-5 与 SQL3-6 所得结果的大小。

SQL 3-6 EWB_STATE 属性的选择区分率查询

```
SELECT DISTINCT(EWB_STATE)/COUNT(EWB_NO)
FROM TABLE3;
```

根据原则描述，应该选择在 SQL3-5 与 SQL3-6 返回结果中数值较大对应的属性列上构建索引。在运单量大的场景下，由于运单状态仅有 4 个值，所以查询结果会更加接近于 0，因此，应该选择在 EWB_DATE 属性列上构建索引来提升查询效率。

原则 I-2：构建索引时，根据查询的具体投影内容，在索引表中冗余查询需要返回的属性列。

如图 3.1 所述，索引的查询可以分为回表查询与不回表查询，其中回表查询在 SDBMS 中一共有 6 个步骤：

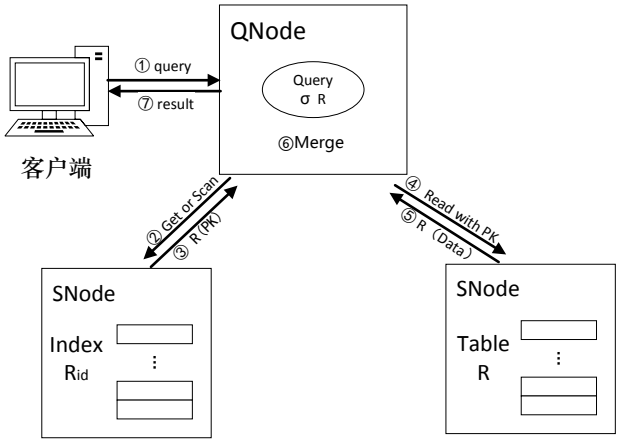

图 3.1　索引回表查询流程

1) QNode 接收到来自客户端的查询请求，通过词法语法解析，判断在查询条件属性列上是否有索引可用，如果有多个索引可用的场景下，还需要判断使用哪一个索引。确定了是否使用索引的相关信息之后，生成相应的执行计划。

2) QNode 根据执行计划中查询条件是等值查询或是范围查询的区别，向存储索引表的各个 SNode 发起 ***Get***(Key,Index R) 请求，或 ***Scan***(Filter,Index R) 请求。

3) SNode 将 ***Get***(Key, Index R) 或是 ***Scan***(Filter, Index R) 读取的索引表数据返回给 QNode。这里返回的索引表数据包含了原数据表的主键信息。

4) QNode 根据索引表返回的数据表主键信息，向存储有数据表的 SNode 发起 ***Get***(Key, R) 或是 ***Scan***(Filter, R) 查询请求，获取满足条件的数据表数据。

5) 各 SNode 将满足条件的数据表数据返回给 QNode。特别地，如果是范围查询请求，QNode 需要将各个 SNode 返回的数据进行合并，形成完整的数据。

6) QNode 将完整的数据返回给客户端。

为了更加详尽的描述是否回表对查询产生的不同执行代价，我们用如表 3.3 所示的符号来表示对关系表 R 的索引查询各个阶段的性能代价：

表 3.3　索引代价分析符号

符号	含义
type	不同的数据获取方式，有 get 和 scan 两种类型
CNLatency(R, type)	关系表 R 不同 type 获取的数据返回客户端的网络代价
DLatency(d_i, type)	SNode 不同 type 获取关系表 R 的 d_i 数据分片时磁盘 IO 代价
NNLatency(d_i, type)	关系表 R 不同 type 获取数据分片 d_i 从 SNode 传输至 QNode 的网络代价

由此，在 SDBMS 的回表数据查询场景下，根据主键进行等值查询时延可以表示为：

$$Query_{\text{get\_latency}} = 2 * \left(Max\left(DLatency(d_i, get)\right) + Max\left(NNLantecy(d_i, get)\right)\right) + CNlatency(R, \text{get})$$

回表数据查询场景下，根据主键进行范围查询需要使用***Scan***(Filter, R)算子，相比于使用 ***Get***(K, R)算子，一方面需要扫描的数据分片d_i会更多，这意味着在 SNode 中读取数据分片的磁盘 IO 代价会更大；另一方面，通常情况下满足Filter 条件的数据会多于满足等值条件的数据，这意味着网络传输时延会增大。总的说来，根据主键进行范围查询时延$Query_{scan_latency}$可以表示为：

$$Query_{scan\_latency} = 2 * \left(Max\left(DNLatency(d_i, scan)\right) + Max\left(NNLatency(d_i, scan)\right)\right) + CNLatency(R, scan)$$

在索引表需要回表的查询处理中，由于索引表中没有存储查询需要的所有列数据信息，所以查询的处理需要先对索引表进行查询，以此来获得数据表的主键信息，再通过主键信息对原数据表进行查询，获取到用户期望获取的数据信息。但是索引表可以通过冗余存储（Storing）额外的列信息，使查询避免回表操作。不需要进行回表操作的索引查询处理步骤如图 3.2 所示，有 4 个步骤：

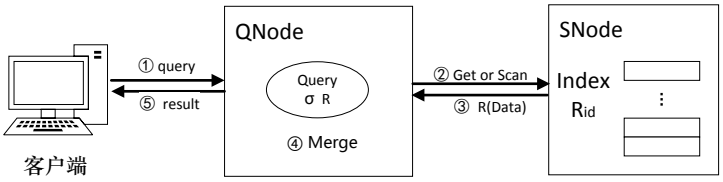

图 3.2　索引不回表查询流程

1) QNode 接收到来自客户端的查询请求，通过词法语法解析，判断在查询条件属性列上是否有索引可用，如果有多个索引可用的场景下，还需要判断使用哪一个索引。确定了使用索引的相关信息后，产生对应的执行计划。

2) QNode 根据执行计划中查询条件是等值查询或是范围查询的不同，向存储索引表的各个 SNode 发起 ***Get***(Key,Index R) 请求，或 ***Scan***(Filter,Index R) 请求。

3) SNode 将 ***Get***(Key, Index R) 或是 ***Scan***(Filter, Index R) 读取的索引表数据返回给 QNode。特别地，如果是范围查询请求，QNode 需要将各个 SNode 返回的数据分片进行合并，形成完整的数据。

4) QNode 将完整的数据返回给客户端。

不回表数据查询场景下，根据主键等值查询时延$Query_{get_latency}$可以表示为：

$$Query_{get\_latency} = Max\big(NNLatency(d_i, get)\big) + Max\big(DLatency(d_i, get)\big) + CNLatency(R, get)$$

不回表数据查询场景下，根据主键范围查询时延与回表数据查询的场景相似，使用 ***Scan***(Filter, R)算子时，相比于 ***Get***(K, R)算子，磁盘 IO 与网络时延会增大，总的传输时延$Query_{scan_latency}$可以表示为：

$$Query_{scan\_latency} = Max\big(NNLatency(d_i, scan)\big) + Max\big(DLatency(d_i, scan)\big) + CNLatency(R, scan)$$

通过代价组成可以看出，相比于回表数据查询，不回表数据查询减少了一次数据分片的磁盘 IO 代价和满足查询条件的数据网络传输代价。在 SDBMS 中认

为，节点间的网络交互是查询中不可忽略的一大代价，所以通过冗余存储列的方式，可以进一步提升索引的查询性能。虽然冗余存储的代价是花费更多的磁盘存储空间，但是由于 SDBMS 的架构特点，我们可以认为磁盘的扩展是廉价且容易的，因为仅需要增加 SNode 节点的数量，就可以快速的扩展系统的存储能力。所以在 SDBMS 中，可以通过冗余存储的方式可以进一步提升索引的查询性能。

例如，考虑如 SQL3-7 的查询，需要根据寄件日期查询某一天订单流水中寄件人与收件人的信息。

SQL 3-7 寄件人收件人信息查询

```
SELECT EWB_DATE,SEND_INFO,RECEIVE_INFO
FROM TABLE3
WHERE EWB_DATE <= VALUE1 AND EWB_DATE >= VALUE2;
```

为了加快查找的效率，需要考虑在 EWB_DATE 属性列上构建索引，其中方案一，如 SQL3-8 所示，仅保存了 EWB_DATE 属性列的信息，为了确保主键的唯一性，SDBMS 会在自动的将 EWB_DATE 与数据表的主键列 EWB_NO 属性列形成联合主键。所以索引表中一共有 EWB_DATE 属性列和 EWB_NO 属性列两列。

SQL 3-8 不带冗余的索引构建

```
CREATE INDEX EWB_DATE_INDEX ON TABLE3(EWB_DATE);
```

还可以使用方案二，如 SQL3-9 所示，不仅存储了 EWB_DATE 属性列和 EWB_NO 属性列作为联合主键列，同时，还通过缓存的方式，冗余存储了 SEND_INFO 属性列以及 RECEIVE_INFO 属性列。

SQL 3-9 带冗余的索引构建

```
CREATE INDEX EWB_DATE_INDEX ON TABLE3(EWB_DATE)
STORING (SEND_INFO,RECEIVE_INFO);
```

相比于方案一没有冗余存储的方式，方案二所示的索引存储方案由于存储了寄件人以及收件人的信息，所以在查询的时候可以避免回表的操作，在查询中获得更好的性能。

3.2.3 SDBMS 索引性能实验

为了验证二次索引对优化查询的有效性，本文设计了相关实验来说明二次索引对系统更新性能的影响以及对查询性能的提升。本文的索引实验在类 SDBMS 的分布式数据库 CEDAR 上进行，CEDAR 系统中的节点可以分为 RootServer，UpdateServer，ChunkServer 和 MergeServer 四种节点类型，它们与 SDBMS 系统中的节点类型存在着一一对应的关系。RootServer 是 CEDAR 系统的主控进程，管理着其它进程的状态等元数据信息，对应 SDBMS 中介绍的 MNode 节点。UpdateServer 是 CEDAR 系统中的更新事务处理进程，存储着系统的增量数据，对应着 SDBMS 中介绍的 TNode。ChunkServer 是 CEDAR 系统中的基线数据存储进程，对应着 SDBMS 中介绍的 SNode。MergeServer 是 CEDAR 系统中对外提供服务的进程，负责处理用户发送的 SQL 请求，对应着 SDBMS 系统中的 QNode 节点。

本文准备了 8 台位于同一机房的商用服务器来部署集群，部署一个 RootServer 节点与一个 UpdateServer 节点在同一商用服务器上，在其余七台服务器中每台服务器上都部署一个 ChunkServer 节点和一个 MergeServer 节点。所有的服务器配置均相同，服务器间采用千兆网络连接，节点间的网络传输速率在 110MB/s 至 120MB/s 之间。每台服务器的操作系统是操作系统是 Red Hat Enterprise Linux Server release6.2，硬件环境上每台服务器配置了 2 颗 Intel(R) Xeon E5-2650 CPU，核心的频率是 2.3GHZ，同时还配备了 500G 的内存以及 3TB 的固态硬盘。

实验采用了开源的 Sysbench 性能测试工具[42]进行数据的插入性能测试，Sysbench 测试工具可以根据用户的不同设置，生成不同大小的数据表，并返回插入数据的耗时，其中 Sysbench 测试所需要使用的数据表表结构如附录中 SBTEST 表所示。

实验 3-1：索引个数对系统更新性能影响测试。实验中分别对产生的关系表 sbtable1 在属性 K 上建立了 0 至 9 个索引，测试了插入 100 万条记录时，需要的总时间对比，每组实验重复了三次，取处理时间的平均值。实验结果如图 3.3 所

示。

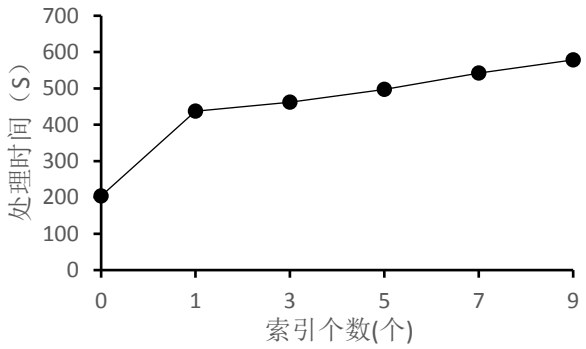

图 3.3　索引个数对更新性能的影响

由实验结果可以看出，是否建立索引，对更新的性能产生影响，当数据表上无索引时，更新 100 万条记录需要 203 秒，当建立了一个索引后，更新 100 万条记录需要 437 秒，之后每增加两个索引，更新 100 万条记录的处理时间大约增加 30 秒。

实验 3-2：索引范围查询性能对比。实验对一张有 1000 万条记录的关系表 sbtable2,在 K 列上分别建立了不冗余属性 C 列的索引 index1，和冗余属性 C 列的索引 index2，通过 SQL3-10 语句来测试范围查询返回数据量 5 大小与处理时间的变化。

SQL3-10 索引范围查询

```
SELECT /*+INDEX(INDEX_NAME)*/ID,K,C
    FROM SBTEST2
WHERE K >= COND1 AND K <= COND2
```

通过改变 hint 语句中 INDEX_NAME 的不同值来确定查询是否需要进行回表操作。当 INDEX_NAME 为 index1 时，需要进行回表查询;当 INDEX_NAME 为 index2 时，不需要进行回表查询。通过控制 COND1 和 COND2 的大小，来获取不同大小的结果集。实验结果如图 3.4 所示：

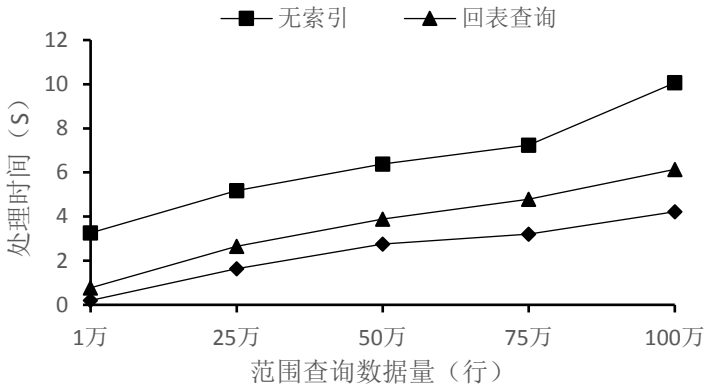

图 3.4　索引范围查询性能对比

由实验结果可以看出，当获取的数据量越多时，查询所需要的时间越多，这是由于一方面，数据量增加，需要计算判断的值增多；另一方面，返回的数据量增大，使得网络的传输时延增加。相比较于无索引的场景，通过构建索引，能够提升数据的查询效率。同时，如原则 I-2 所述，由于不回表的查询能够进一步加快查询的处理效率，所以在索引的设计过程中，应该考虑存储一定的冗余列，尽可能的避免需要回表操作的索引查询，这样可以进一步的提升查询性能。

3.3 SDBMS 定期合并优化策略

在 SDBMS 中，静态数据以数据分片的形式冗余存储在 SNode 中，数据分片的大小会被设置为系统的默认值或是在系统管理人员创建关系表时特别指定。通常系统将数据分片的大小默认设置为 256MB。当数据分片分布式地存储在多个 SNode 中，对于客户端发送的数据查询请求可以被 QNode 拆分为多个子请求，QNode 再通过并发的方式向各个 SNode 发送数据读取请求，各 SNode 向发起请求的 QNode 返回部分结果集，QNode 将接收到的所有结果集进行合并后返回给客户端。如图 3.5 所示，表示了在并发请求的场景下，数据返回的流程。整个并发查询的请求执行流程如下所述：

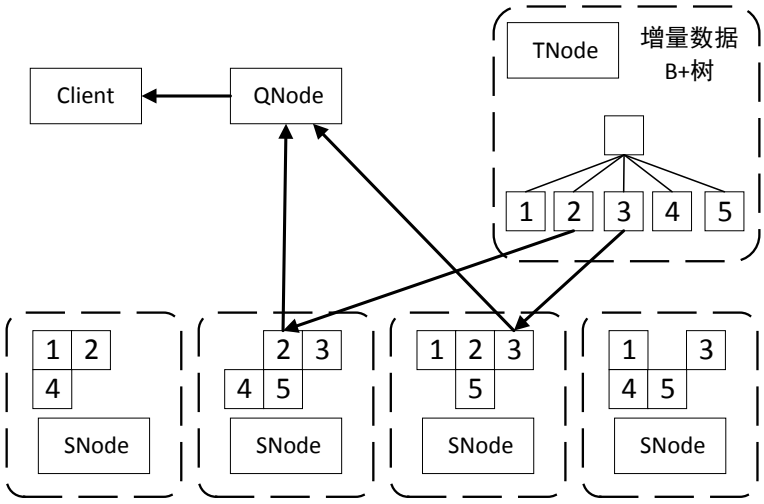

图 3.5　并发查询数据返回流程

1) QNode 接收到客户端发送的查询语句，进行语法解析，生成执行计划，根据本地缓存的相关数据分片位置信息获取可以读取数据的 SNode 地址。

2) 如果发现需要读取的数据全部位于一个数据分片内，直接将读取请求发送给相关的 SNode 执行；如果查询请求涉及多个数据分片，将查询的读取请求划分成多个子请求，划分的依据是每个子请求对应着一个数据片，将所有的查询请求并发的发送给各个 SNode。QNode 等待从各个 SNode 中返回的数据。

3) SNode 执行接收到的子请求，并根据传过来的过滤条件，筛选出符合条件的数据向 TNode 发起增量数据的读取请求并等待 TNode 返回增量数据。

4) TNode 将数据返回至 SNode 后，SNode 将增量数据与基线数据进行融合，并向 QNode 发送子请求的查询结果。

QNode 接收到所有的自请求结果，并进行合并操作，得到完整的数据，再对数据根据查询需要，进行对应的排序或聚合操作，将处理完成的数据返回至客户端。从流程中可以看出，相比于串行地向一个节点请求数据，并发地向不同节点发起查询请求可以提升系统的查询性能。

在如 3.1 节中所述的一些业务系统中，会产生某一时间区间内，系统接收了大量其他系统中导入的数据，由于系统还没有发生定期合并的流程，此时这些数据都被维护在 TNode 中，没有在 SNode 中形成相应的数据分片。这时，系统中接收到的查询请求，大量的数据需要从 TNode 中获取，使得大量的数据需要经过 TNode 与 SNode 间的网络传输。对于这样的场景如果可以在定时地导入数据操作后，先发起定期合并请求，使维护在 TNode 中的增量数据批量的合并到 SNode 中，形成相应的数据分片。通过并发查询的方式，可以达到提升系统查询性能的效果。对于上述原理，可以得到原则 P-1：

原则 P-1：当 SDBMS 进行数据导入操作后，应该先发起定时合并任务，待合并完成，数据以数据分片形式存储在 SNode 中之后，再接受业务的查询操作。

3.4 本章小结

本章描述了 SDBMS 在调优过程中与存储有关的内容，首先说明了由于 SDBMS 存储结构良好的扩展性，在库表结构的模型设计中，采用一些反范式的设计，通过冗余存储可以提升查询性能的策略。之后介绍了在 SDBMS 中对查询性能提升有重要作用的二次索引有关内容，介绍了它与传统数据库管理系统中二次索引的异同，构建流程，以及在查询中索引的回表查询与不回表查询的异同，给出了在 SDBMS 中构建索引的建议。最后针对 SDBMD 中数据是以数据分片冗余存储在存储节点中的特点，介绍了定期合并以及数据分片分裂对查询性能影响的相关内容。

第4章 SDBMS 的查询调优

可扩展数据库管理系统中查询性能较低的一个重要原因是数据分散地存储不同的节点中，查询处理节点需要通过远程过程调用向存储节点请求数据。为了克服这类架构系统在查询上性能较低的难题，本章介绍了查询过程中关于表达式计算的相关优化方案，以及连接这一常见的查询操作在可扩展数据库管理系统中的优化方案。给出了相应的性能分析以及优化原则。

4.1 表达式计算优化

原则 E-1：在日期类型的数据列上对输入的字符串日期进行过滤查询时，调用转换表达式进行提前计算。

转换表达式指的是 *to_timestamp()*、*to_date()*、*add_day()*等常用的一些函数操作，它们的功能是将用户输入的字符串类型日期转化为数据库系统内部的存储格式，或是对传入的日期字符串进行一些日期加减的计算操作。在类 SDBMS 的 CEDAR 数据库管理系统中，虽然支持这类函数的操作，但是实现逻辑较为简单，存在着优化的空间。

4.1.1 表达式计算优化场景

在实际的业务场景中，一种常见的业务需求是客户端向数据库发起请求，希望获得输入的时间区间内一个产品类型的交易总金额。具体的查询如 SQL 4-1 所示。

SQL 4-1 不同产品类型销售总额查询

```
SELECT SUM(A.EWB_CHARGE), B.PRODUCT_ID
    FROM TABLE3 AS A JOIN TABLE4 AS B ON A.EWB_NO = B.EWB_NO
WHERE A.EWB_DATE >= VALUE1 AND A.EWB_DATE <= VALUE2
GROUP BY B.PRODUCT_ID;
```

查询语句中的 value1 与 value2 为实际应用系统中由用户输入的日期，通常是一天的时间间隔，数据类型为字符串类型。在 DBMS 中，DATE 类型的数据通常是将年月日信息通过一定的计算规则，转换为 64 位的整形进行存储。在 CEDAR 数据库系统中如果客户端发起的查询请求中传入的 VALUE 值是以字符串信息传入的日期值，那么在记录的查找过程中，对于每一条记录都要对 VALUE 值进行一次字符串类型到日期类型的转换，这样大量的重复计算无疑严重地影响了系统的查询性能。*to_date()*函数为 DBMS 中实现的一个功能函数，目的是将传进来的值从字符串类型转化成相应的 DATE 类型，进行后续的比较操作。所以对于 SQL4-1 对应的查询操作可以改造成为如 SQL4-2 所示的查询操作。

SQL 4-2 使用日期函数优化不同产品类型销售总额查询

```
SELECT SUM(A.EWB_CHARGE), B.PRODUCT_ID
    FROM TABLE3 AS A JOIN TABLE4 AS B ON A.EWB_NO = B.EWB_NO
WHERE A.EWB_DATE >= to_date(VALUE1) AND
       A.EWB_DATE <= to_date(VALUE2)
GROUP BY B.PRODUCT_ID
```

SQL4-2 查询的处理逻辑是 *to_date()*作为一个系统函数，集成实现在系统中，当系统接收到查询请求时，调用 *to_date()*函数将传入的字符串参数通过计算转换成与之对应的整形值，即时间戳类型。直接使用时间戳构造过滤条件，对数据进行查询，期间对输入的字符串类型数据仅进行一次计算。相比于 SQL4-1 所示的查询，通过减少重复计算相同字符串类型数据到日期类型数据转换的方式，提升了这种场景下查询的效率。与此类似的还有 *to_date()*，*add_date()*等涉及日期运算的函数。基于上述原理，对于类似 SQL4-1 的查询类型，在应用改造的过程中，可以按照原则 E-1 所述对应用进行改造，提升查询性能。

4.1.2 表达式计算优化性能实验

本小节的实验使用了和 3.2.3 小节中相同的硬件和软件配置，使用了如表 4.1 所示的数据表格，通过导入实际业务中的部分数据来控制表的规模大小。使用 SQL4-1 所示的 SQL 语句对数据库进行查询，其中 value 的值分为直接传入字符

串数据类型的日期和调用 *to_date()*函数进行数据类型转换的日期两组对照，日期的选取范围是固定读取 1 天的交易，每组 SQL 的执行时间是 3 次测量后的平均值。实验结果如图 4.1 所示：

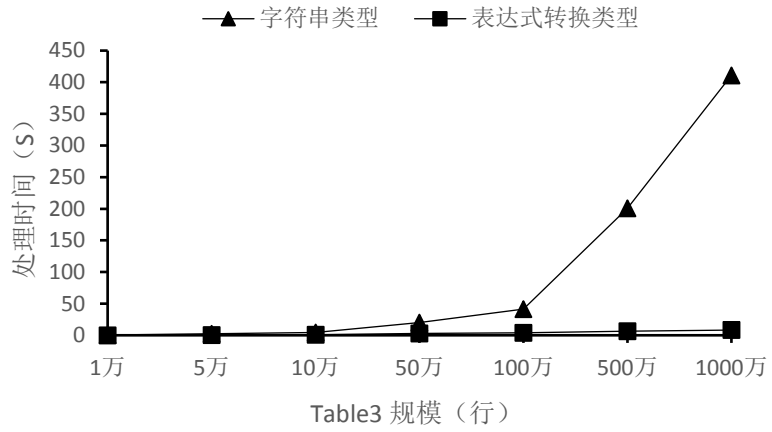

图 4.1　表达式优化性能测试结果

由实验结果可以看出，当查询的关系表数据量较小时，由于扫描全表的速度较快，所以是否进行原则 E-1 所述改造性能区别不大。当查询的关系表数据量不断增大的时候，通过调用类型转换函数的方式对性能的影响不大，由于表数据量的增大，查询时间会略微增长；未调用转换函数的查询语句性能受到了很大的影响。所以当查询的过滤条件涉及到选择日期类型的数据，建议按照原则 E-1 所述方式，对业务进行改造。

4.2 连接优化

连接操作时业务查询中极其常见的操作，本节对 SDBMS 不同数据场景下连接算法的选取做了总结和归纳，提出了相应的原则，并通过实验验证了所提出原则的正确性。

4.2.1 SDBMS 连接算法

在传统的数据库管理系统中处理连接操作时，主要有嵌套循环连接（Nested Loop Join），排序归并连接（Sort Merge Join），哈希连接（Hash Join）三种不同的算法[43]。类似的在 SDBMS 中，也可以实现一样的算法来处理连接操作，同

时，为了减少 SNode 与 QNode 间的数据传输，还可以采用半连接（Semi Join）算法来处理查询。本节中首先介绍了传统数据库连接操作中具有代表性的排序归并算法在 SDBMS 系统中的实现，通过用例的形式介绍了半连接算法的执行流程。

排序归并连接算法处理等值连接时，基于如果两个表在做连接操作前，在连接属性上就已经是有序的场景下，不再需要对其中的一个关系表进行多次扫描就可以得到连接结果集。以 SQL4-2 为例，它的具体执行流程是：首先分别读取两张经过排序处理关系表，假设读取 Table3 的第一条记录 t3 以及 Table4 的第一条记录 t4，如果它们符合连接条件，那么直接输出该行，并获取 Table4 的下一行，如果 t3 和 t4 不满足连接条件，则判断 t3 和 t4 的大小关系。如果 t3 小于 t4 则获取 Table3 的下一条记录，如果 t3 大于 t4 则获取 Table4 的下一行。如此往复循环，直到 Table3 或 Table4 全部读取结束。该算法在 SDBMS 中的简单执行流程如下代码所示：

算法 1. SDBMS 排序归并连接

```
Input：Table3,Table4
 1.  Sort(EWB_NO, Table3) and Sort(EWB_NO, Table4)
 2.  Get first row t3 from sorted Table3
 3.  Get first row t4 from sorted Table4
 4.  while not at the end of either sorted Table3， sorted Table4
 5.  do
 6.    if t3 and t4 join to make a tuple t then
 7.      Join(t3,t4) and OutPut t
 8.      Get next row from sorted Table4
 9.    else if t3 < t4
10.      Get next row from sorted Table3
11.    else
12.      Get next row from sorted Table4
13.    end if
14.  end while
```

QNode 接收到客户端的请求，通过词法语法解析，生成相应的执行计划。按照生成的执行计划，QNode 通过构造 ***Scan***(Filter, Table3)算子向 SNode 获取关系表 Table3 中满足过滤条件的记录以及关系表 Table4 完整数据分片。当 SNode 返回所有数据，QNode 将所有数据存储在本身节点的内存中之后，需要调用内存排序算法，根据关系表的连接属性进行升序排序，待排序完成后，可以按照如上所述的算法继续进行连接操作。

半连接算法最早在文献[44]中被提及。半连接算法的含义是通过减少数据传输来优化可扩展数据库管理系统中连接操作的查询性能。在 SDBMS 中，通常认为数据间的网络传输对算法的效率来说是不可忽视的代价之一。因此，在一个规模较小的关系表与一个规模超大的关系表进行连接操作，并且连接后产生的结果集很小的场景下，或是在一个规模超大，但是根据过滤条件进行选择操作后可以变为一个规模较小的临时表与一个规模超大的关系表进行连接操作的场景下进行自然连接或是内连接操作时，提升算法效率的方式。根据过滤方式的不同，可以分为基于 In 表达式的半连接算法，基于 Between 表达式的半连接算法[45]以及基于布隆过滤的半连接算法[46]。算法的思想在于，通过小表的数据特征对超大关系表进行过滤，减少超大规模数据表的网络传输，达到提升连接操作性能的效果。它在 SDBMS 中具体的执行流程如图 4.2 所示：

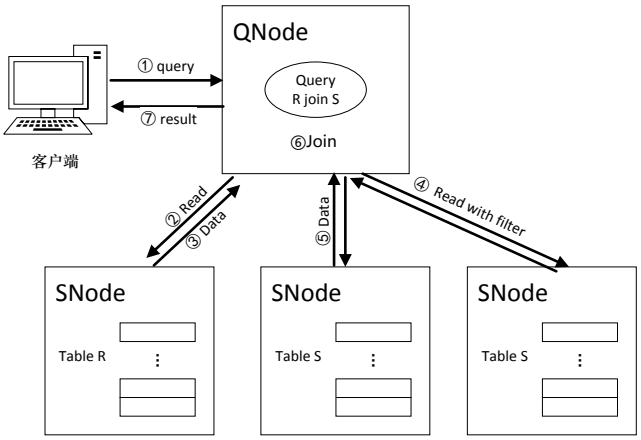

图 4.2　Semi Join 查询流程

1) QNode 接收到来自客户端的查询请求，并进行词法语法解析，生成相应的执行计划。

2) QNode 先通过 ***Scan***(Filter, Table3)算子向 SNode 发送读取请求，获得 Table3 中满足 Filter 表达式的记录，这里有可能需要向多个 SNode 发送请求，为了表示方便，仅画出向一个 SNode 请求数据作为示意。

3) QNode 接收到返回的 Table3 数据后，通过去重，构建出了 Table3 在连接属性 EWB_NO 所有出现的值，并根据数据量的大小，决定构造 In 表达式使用 ***Get***(K, Table4)算子精确过滤 Table4 中的数据；或是根据数据的最大值和最小值，构建 Between 表达式，使用 ***Scan***(Filter, Table4)算子通过范围查询 Table4 中的数据，其中 Filter 即为构建的 Between 表达式；或是通过构建布隆过滤器[47]，作为表达式，使用 ***Scan***(Filter, Table4)算子通过扫面全表，过滤 Table4 中的数据。

4) 将构建的表达式向各存储有 Table4 数据分片的 SNode 发起带过滤条件的 Table4 读取请求。

5) 各 SNode 向 QNode 返回满足过滤条件的数据。

6) 在 QNode 中执行排序归并连接算法获得连接结果集。

7) 向客户端返回正确的结果集。

在本小节中，我们介绍了传统集中式数据库管理系统中排序归并连接算法的执行流程，并简要的介绍了它们在 SDBMS 中的实现方式。之后介绍了在可扩展的场景下，由于网络传说成为了影响算法性能的重要代价之一，为了减少数据传输而提出的半连接算法的主要内容，并详细解释了它在 SDBMS 中的执行流程，在下一个小节中，将详细的分析各种连接算法的执行代价，并给出不同算法的适用场景。

4.2.2 SDBMS 连接性能分析

如前所述，SDBMS 中数据分散、冗余地存储在不同的节点中，在执行连接操作时，需要将数据从各节点传输至一个 QNode 节点的内存中，再根据不同的算法执行相应的内存连接算法。这里为了代价分析描述的方便，将 4.2.1 小节所

述用例 Table3 与 Table4 抽象地使用 R 与 S 进行表示，与 3.2.2 小节相似的是，本文使用表 4.2 中的符号来表示各项操作的代价。

表 4.1　连接操作代价分析符号

符号	含义
Size(R)	关系表 R 中共含有的记录数
CNLatency(R,S)	关系表 R 与关系表 S 连接操作结果集传输至客户端的网络代价
CPU(R,S)	关系表 R 与关系表 S 连接操作的计算代价
DLatency(d_i,R)	在 SNode 中读取关系表 R 的 d_i 数据分片时磁盘 IO 代价
NNLatency(d_i,R)	关系表 R 的数据分片 d_i 从 SNode 传输至 QNode 的网络代价

在 SDBMS 中，排序归并连接对于 R 与 S 的数据读取请求可以同时发出请求，之后再异步地等待数据返回，因此网络时延$Net_{lantency}$可以表示为：

$$Net_{lantency} = Max\big(NNLantency(d_i,R), NNLatency(d_i,S)\big) + CNLatency(R,S)$$

在 SNode 中，Table3 需要通过全表扫描的方式来读取满足过滤条件的数据，Table4 需要通过全表扫描将数据传输至 QNode，所以无论是 Table3 还是 Table4 都需要扫描读取磁盘时延$Disk_{latency}$可以表示为：

$$Disk_{latency} = Max\big(DLatency(d_i,R), DLatency(d_i,S)\big)$$

假设表 R 的规模是 m，表 S 的规模是 n，那么排序归并连接算法的时间复杂度可以表示为$O(m+n)$，但由于还有排序操作，可以认为排序的时间复杂度分别为$O(m \cdot logm)$与$O(n \cdot logn)$，所以排序归并连接的计算代价$CPU(R,S)$可以表示为：

$$\begin{aligned} CPU(R,S) \ &= O\big(SIZE(R) + SIZE(S)\big) + \\ &Max\big(SIZE(R) \cdot logSIZE(R), SIZE(S) \cdot logSIZE(S)\big) \end{aligned}$$

半连接算法的场景下，与上述三种方式不同，需要先请求读取关系表 R 中

的数据，构建出过滤条件后，才可以根据过滤条件请求读取关系表 S 中的数据。另一方面构建 In 表达式、构建 Between 表达式和构建布隆过滤器这三种不同的方式执行代价会有所不同，以下分别对三种不同的场景进行分析：

构建 In 表达式，使用 ***Get***(K,S)算子过滤关系表 S 的场景下，可以实现对数据的精确过滤，即通过过滤条件获取的关系表 S 的每一个元组都必然是可以与关系表 S 产生连接结果的元组。对于 in 表达式中网络时延$Net_{latency}$可以表示为：

$$Net_{latency} = \mathrm{Max}\big(NNLatency(d_i,R)\big) + Max\big(NNLatency(d_i,S_{In})\big) + CNLatency(R,S)$$

类似地，磁盘的时延$Disk_{latency}$可以表示为：

$$Disk_{latency} = Max\big(DLatency(d_i,R)\big) + Max\big(DLatency(d_i,S)\big)$$

QNode 端的连接操作如果选用排序归并连接，计算代价CPU(R,S)可以表示为：

$$\mathrm{CPU(R,S)} = O\big(SIZE(R) + SIZE(S_{In})\big) + Max\big(SIZE(R)\cdot logSIZE(R), SIZE(S_{In})\cdot logSIZE(S_{In})\big)$$

由于 In 表达式的精确过滤，在假设的场景下对于关系表 R 而言，仅需要传输少量的数据。相比于排序归并连接算法，数据的读取方式虽然从同时读取左右表的数据分片变为先读取左表数据，再读取右表数据。但是在左关系表通过查询条件过滤后不重复元组数很小而右关系表数据量极为庞大的场景中基于 In 表达式的半连接算法依旧能很好的提升查询的效率。这是因为在假设的场景中，认为传输代价占据了大部分的时长。在数据量较小的场景下，In 表达式能够发挥良好的性能，同时通过在右表的连接属性上构建二次索引等方式，可以进一步的加快过滤条件的查找速度，获得更好的查询性能。但是当过滤后左表元组数仍然较大的场景下，In 表达式的查询性能会成为瓶颈，即当 In 表达式中的过滤值超过一定个数时，查找效率迅速降低。同时在中小规模表场景下，表数据的传输往往比较迅速，基于 In 表达式的半连接操作由于不能同时返回待连接表的数据，而 In

表达式的过滤性不足的场景下，算法的性能会低于全数据传输的排序归并连接等算法。

针对 In 表达式在超过一定规模后查找效率迅速降低的问题，一种简单可行的方案是在读取左关系表数据的时候维护左关系表的最小值与最大值，通过构造的 Between（Min_value，Max_value）表达式使用 Scan(Filter,S)算子进行范围查询，在这样的场景中网络时延$Net_{latency}$可以表示为：

$$Net_{latency} = \mathrm{Max}\big(NNLatency(d_i, R)\big) + Max\big(NNLatency(d_i, S_{Between})\big) + CNLatency(R, S)$$

磁盘的时延表达式与 In 表达式中磁盘时延一致，在 QNode 端如果采用排序归并连接算法，计算代价CPU(R, S)可以表示为：

$$\mathrm{CPU(R,S)} = O\big(SIZE(R) + SIZE(S_{Between})\big) + Max\big(SIZE(R)\cdot logSIZE(R), SIZE(S_{Between})\cdot logSIZE(S_{Between})\big)$$

在构建 Between 表达式的半连接算法中，仅是通过属性上的最小值与最大值进行范围过滤，因此相比 In 表达式的半连接算法，传输的数据量会更大，即：

$$\mathrm{Size}(S_{Between}) \geq \mathrm{Size}(S_{In})$$

而且可能会存在左关系表的在连接属性上的取值范围大于等于右关系表在连接属性上的取值，即构建的 Between 表达式完全无法过滤右关系表的极端情况。但是在连接属性上的值恰好是递增有序，而左表根据查询的范围条件，在链接属性上的值又超过了 In 表达式的处理能力场景下基于 Between 表达式的半连接算法能够发挥出良好的性能。例如在实际应用场景需要查询金融系统或快递系统中一天的交易，进行一定的聚合分析，需要根据交易流水号或是快递单号这样具有递增属性的列对某两张关系表进行连接操作，如果每天的交易单数超过了 In 表达式的处理能力，使用 Between 表达式的半连接操作可以获得良好的性能。

虽然构建 Between 表达式的半连接算法可以很好的改善连接条件具有自增性质场景下的连接操作，但这不能概括真实场景中所有连接操作的特点，所以还

需要能够处理更连接属性列更普通、无规律的场景。解决这一问题的核心是，如何构建一个精简的数据结构解决中大规模数据量场景下，快速、有效地判断右关系表连接属性列上的值在左连接关系表中是否有相同的值与之对应。布隆过滤器正是这样一个精简的数据结构。通过获取左表所有符合查询过滤条件的元组信息，对连接列上的所有值进行多组哈希计算，构建布隆过滤器，并将这个布隆过滤器通过 QNode 以过滤表达式的形式传递给右关系表，对右关系表的每一个元组使用相同的哈希函数进行计算，判断每一个元组在左关系表是否有相同的值与它对应，如果有则返回该元组，如果没有则不返回该元组。对由于布隆过滤器假阳性的特征，即经过运算，如果判断右关系表中的某元组在左关系表的连接属性上不存在与之相等的元组，则一定不存在；如果判断有关系表中的某元组在左关系表的连接属性上存在与之相等的元组，那么及有可能存在与之相等的元组。由于布隆过滤器较低的误判性，我们可以认为使用布隆过滤的方式能够很好的减少右关系表中不满足连接条件的元组进行网络传输。类似的我们可以得到，基于布隆过滤的半连接算法，它的网络时延$Net_{latency}$是：

$$Net_{latency} = \mathrm{Max}\big(NNLatency(d_i, R)\big) + Max\big(NNLatency(d_i, S_{Bloom})\big) + CNLatency(R, S)$$

磁盘时延表达式与基于 in 表达式过滤的半连接算法相同，计算代价 CPU(R, S)可以表示为：

$$\mathrm{CPU(R,S)} = O\big(SIZE(R) + SIZE(S_{Bloom})\big) + Max\big(SIZE(R) \cdot logSIZE(R), SIZE(S_{Bloom}) \cdot logSIZE(S_{Bloom})\big)$$

在传输的数据规模上基于 In 表达式的半连接算法、基于 Between 表达式的半连接算法以及基于布隆过滤的半连接算法之间数据传输量大小关系如下：

$$\mathrm{Size}(S_{Between}) \geq \mathrm{Size}(S_{Bloom}) \geq \mathrm{Size}(S_{In})$$

以上介绍了三种基于不同表达式的半连接算法，从数据传输量上来看，基于 In 表达式的半连接算法由于获取的都是精确值，所以传输的数据量是最小的；基于布隆过滤的半连接算法由于布隆过滤器存在一定的误判性，但误判的可能往往

很小，所以传输的数据量会略大于基于 In 表达式的半连接算法；基于 Between 表达式的半连接算法往往只在连接属性上具有自增特点的场景下传输的数据量接近于其它两种算法，而大部分场景下，基于 Between 表达式的半连接算法传输的数据量会大于其它两种算法。当然，数据网络传输的时延仅是仅是半连接算法中的部分代价，基于不同表达式的半连接算法能否对右关系表能否进行快速查找也是一个不可忽视的条件。在这一问题上，基于 In 表达式的半连接算法可以通过在连接属性列上建立二次索引，采用如 3.2 节中介绍的 Get 方式进行等值查询，这样数据的查找代价往往是很低的，在小数据量的场景下能发挥极好的性能；同样的，基于 Between 表达式的半连接算法也可以在连接属性列上构建二次索引，通过如 3.2 节中介绍的 Scan 方式对索引进行范围查询；基于布隆过滤的半连接算法则只能通过全表扫描，对每一个元组进行哈希函数计算来判断这一元组是否应该被传输。所以在数据的查找代价上，基于布隆过滤的半连接算法代价往往会大于其它两种算法。这里总结关于选择连接算法的 4 条原则，如下：

原则 J-1：当排序归并的网络传输时延代价小于半连接的网络传输代价时，使用排序归并连接算法进行连接操作。

原则 J-2：当半连接的网络传输代价小于排序归并的网络传输时延代价且连接属性列上有可用索引时，使用基于 In 表达式的半连接操作。

原则 J-3：当半连接的网络传输代价小于排序归并的网络传输时延代价且连接属性列为主键或连接属性列上有自增特性时，使用基于 Between 表达式的半连接操作。

原则 J-4：当半连接的网络传输代价小于排序归并的网络传输时延代价且连接属性列无自增特性无可用索引时，使用基于布隆过滤的半连接操作。

以上总结了在 SDBMS 系统中不同连接算法的执行代价，并提出了在应用改造过程中具体选用哪一种连接算法的四条原则，即介绍了不同连接算法的适用场景。在下一个小节中，通过一组相关实验验证了以上原则的正确性。

4.2.3 SDBMS 连接性能实验

为了验证在不同场景下连接操作的性能，本文设计了一组实验来验证 4.2.2 节中提出的相关原则。本实验的软硬件环境与 3.2.3 节中的实验配置相同。实验依旧使用 Sysbench 来生成数据集，设置了两张关系表分别为 SBTEST1 和 SBTEST2，并在 SBTEST2 的属性列 K 上建立了二次索引。使用 SQL4-3 进行连接操作的性能测试：

SQL 4-3 不同连接算法查询

```
SELECT /*+JOIN(JOIN_TYPE)*/ A.ID,B.ID
FROM SBTEST1 A JOIN SBTEST2 B ON A.K=B.K
WHERE A.ID<=COND;
```

通过输入不同类型的 JOIN_TYPE，使系统在进行连接操作时可以选择不同的连接算法。由于在 CEDAR 系统中目前只实现了排序归并连接算法，基于 In 表达式的半连接算法，基于 Between 表达式的半连接算法和基于布隆过滤的半连接算法，所以本文实验仅对这四种不同的连接算法进行性能比较。通过输入不同的 COND 条件，使左表可以有不同大小的待连接操作数据规模。由于 K 的取值范围是[1, table_size]的随机值，table_size 是关系表的大小，所以在这样的关系表中，K 的值具有很强的选择性，即 K 上不重复值与记录总数的比值接近于 1。实验以系统返回第一条连接结果作为连接操作完成的时间，每组重复 3 次，取执行时间的平均值。

实验一：右关系表不同规模场景下连接算法性能测试。实验设置了左、右表规模相同，分别为 1 万、10 万、100 万、1000 万的场景下，左关系表通过 COND 条件值为 4000，限制了左关系表的进过过滤条件过滤后表的规模大小。分别测试了不同场景下各连接算法的效率，如图 4.3 所示，每组实验的测试结果为执行 3 次后的平均处理时间。

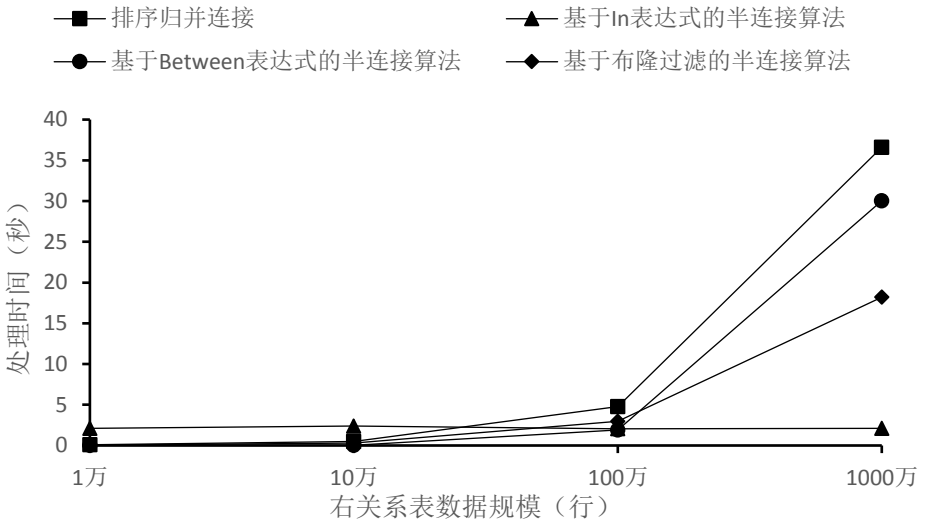

图 4.3　右表不同规模场景下连接算法性能测试

由实验结果可以看出，在小规模数据场景下，排序归并连接算法能够发挥出良好的性能，甚至优于半连接算法。由于索引可以进行快速查找，基于 In 表达式的半连接算法的执行效率几乎不受右关系表的数据规模的影响，即使右关系表的数据规模不断增大，查询的处理时间也处于一个平稳的状态。基于 Between 表达式的半连接算法，由于右表规模的不断增大，在左关系表的过滤区间内，最大值与最小值的过滤性会越来越差，以至于在连接列上如果是随机的数据集，基于 Between 表达式的半连接算法执行效率会近似于排序归并连接算法。基于布隆过滤的半连接算法由于需要对右关系表进行一次扫描，所以查询处理时间随着右关系表数据规模的增长而增长。

实验二：左表过滤数据规模不同场景下连接算法性能测试。实验设置了左右表的大小均为 1000 万的场景下，通过设置不同 COND，使左表分别为一百，一千，一万，十万的场景下，不同连接算法的处理时间。实验结果是经过三次测量后的均值，如图 4.4 所示。

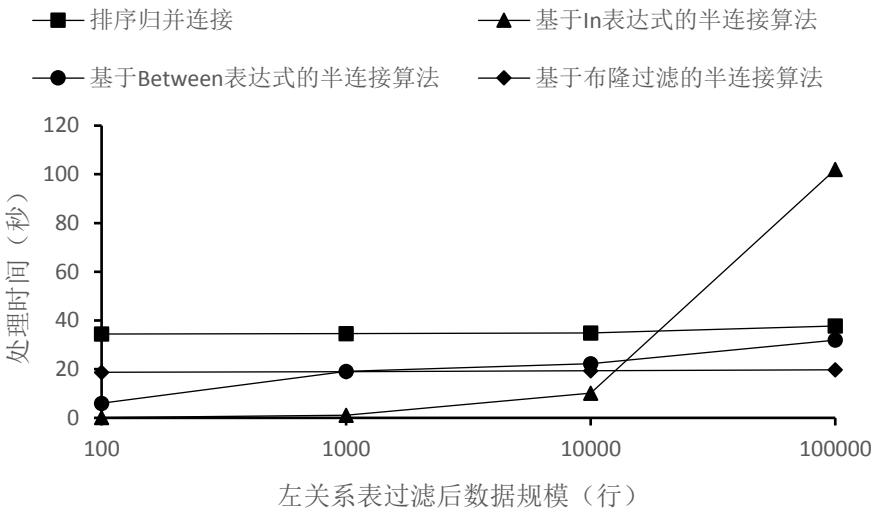

图 4.4　左表过滤数据规模不同场景下连接算法性能测试

由实验结果可以看出，由于右关系规模固定，随着左关系表过滤后数据的增加，归并排序连接对右关系表需要处理的数据量不变，对左关系表需要处理的数据量略微增多，所以总的查询处理时间略微增加。基于布隆过滤的半连接算法由于相似的特性，所以查询处理时间的变化趋势和排序归并连接算法相似。基于 In 表达式的半连接算法，由于左表通过过滤得到的数据不断增多，需要频繁的调用 ***Get***(Key, R)算子从右关系表中读取数据，所以当左关系表过滤后数据量增大候，对应的查询处理时间也极具增长。基于 Between 表达式的半连接操作，由于左关系表过滤都数据量的增多，对右关系表的过滤性能同样降低，所以基于 Between 表达式的半连接算法查询性能也逐渐降低，但由于还是有数据被过滤，因此执行效率上还是略好于排序归并连接算法。

实验三：基于 Between 表达式的半连接算法在不同数据密度下的性能测试。这里的数据密度指的是，需要返回的数据集大小与实际返回数据集大小的比值。假设实际返回结果集大小为 100，但是由于落在 Between 表达式范围内的数据量为 1000，有 900 条不符合连接条件的数据。这时就称数据密度为 100/1000，即 0.1。实验设置了不同左表数据量的场景下，不同数据密度的连接算法性能。实验过程中右表数据量维持在 1000 万条不变，左表数据分别设置为 100、1000、10000、100000 条，密度分别设置为 0.1、0.01、0.001、0.0001。实验结果如图 4.5 所示：

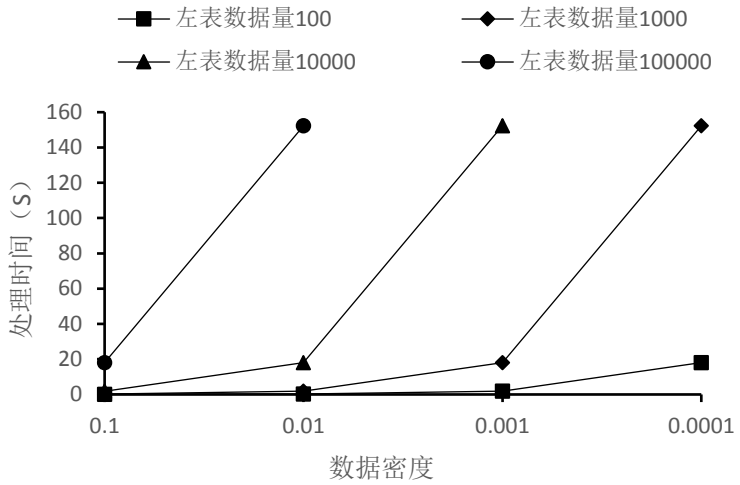

图 4.5　数据密度性能测试

由图可以看出，当数据密度为 0.1 的情况下，随着左表数据的增加，响应时间基本维持在 10 倍的增长，这与左表数据的增长情况相吻合。但是随着数据密度的不断变小，也就是不满足连接条件的数据量的不断增多，导致了响应时间的增加。因此得出的结论为基于 Between 表达式的连接查询效率主要取决于数据密度的大小。数据密度越小，不满足连接条件的数据越多，网络传输时间越长继而响应时间越长。反之数据密度越大，满足连接条件的数据与最终结果集的大小越接近，网络传输时间越短继而响应时间越快。

4.3 本章小结

本章主要介绍了 SDBMS 中与查询语句有关的优化内容。首先说明了和表达式计算有关的优化方案。之后针对查询中连接操作这一重点问题，介绍了传统连接算法在 SDBMS 架构下的简单实现，又根据 SDBMS 的架构特点，介绍了三种不同方式的半连接操作，并分析了它们的执行代价，给出了各自的使用场景。最后通过实验验证了结论的正确性。

第5章 数据迁移与改造案例

5.1 数据迁移方式

在企业应用进行业务迁移的场景中，需要将原有系统中的数据迁移至新的数据库中，通常会面临全量同步以及增量同步两种不同的需求。其中全量同步指的是数据迁移或数据复制，即把原业务系统中使用的数据库（称为源数据库）中的表或视图数据完全地从库中抽取出来，完整地同步到新的数据库中。增量同步指的是从抽取源数据库中上一次抽取以来源数据库中新增或修改的数据，并同步到待同步的数据库系统中。通常来说，根据更新方式的需求不同，会有不同的迁移方案。下面简要的介绍两种同步需求的不同实现方案。

全量同步时，常用的方法有数据文件导入/导出和 ETL（Extract-Transform-Load）数据导入/导出两种方式。数据文件导入/导出指的是通过发送指令让源数据库将指定的表数据完全导出，以文件的形式写入磁盘，在通过对文件的解析，写入待同步的数据库中，如 MySQL 数据库提供的 MySQLdump 与 Source 工具。ETL 数据导入/导出指的是通过 JDBC 接口对源数据库中需要同步的进行数据查询，将查询得到的数据通过一定规则的转换，再通过 JDBC 接口写入待同步的数据库中。相比较于数据文件导入/导出方式，ETL 的数据导入/导出显得更加灵活，且实现较为简单；但是一次性拉取全部数据，如果查询时间过长，会对系统的性能产生影响。

增量同步时，可以采用设置触发器，解析日志，物化视图[48]的方式。设置触发器指的是在源数据库系统中构建触发器，通常需要构建插入、更新、删除三种不同类型的触发器，每当表中数据发生变化时，通过触发器将变化记录到一个临时表中。抽取线程对临时表进行查询，将查询到的结果同步到待同步的数据库中。这样的方式实现简单，抽取数据性能高，不需要对原有的业务逻辑进行任何

修改，但是建立的触发器可能会对系统性能产生影响。解析日志指的是解析数据库系统产生的系统日志，如 MySQL 数据库中的 bin-log，Oracle 数据库中的 LogMiner 工具[49]等，通过解析日志格式将产生的插入、更新、删除操作同步到待同步数据库中。这样的实现方式优点是几乎不影响原系统的操作，避免了对业务系统性能的干扰，但需要开发人员熟悉不同数据库系统的日志格式，或源数据库系统本身提供了日志解析工具。物化视图指的是当源数据表发生变化时，这些变化被记录在视图中。同步工具通过对视图中记录的解析，将数据同步至待同步的数据库中。不同于触发器的是，物化视图会将构建视图前源数据表中的数据也存储在视图中，即物化视图的方式既可以用于增量同步也可以用于全量同步，而触发器的方式仅能用于增量同步。但与触发器相同的是，构建物化视图也会对原有的系统性能产生影响，在我们的测试中，构建物化视图后业务系统的性能大约降为原系统的 70%。

以上简要介绍了在业务迁移中，对数据库原始数据迁移至 SDBMS 的一些方式，但是在真实的迁移场景中，需要根据源数据库产品支持的功能，以及业务上对牺牲系统性能的接受程度综合的来考虑、选择合适的数据迁移方案。

5.2 应用改造案例

本小节展示了一个金融行业的真实应用和一个物流行业的真实应用从传统数据库管理系统迁移至类 SDBMS 的 CEDAR 数据库系统的过程。描述了应用迁移的实验背景、软硬件配置、调优策略以及实验结果四个方面。实验结果显示，通过一系列的调优策略，业务的主要操作在 CEDAR 数据库系统中能够得到与传统数据库管理系统类似的性能。

5.2.1 供应链金融应用调优

供应链金融作为银行对企业提供的一项金融服务，旨在于管理中小企业资金流向与物流，通过多方面获取信息的方式，将风险控制降至最低。这个业务的特点是对数据进行简单的读写操作的同时，业务还伴随有一定的复杂查询，如连接操作等。以某国有银行的供应链金融业务为例，这类应用中数据表的规模在百万

级左右，但是对查询处理的响应时间要求较高。

出于金融企业对应用信息保密的需求，在此，我们不介绍查询的详细逻辑以及测试环境的详细信息，仅给出相应的改造策略以及改造效果。原有的供应链金融业务系统运行在 DB2 数据库中，一共有 1018 个 XML（Extensible Markup Language）文件，它们通过用户输入的条件不同，映射成不同的查询 SQL。由于 CEDAR 系统的 SQL 语法与 MySQL 数据库管理系统一致，这就存在 CEDAR 系统与 DB2 系统的 SQL 语法不一致的问题。为此，在应用改造开始前，通过开发的 SQL 语法转换工具，解决了 SQL 语法不一致问题。在供应链金融业务从 DB2 数据库管理系统中迁移至类 SDBMS 的开源数据库 CEDAR 改进版过程中，经过测试，一共有 87 个 XML 文件中的 SQL 查询语句执行效率无法满足业务系统中响应时间小于 900 毫秒的要求。通过调研发现造成这些查询超时的核心问题数据根据条件进行范围查询的性能以及多表连接操作的性能不能达到业务要求。为了解决这一问题，我们做了两方面的业务改造：一方面，在条件查询的属性列以及连接条件的属性列上构建二次索引；另一方面，对于多表连接查询，并且返回结果集较小的场景下，使用不同的半连接算法。

表 5.1 供应链金融业务改造方法统计

改造方法	数量统计
仅索引改造	14
仅半连接改造	25
混合策略	48

经过统计，两种不同策略的改造用例占需要进行应用改造的总体用例百分比如表 5.1 所示。我们在 62 个 XML 文件涉及的数据表中构建了索引。由于数据表规模较小，对查询的响应时延要求严格，所以在多表连接的改造中没有选择基于布隆过滤的半连接算法，而是选择了基于 In 表达式的半连接算法。根据统计，一共将 73 个 XML 文件中的连接操作由排序归并算法改为了基于 In 表达式的半连接算法。

为了更加全面的展示通过应用改造后，供应链金融业务的具体查询性能情况，

这里定义 QP（p%）符号，其中 QP 是 Query Processing 的首字母缩写，它的具体含义是系统中运行最快的 p%查询请求最长等待时延。如 QP（10%）=20ms，代表一个系统中有多个查询请求，其中运行最快的 10%查询请求响应时延不超过 20 毫秒。通过改造后的实验统计，对于供应链金融业务，得到如表 5.2 所示的性能结果。

表 5.2　供应链金融应用改造性能对比

DB2 时延（ms）		改造前时延（ms）	改造后时延（ms）
QP（10%）	31	57810	26
QP（30%）	50	55120	10
QP（50%）	75	23300	323
QP（70%）	149	6890	640
QP（90%）	284	49700	2360
QP（100%）	316	66340	3640

这里的原系统时延指的是供应链金融业务运行在传统数据库管理系统中的时延，即 DB2 系统中的时延；改造前时延指的是业务迁移至 CEDAR 数据库中未经过应用改造的处理时延；改造后时延指的是业务迁移至 CEDAR 数据后经过应用改造后查询处理时延。QP（p%）是对原系统而言，即以 DB2 中运行最快的查询请求为基准，得到与之对应的查询请求时延。其中部分查询请求在 CEDAR 中要优于 DB2 系统是由于这些连接操作所得结果为空集，得益于基于 In 表达式的半连接算法对这样的场景可以立即响应，所以查询性能会优于 DB2。其中 QP(70%)处改造前时延要明显小于其他查询的时延，这是由于每一个查询都是相互独立的，它们之间的时延并不存在相关性，由于 DB2 数据库系统与 CEDAR 数据库系统架构的不同，所以两个系统间不同类型的操作也不会是成比例的关系。通过对比，可以看出业务系统中大部分查询，如前 70%的查询请求能达到近似于 DB2 系统的性能。其它运行较慢的查询请求相比于应用改造前，查询性能大幅提升，但由于 CEDAR 数据库网络交互的特点，在查询语句层面难以进行进一步优化，需要通过继续修改系统执行流程，开发效率更高的连接查询算法来实现性能的进一步提升。

5.2.2 物流产品报表分析应用调优

物流产品报表分析应用，这类应用一方面对数据库的响应时间要求不像供应链金融业务那样严格，响应时间通常维持在秒级；另一方面，这类应用不严格要求数据的实时性，通常说来是对一天前、一周前或是一个月前的数据进行统计分析。这一类应用在查询上性能较低的主要原因在于，业务的查询 SQL 主要是由于操作的数据量大、数据结构多样，操作多为复杂查询，如涉及到多表连接，聚合运算等操作。

本小节中介绍的产品报表分析应用是某物流公司的真实线上业务，它的主要功能是展示物流公司各省市地区配送站点中不同物流产品的收营状况以及各物流线路上货物流转情况等。根据分析，该报表业务需要进行大量类似于根据物流单号进行连接操作，之后再根据具体物流产品类型进行聚合计算的操作。经过长期的业务发展，该公司的数据表规模较大，许多记录单号详细信息的关系表数据量都在千万条记录，有的表数据量甚至达到了上亿条记录。如前分析，由于 SDBMS 中节点分布式的架构，通过网络传输的大规模表连接操作会使得查询响应时间十分漫长。但是此类应用还具有另外两个特点：

1）报表数据的访问人员仅需要访问前一天或是前一段时间的统计数据，对实时产生的详细数据并不关心。

2）业务人员仅在白天工作时间需要访问数据库管理系统，凌晨至次日工作时间之前，系统属于无访问，无数据写入状态。

根据上述两个业务特点，对业务进行了如下改造：

首先，调研并总结出业务系统中的涉及聚合操作和连接操作的 SQL；其次，根据这些 SQL 建立中间加工表用于存放 SQL 的执行结果，目的在于将耗时的聚合操作与连接数操作在系统空闲的时间内完成，也就是在凌晨；然后，重写 SQL，使其具有与原 SQL 相同的语义，最终仅仅是对加工表的查询操作；最后，使用与供应链金融应用相似的调优策略对改写后的 SQL 进行在不改变语义的情况下的二次改造，进一步减少响应时间。

本文准备了 4 台位于同一机房的商用服务器来部署集群，部署一个 RootServer 节点与一个 UpdateServer 节点在同一商用服务器上，在其余三台服务器中每台服务器上都部署一个 ChunkServer 节点和一个 MergeServer 节点。所有的服务器配置均相同，服务器间采用千兆网络连接，节点间的网络传输速率在 110MB/s 至 120MB/s 之间。每台服务器的操作系统是 64 位操作系统，版本是 CentOS release 6.5(Final)，硬件环境上每台服务器配置了 2 颗型号为 Intel(R)Xeon E5-2620 的 CPU，核心的频率是 2.1GHZ，同时还配备了 128G 的内存以及 3TB 的硬盘。

由于原报表业务是运行在 Oracle 数据库系统中，所以在上述环境中部署了单机环境的 Oracle 数据库系统，以及在 4 台相同配置的物理机上搭建了 CEDAR 数据库系统，共部署了 1 个 RootServer 节点，1 个 UpdateServer 节点，3 个 MergeServer 结点，3 个 ChunkServer 结点。在此，为了使 Oracle 数据库系统中的真实业务数据能够迁移至 CEDAR 数据库系统中，使用阿里巴巴开源数据迁移工具愚公[50]进行数据迁移工作。数据迁移的原理如 5.1 节所述，是通过 Oracle 数据库系统的物化视图功能，通过对物化视图的抽取、转换，再通过与 CEDAR 数据库建立 JDBC 连接，进行数据的导入。完成数据导入工作后，对改造后的业务性能进行了测试。

据统计，报表分析业务系统中共产生 82 张数据表，其中 64 张为基础数据，18 张为分析具体业务后，设计的中间加工表。原系统中共涉及业务查询的 SQL 操作共有 74 条，其中 46 条为具有复杂的聚合、连接操作。剩下 28 条为简单的选择查询操作。调优后，系统中的查询 SQL 总数不变，但是原先复杂的聚合、连接操作被改为简单的从加工表中读取数据。业务系统每天夜间执行复杂的聚合、连接操作，将生成的结果集写入到设计的加工表后，CEDAR 数据库再执行当天的定期合并任务。通过模拟业务分析人员分别对 Oracle，业务改造前的 CEDAR 以及业务改造后的 CEDAR 三个不同的系统以查询时间跨度为一天，进行了查询性能测试实验。实验结果如表 5.3 所示：

表 5.3 物流报表分析应用改造性能对比

原系统时延（ms）		改造前时延（ms）	改造后时延（ms）
QP（10%）	40	682	84
QP（30%）	62	939	157
QP（50%）	313	8950	2130
QP（70%）	1260	10203	2480
QP（90%）	1480	2971000	134600
QP（100%）	2910	3602300	152300

从测试结果可以看出，执行流程较为简单的普通查询请求，经过应用改造后在 CEDAR 数据库中的运行效果能够达与 Oracle 数据库近似。运行效率慢的是执行流程较为复杂，通过查询生成加工表数据的查询请求，这类请求在 CEDAR 系统中通常要运行近 1 个小时，通过改造后可降至 3 分钟左右。虽然上述查询请求在 CEDAR 系统中的运行效率与 Oracle 系统相比性能有较大差距，但是这类查询请求被设置在系统无人使用的夜间执行，不影响业务人员正常使用时的体验。

5.3 本章小结

本章首先介绍了企业在进行业务迁移时首先需要解决的问题，数据迁移问题。描述了传统数据库管理系统将数据迁移至 SDBMS 中可用的方式。之后介绍了一个金融行业的真实应用改造与一个物流行业的真实应用改造，并进行了相关的实验测试。实验证明，使用如前所述的策略对应用进行改造后，这些业务的查询效率可以达到与传统数据库管理系统近似的效果。这样企业在获得 SDBMS 类系统良好的扩展性以及更廉价的采购成本等好处的同时，能够获得近似于传统数据库管理系统的查询效率。

第6章 总结与展望

6.1 本文总结

信息技术的蓬勃发展使得商业活动越来越多元化。许多传统行业无论是主动地还是被动地，都必须适应新时代的业务特点。金融行业，例如银行业务，不再仅仅局限于柜台的业务办理，网上银行的普及使得银行的后台电子系统可能随时面临着大规模的用户涌入；由于电子商务的发展，商品线上促销等活动的流行，物流行业的后台电子系统也随时可能面临着大规模的业务订单涌入。同时随着时间的累积、业务的发展，这些企业需要存储的数据越来越多，越来越详细。传统的数据库管理系统由于硬件设备的限制，很难适应如今快速变化的商业环境。可扩展数据库管理系统由于它的灵活性等优点，受到越来越多的关注。但是没有一个数据库管理系统是完美的，企业在计划将原先的业务系统迁移到新的数据库管理系统时，总是会遇到各种各样的问题。

本文就企业将线上业务系统迁移到可扩展数据库管理系统过程中，缺乏完整的数据迁移方案以及系统查询性能过慢是最迫切需要两大问题。本文从这两大问题入手，一方面介绍了将传统数据库管理系统中存储的企业数据迁移至可扩展数据库管理系统中的常用方法。另一方面，从数据库的存储层面和数据库的查询层面提出了优化业务性能的方式。由于可扩展数据库管理系统发展时间有限，其架构相比于传统的数据库管理系统又发生了改变的原因，本文所进行的研究分析意义在于，从真实的应用场景出发，通过分析真实应用在可扩展数据库系统中运行过程中，无法满足业务性能要求的业务逻辑特点。通过业务改造的方式能够以最快的速度帮助企业在真实环境中提升业务查询性能，达到业务性能要求。通过总结业务改造中使用方法，构建代价模型，分析改造有效性的方式，可以为后续可扩展数据库管理系统的研究发展提供了参考意见。本文通过分析针对数据库系统

的存储结构，阐述了三个方面的优化策略，分别是表结构优化策略，索引结构优化策略以及数据分片优化策略；针对查询语句，提出了业务迁移至可扩展数据库管理系统后的两个优化策略，分别是表达式优化策略和连接操作优化策略。本文详细介绍了它们在实现上与传统数据库管理系统的异同，通过代价分析，解释了它们在可扩展数据库管理系统的架构下，导致查询性能劣于传统数据库管理系统的原因，给出了在进行应用改造时可以弥补这些劣势的原则。总结的原则如表 6.1 所示：

表 6.1 SDBMS 应用改造原则

改造策略	原则详情	对应编号
库表结构	对于不要求数据实时性的多表连接操作，创建加工表存储查询快照。	T-1
	当连接操作在多个属性列上等价时，选择整形等简单数据类型作为连接条件。	T-2
二次索引	当查询的过滤条件涉及多个属性列时，在选择区分率最高的列上构建二次索引。	I-1
	构建索引时，根据查询的具体投影内容，在索引表中冗余查询需要返回的属性列	I-2
定期合并	当 SDBMS 进行数据导入操作后，应该先发起定时合并任务，待合并完成，数据以数据分片形式存储在 SNode 中之后，再接受业务的查询操作。	M-1
表达式	在日期类型的数据列上对输入的字符串日期进行过滤查询时，调用转换表达式进行提前计算。	E-1
连接操作	当排序归并的网络传输时延代价小于半连接的网络传输代价时，使用排序归并连接算法进行连接操作。	J-1
	当半连接的网络传输代价小于排序归并的网络传输时延代价且连接属性列上有可用索引时，使用基于 In 表达式的半连接操作。	J-2
	当半连接的网络传输代价小于排序归并的网络传输时延代价且连接属性列为主键或连接属性列上有自增特性时，使用基于 Between 表达式的半连接操作。	J-3
	当半连接的网络传输代价小于排序归并的网络传输时延代价且连接属性列无自增特性无可用索引时，使用基于布隆过滤的半连接操作。	J-4

对于不同的改造策略，设计并完成了相应的实验，证明了策略建议的正确性以及改造方案的有效性。最后通过分别对金融行业和物流行业的实际线上应用进行了数据迁移与应用改造，并给出了改造前后的查询时延对比。实验结果表明，通过一定策略的业务改造，可以使大部分的查询得到近似于传统数据库管理系统中的查询效率，可以满足企业对业务的需要。

6.2 未来展望

基于本文所展示的应用改造原则，虽然快速有效的提升了业务性能，使数据库系统的响应时间能够达到业务要求，帮助真实应用迁移后的尽快上线。但是从长远的角度来看，为了使可扩展数据库管理系统能够在企业的业务系统中获得更广泛的使用，还可以从以下三个方面出发，考虑可扩展数据管理系统的后期研究与开发工作。

1) 虽然基于不同方式的半连接算法解决了带过滤条件的数据表与大规模数据表的连接操作性能问题。但是不可回避的是，企业的真实应用中任然会存在大规模数据表与大规模数据表的连接操作。而相比于传统的数据库管理系统，可扩展数据库管理系统在处理这样场景下的连接操作的性能还有很大差距。解决可扩展数据库管理系统中大规模数据表与大规模数据表的连接操作性能问题还需要未来的进一步研究与实践。

2) 虽然通过业务改造的方式，可以快速的帮助某个具体的业务完成性能优化，帮助业务尽快顺利上线。但这样的改造是高度定制化的优化方式，期间充满了大量繁琐重复的工作，当一个新的业务系统性能受到影响的时候，这些工作并不能快速简单的迁移至新的业务系统中，还是需要大量的重复工作，分析业务逻辑才可以查找出业务系统的查询瓶颈，进行改造。因此，基于本文提出的一些原则，开发查询优化器是有必要的。通过查询重写，代价估计等手段，尽可能的让一些简单的查询优化方案可以自动化地让系统判别，减少手工改造时需要进行大量重复繁琐的工作。

3) 性能仅是衡量一个数据库管理系统价值的一个方面，同时，评判一个系统的稳定性与易用性也是不可或缺的条件。而由于发展时间有限等原因，可扩展数据库管理系统还缺乏简单易用的可视化客户端，以及自动化运维工具。由于没有可视化客户端，增加了不熟悉 SDBMS 系统人员在管理使用系统时的学习成本。由于缺乏自动化的运维工具，缺乏对系统运行时节点状态的可视化监控，而 SDBMS 又是一个分布式的系统，部署节点众多。在实际的生产环境中使用 SDBMS 会极大的增加系统运维人员的工作量，也给系统出现异常时，错误排查增加了难度。所以开发丰富系统的外围工具，让系统运行时的状态清晰可见，减少运维人员的相关工作量，也是可扩展数据库管理系统在企业应用中需要重点研究和解决的方向。

在可扩展数据库管理系统未来的研究与开发过程中，可以继续以企业真实业务出发，不断总结和提炼新的负载模型，为可扩展数据库管理系统后续的研究与发展提供指导意义。

参考文献

[1] 新浪财经. 2015 阿里双 11 交易额 912.17 亿 同比增长 59.7%. http://finance.sina.com.cn/china/20151112/000623742971.shtml. [Online;accessed 16-September-2016].

[2] 新华网. 国家邮政局:“双 11”当天产生物流订单 4.6 亿件 同比增长 65%. http://news.xinhuanet.com/fortune/2015-11/12/c_128418876.html. [Online;accessed 16-September-2016].

[3] Rothnie J B, Bernstein P A, Fox S, et al. Introduction to a system for distributed databases (SDD-1) [J]. In: ACM Transactions on Database Systems, 1980, 5(1): 1-17.

[4] Lampson B W, Sturgis H E. Crash recovery in a distributed data storage system[M]. Palo Alto, California: Xerox Palo Alto Research Center, 1979.

[5] Furman J, Karlsson J S, Leon J M, et al. Megastore: A scalable data system for user facing applications[C]//ACM SIGMOD/PODS Conference. 2008.

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

[7] Shute J, Oancea M, Ellner S, et al. F1-The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business[J]. 2012.

[8] Han J, Haihong E, Le G, et al. Survey on NoSQL database[C]//Pervasive computing and applications (ICPCA), 2011 6th international conference on. IEEE, 2011: 363-366.

[9] Decandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly av

ailable key-value store [C]. In: ACM Symposium on Operating Systems Principles, 2007, 41(6): 205-220.

[10] Salvatore Sanfilippo. Redis . http://redis.io/. [Online;accessed 16-September-2016].

[11] Lakshman A, Malik P. Cassandra: a decentralized structured storage system [C]. In: ACM Special Interest Group on Operating Systems, 2010, 44(2): 35-40.

[12] Apache. Apache HBase. http://hbase.apache.org/. [Online;accessed 16-September-2016].

[13] Chodorow K. MongoDB: the definitive guide[M]. " O'Reilly Media, Inc.", 2013.

[14] Anderson J C, Lehnardt J, Slater N. CouchDB: The Definitive Guide [M]. O'Reilly Media, 1 edition, 2010.

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

[16] 华东师范大学数据科学与工程学院. Cedar. https://github.com/daseECNU/Cedar. [Online;accessed 16-September-2016].

[17] Bachman C W. The programmer as navigator[J]. Communications of the ACM, 1973, 16(11): 653-658.

[18] Kamfonas M. The Relational Journal [J]. The Relational Journal, 1992, 27 (10).

[19] Codd E F. A relational model of data for large shared data banks[J]. Communications of the ACM, 1970, 13(6): 377-387.

[20] Palermo F P. A data base search problem[M]//Information Systems. Springer US, 1974: 67-101.

[21] Smith J M, Chang P Y T. Optimizing the performance of a relational alg

ebra database interface[J]. Communications of the ACM, 1975, 18(10): 568-579.

[22] Hall P A V. Optimization of single expressions in a relational data base system[J]. IBM Journal of Research and Development, 1976, 20(3): 244-257.

[23] Microsoft. Microsoft SQL Azure. https://azure.microsoft.com/en-us/services/sql-database/. [Online;accessed 16-September-2016].

[24] Graefe G. Volcano-an extensible and parallel query evaluation system[J]. IEEE Transactions on Knowledge and Data Engineering, 1994, 6(1): 120-135.

[25] Graefe G. The cascades framework for query optimization[J]. IEEE Data Eng. Bull., 1995, 18(3): 19-29.

[26] Oracle. Oracle Database. https://www.oracle.com/database/index.html [Online;accessed 16-September-2016].

[27] The PostgreSQL Global Development Group. PostgreSQL https://www.postgresql.org/. [Online;accessed 16-September-2016].

[28] 彭智勇, 彭煜玮. PostgreSQL 数据库内核分析[M].北京：机械工业出版社, 2012 : 268-276.

[29] IBM. DB2. http://www-03.ibm.com/software/products/zh/db2enterprise-server-edition [Online;accessed 16-September-2016].

[30] Zilio D C, Rao J, Lightstone S, et al. DB2 design advisor: integrated automatic physical database design[C]//Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. VLDB Endowment, 2004: 1087-1097.

[31] Kuhn D, Alapati S R, Padfield B. SQL Tuning Advisor[M]//Expert Indexing in Oracle Database 11g. Apress, 2012: 205-231.

[32] Robert B, Irene G. Database performance Optimization andcapacity planni

ng[J]. Business Communications Review, 1997, 4(16): 28-32.

[33] Shasha D, Bonnet P. Database tuning: principles, experiments, and trouble shooting techniques[M]. Morgan Kaufmann, 2002.

[34] Belknap P, Dageville B, Dias K, et al. Self-tuning for SQL performance i n Oracle database 11g[C]//2009 IEEE 25th International Conference on Dat a Engineering. IEEE, 2009: 1694-1700.

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

[36] Oracle. Tuning the Database Buffer Cache. http://docs.oracle.com/database/ 121/TGDBA/tune_buffer_cache.htm#TGDBA294. [Online;accessed 16-Septem ber-2016].

[37] Oracle. The InnoDB Buffer Pool https://dev.mysql.com/doc/refman/5.5/en/in nodb-buffer-pool.html. [Online;accessed 16-September-2016].

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

[39] Google. LevelDB. https://github.com/google/leveldb. [Online;accessed 16-Se ptember-2016].

[40] 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.

[41] 翁海星, 宫学庆, 朱燕超,等. 集群环境下分布式索引的实现[J]. 计算机应用, 2016, 36(1):1-7.

[42] Alexey Kopytov. Sysbench: a system performance benchmark. http://sysben ch.sourceforge.net. [Online;accessed 16-September-2016].

[43] Garcia-Molina H, Ullman J D, Widom J. Database system implementation [M]. Upper Saddle River, NJ:: Prentice Hall, 2000 : 719-735.

[44] Apers P M G, Hevner A R, Yao S B. Optimization algorithms for distributed queries[J]. IEEE Transactions on Software Engineering, 1983 (1): 57-68.

[45] 钱招明, 王 雷, 余晟隽, 等. 分布式系统中 Semi-Join 算法的实现[J]. 华东师范大学学报 (自然科学版), 2016(5): 75-80.

[46] 茅潇潇, 段惠超, 高明. OceanBase 中基于布隆过滤器的连接算法[J]. 华东师范大学学报 (自然科学版), 2016(5): 67-74.

[47] Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended bloom filter: an aid to network processing[J]. ACM SIGCOMM Computer Communication Review, 2005, 35(4): 181-192.

[48] Bello R G, Dias K, Downing A, et al. Materialized views in Oracle[C]// VLDB. 1998, 98: 24-27.

[49] Wright P M. Oracle database forensics using LogMiner[C]//June 2004 Conference, SANS Institute. 2005.

[50] Ali YuGong. https://github.com/alibaba/yugong. [Online;accessed 16-September-2016]

附录

在本节中展示了本文调优事例涉及到的所有表结构。

Table 1. 全国站点信息表

列名	列含义	列数据类型	是否主键
SITE_ID	站点编号	INT	是
SITE_AREA	站点所属地区	VARCHAR(20)	否
SITE_NAME	站点名称	VARCHAR(20)	否
SITE_LEVEL	站点等级	INT	否
SITE_MANAGER	站点负责人	VARCHAR(20)	否

Table 1 是物流公司维护了公司在全国范围内所有配送货物站点的信息，出于对企业信息保密等诉求，隐藏了表的实际表名，仅用 Table 1 作为表名示意。表中展示的列信息并不是实际表中的全部列信息，仅罗列了可以用以说明本文查询含义的相关列信息，本文后续的其他表结构也是相同的处理方式。其中站点编号是公司内部对不同站点的统一编号，所属区域表示该配送站点所属的省市信息，站点等级是公司内部对该配送站点硬件环境等的评级。

Table 2. 站点到件扫描表

列名	列含义	列数据类型	是否主键
ARRIVE_TIME	到站时间	DATE	是
EWB_NO	运单编号	VARCHAR(14)	是
SITE_ID	站点编号	INT	否
WEIGHT	货物重量	DECIMAL(8,2)	否
EMPLOYEE_ID	操作人员	INT	否
LAST_SITE_ID	上一站点编号	INT	否

Table 2 维护了全国各个站点每天到达货物的收入信息，其中运单编号是由公司按照流水自动生成的唯一编号，每一个编号对应了唯一的一件货物。货物重

量中记录了该货物寄送时测量的重量。操作人员存储的是处理该货物的员工在公司人员名册中存储的唯一工号。上一站点编号存储的是该货物来自于哪一个站点。

Table 3. 运单流水表

列名	列含义	列数据类型	是否主键
EWB_NO	运单编号	VARCHAR(14)	是
SEND_INFO	寄件人信息	VARCHAR(60)	否
RECEIVE_INFO	收件人信息	VARCHAR(60)	否
EWB_DATE	寄件日期	DATE	否
EWB_STATE	运单状态	INT	否
EWB_CHARGE	运单费用	DECIMAL(6,2)	否

Table 3 存储了公司所有运单的流水信息，其中寄件人信息中存储的是寄件人的地址信息，当然还有其他列用于存储寄件人的姓名，联系方式等信息，但这不是本文分析的重点，作省略处理，仅使用寄件人信息一列用于说明用例，收件人信息也是同理。运单状态记录了货物所处的状态编号，根据编号不同有已收件，运输途中，等待配送，已签收等状态。运单费用记录的是该货物的运送费用。

Table 4. 运单详情表

列名	列含义	列数据类型	是否主键
EWB_NO	运单编号	VARCHAR(14)	是
EWB_COST	运输成本	DECIMAL(6,2)	否
PRODUCT_ID	产品类型	INT	否

Table 4 是对 Table 3 的补充，Table 4 所示的运单编号与 Table 3 的运单编号存在着一一对应的关系。运输成本记录了公司运输该货物需要付出的成本。产品类型记录的是公司提供的不同物流产品类型编号，如货物重量、体积较小的小包配送，对配送时效有严格要求的定时配送等产品。

SBTEST. Sysbench 性能测试表

列名	数据类型	是否主键
ID	INTEGER	是
K	INTEGER	否
C	CHAR(120)	否
PAD	CHAR(60)	否

SBTEST 表是 Sysbench 性能测试工具产生的表结构。由于数据为自动生成，所以这里不解释其具体含义，只介绍它的数据特征。其中，ID 为关系表的主键，具有递增性质，K 是一个随机生成的整形，而 C 以及 PAD 是随机生成的字符串类型。

致谢

转眼两年半的研究生生活即将过去，想起两年前七月那个阳光炽热的下午，是李宇明同学接应的我，带着我踏进了数学馆的大门。那一刻的我大概还没有想过我的研究生生活就这样开始了。回首两年来，从数学馆东边的实验室到西边，再到地理馆,再到交通银行,日子就像幻灯片在眼前一幕幕的划过。两年半以来，要感谢的人实在太多。

首先我要感谢我的导师，钱卫宁老师，刚开始接触到数据库方向时，我对自己并没有什么信心，是钱老师的鼓励与督促让我不断向前。感谢钱老师对我毕业论文的悉心指导，撰写毕业论文期间，每一次与钱老师沟通修改的过程中，钱老师严谨认真的学术态度，清晰缜密的逻辑思维都深深的影响着我。

其次我要感谢宫学庆老师和张蓉老师。张老师在我刚来学校时，对我生活上的照顾使我铭记在心，忘不了张老师在我刚接触到数据库方向时，带着我们做测试的时光。感谢宫老师在交通银行项目期间，对我们学习上的启发以及生活上的关照。美好的时光总是让人难忘，宫老师在生活中平易近人，在学术上睿智严谨以及工作中的认真负责都给我留下了深刻的印象。每一次与宫老师的交谈都让我受益无穷。

感谢 Cedar 系统组的其他老师，周敏奇老师，高明老师，蔡鹏老师，张召老师。每一次的论文研讨会中，老师们睿智的提问与渊博的学识以及对学术的思维都有意无意的影响着我，培养着我对学术，对工作的思考方式，在此衷心的感谢各位老师。

此外还要感谢实验室的各位同学，感谢王雷，忘不了在交通银行时一起讨论设计方案，一起修改 BUG 的夜晚。感谢郭进伟博士、朱涛博士、李宇明博士，与你们一起讨论问题，谈论生活的日子总是那么令人难忘。感谢钱招明，刘柏众，储佳佳，朱燕超，祝君，熊辉同学，与大家一起为交通银行项目奋斗期间，感谢

大家对我工作上的支持与理解以及对生活上的关照与谦让。感谢实验室中其他的小伙伴们，与大家在实验室的日子总是那么愉快。

曲终未必人散，有缘自会重逢。希望大家学业顺利，前程似锦，有缘再见！

余晟隽

二零一六年九月二十八日

发表论文和科研情况

■ 已发表或录用的论文

[1] **余晟隽**, 宫学庆, 祝 君, 钱卫宁. 基于 Map/Reduce 的分布式数据排序算法分析[J]. 华东师范大学学报(自然科学版),6(5):121-130,2016.(CSCD).

[2] 钱招明, 王 雷, **余晟隽**, 宫学庆. 分布式系统中 Semi-Join 算法的实现[J]. 华东师范大学学报(自然科学版),6(5):75-80,2016.(CSCD).

[3] 祝 君, 刘柏众, **余晟隽**, 宫学庆, 周敏奇. 面向 CEDAR 的存储过程设计与实现[J]. 华东师范大学学报(自然科学版),6(5):144-152,2016.(CSCD).

■ 参与的科研课题

[1] 国家自然科学基金重点项目（U1401256），2014-2018，参与人

[2] 国家高技术研究发展计划（863 计划）课题，基于内存计算的数据库管理系统研究与开发（2015AA015307），2015-2017，参与人