2015届研究生硕士学位论文

分类号：_____ 学校代号: 10269

密　级：_____ 学　　号: 51121500012

華東師範大學

East China Normal University

硕士学位论文

MASTER'S DISSERTATION

论文题目：面向海量金融数据并行加载技术研究与实现

院　　系:	软件学院
专　　业:	软件工程
研 究 方 向:	WEB数据管理和WEB挖掘
指 导 教 师:	周傲英　教授
学位申请人:	李永峰

2015 年5 月5 日

Dissertation for master's degree in 2015 School Code: 10269

Student ID: 51121500012

East China Normal University

Title: DESIGN AND IMPLEMENTATION OF PARALLEL LOADING TECHNOLOGY FOR MASSIVE FINANCIAL DATA

Department:	Software Engineering Institute
Major:	Software Engineering
Research direction:	WEB Data Management and WEB Mining
Supervisor:	Prof. ZHOU Ao-ying
Candidate:	LI Yongfeng

May, 2015

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《面向海量金融数据并行加载技术研究与实现》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名:_____ **日期**: 年 月 日

华东师范大学学位论文著作权使用声明

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

本学位论文属于（请勾选）

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

（ ）2.不保密，适用上述授权。

导师签名_____ 本人签名_____

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）。

李永峰 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
王晓玲	教授	华东师范大学	**主席**
金澈清	教授	华东师范大学	
钱卫宁	教授	华东师范大学	
高明	副教授	华东师范大学	
张蓉	副教授	华东师范大学	

摘　要

随着互联网技术的快速发展，金融、通信、教育等行业对信息化的需求不断地增加。在国内，信息化发展已经有几十年，金融行业更是成为信息化程度最高的行业。随着用户的增长和业务的更新，金融行业数据库的数据量也不断地增长，其数据量高达几百TB甚至PB级。通常，金融企业需要大型数据库系统来存储和管理海量金融数据。同时由于业务需求，不同金融系统间需要大量的数据共享，因此不同系统间需要大量的数据迁移和加载。海量金融数据的存储和加载，给金融系统提出了严峻的挑战。

本文主要针对海量金融数据的数据加载问题展开研究，并以一个实际金融系统作为研究对象，结合其底层数据存储架构及数据加载特点，设计和实现适用于该系统的海量数据加载方法。主要贡献如下：

1. 基于交通银行历史库系统，我们分析了历史库系统的海量数据存储和加载实现，其底层数据存储采用分布式数据库OceanBase来解决海量数据存储的问题。通过分析历史库的数据加载特点，我们发现新存储架构下的历史库系统面临海量数据加载问题。为此，我们提出了两种解决思路。

2. 针对OceanBase数据加载的实现，我们设计和实现了两种加载方法：基于SQL INSERT的数据加载和直接更新内存表的数据加载。前者是一种常见的数据导入技术，主要通过并发执行插入SQL来实现数据导入。后者则根据OceanBase特有的存储架构，将数据加载问题转化为B+树的并发插入问题。这种加载方法只适用于OceanBase。相比于前者，该方法可以减少网络传输和事务处理量，从而提高加载效率。实验表明该加载方法较好地解决OceanBase 数据加载问题。

3. 根据历史库系统的数据加载特点，为了提高整体的数据加载效率，我们提出一种多任务并行加载的方法。该方法将所有加载任务切分到多个加载服务器上，充分利用加载服务器和数据库系统的资源，使得加载任务并行运行于不同加载服务器上。

4. 为了获取更好的并行加载效率，我们提出了两种任务调度策略：基于表级任务调度和基于细粒度的两阶段任务调度。这两种调度策略分别基于不同的划分粒度，被应用于多任务并行加载过程中，使尽可能多的加载任务并行执行。实验表明，两阶段调度策略可以更充分地利用加载服务器资源，获取更好的加载效率。

关键词：海量金融数据，数据加载，并行加载，任务调度

Abstract

With the rapid development of Internet, the demand for information technology in finance, transportation, telecommunications and other industries, continue to increase. In China, financial industry has been at the forefront of information technology. Along with the growth of users and business updating, the amount of data in the financial database is also constantly increasing, which can be hundreds of terabytes and even petabytes. Generally, financial companies need a large database system to store and manage massive financial data. Meanwhile, due to business needs, sharing and transferring of huge information is required between different financial systems, which requires the loading of data between different databases. So, massive financial data storing and loading have been a prominent challenge in financial systems.

This thesis focuses on massive financial data loading technology, and with a real financial system as the research object, combining with its underlying data storage architecture and data loading characteristics, design and implement a mass data loading method, which is suitable for the system. Major contributions are listed as follows:

1. Based on history database system in bank of communication, we analysis its underlying data storage architecture, which is use of distributed database OceanBase to solve the problem of massive data storage. According to analysing loading characteristics, we discover that history database system faces massive data loading problem. To this, we propose two solutions.

2. To address the issue of data loading into OceanBase, we design and implement two data loading method: data loading based on SQL INSERT, and direct update Memtable(DUMT). The former is common loading technique, which is implemented in SQL execution mode. But, the latter is based on OceanBase， which is only suitable for OceanBase. Compared to the former, this approach can reduce the volume of network transmission and transactions, and improve loading efficiency. Empirical studies show the loading efficiency of the latter is better than the former.

3. According to characteristics of data loading in history database system, we propose a multi-task parallel-loading method, which splits the loading task up across multiple loading servers. This loading method makes full use of computing resources in database and loading server, to run loading task in parallel on different loading server.

4. To get a better parallel-loading efficiency, two task scheduling strategies are proposed, including scheduling based on table loading tasks, and two-phase scheduling based on fine-grained loading tasks. Both scheduling strategies are based on different task granularity, and make the execution of tasks parallelize as much as possible. Empirical studies show two-phase scheduling get a better loading efficiency, which can make better use of computing resource of loading servers.

Key Words: *Massive Financial Data, Data Loading, Parallel Loading, Task Scheduling.*

目录

插图

表格

第一章　绪 论

1.1　研究背景

上个世纪90年代以来，随着互联网技术的快速发展，金融、通信、教育等行业对信息化的需求不断增加。同时，国内的信息化发展已经有几十年时间，金融行业更是成为信息化程度最高的行业。随着企业的快速发展，用户量和业务处理量也不断增加，导致企业的信息量和数据量也不断地增长。目前，企业级数据库的数据量高达几百TB甚至PB级，尤其是金融行业，通常这些企业需要大型的数据库系统存储和管理海量数据。目前，国内主要的商业银行的客户数量均以亿计，存款账户数量甚至突破十亿，每天的交易量巨大。再加上一些金融类产品譬如基金、外汇、债券等，以及一些新兴业务如电子网银等，银行每天会产生极其巨大的交易数据量。要同时满足存储和处理这些日益增长的海量数据，必须借助数据库技术与硬件，从而来解决业务系统的性能瓶颈。随着金融行业日益加快的发展步伐，业务不断地增加和扩张，无论是国有五大银行还是其他商业银行，其面临的海量数据的存储和管理问题都会日益加重。

目前，绝大多数的金融行业都采用”IBM小型机+DB2/Oracle数据库”来构建其业务系统的底层数据存储系统，利用高端存储阵列和高可靠数据库软件来保证业务的高可靠性和高可用性。随着业务处理的不断增长，无论是数据量还是访问量，即使采用昂贵的IBM小型机甚至大型机，单台数据库管理系统都无法承受。而在原有的架构下，银行业的常见解决方法是根据业务特点对数据库进行拆分，将数据水平切分存储于不同的数据库服务器上，而客户端通过数据库中间层将不同的请求分发到不同的数据库服务器上。随着业务处理和数据量不断增长，需要不断地增加服务器，这种方法需要大量的人工维护成本和硬件开销。因此，很多企业把眼光转向已有十几年研究经验的分布式数据库系统，并利用其良好的可扩展性和容错性等来满足存储海量数据的应用需求。近几年，随着分布式技术的快速发展，各种分布式数据库系统陆续出现，譬如Google Spanner[1]、VoltDB[2]、以及阿里巴巴的OceanBase[3] 等，其中，已经有一些银行系统采用分布式数据库替换掉原有的DB2或Oracle[4]。

本文基于交通银行的一个特定的应用系统——历史库系统，其主要用于存储和管理海量历史交易数据，并支持用户的历史交易信息的查询和分析。历史库系统包含20多个子系统，共涉及300多张数据表，通常需要保存3-5年的用户历史交易数据，数量达到几百TB甚至PB级别。从数据库管理系统的角度看，该系统的需求主要有数据量大、高可用和可扩展等特点。面对如此海量数据的存储和管理，交通银行历史库系统率先采用分布式数据库OceanBase来构建底层数据存储系统，通过分布式存储和分布式查询的技术来解决海量数据存储和查询。

解决了数据存储问题，同时也带来了新的问题和挑战。由于历史库系统的数据来源于各个核心业务系统的交易数据，而各个核心业务系统会在银行每天下班开始陆续将当天的交易数据从源系统导出，然后推送到历史库系统进行加载，并要求在银行第二天营业前完成数据加载。然而，核心业务系统每天产生的交易数据也有几百GB甚至TB，通过对历史库每天加载数据分析，三个分行每天产生的历史交易数据达到100GB 左右。每天需要完成如此海量数据的加载，给历史库系统的数据库OceanBase带了很大的挑战。

因此，在基于分布式数据库OceanBase的海量数据存储的应用背景下，我们需要针对OceanBase设计和实现相应的数据加载技术，根据历史库系统的数据加载需求以及加载任务特点，设计和实现一种高性能、高可用的海量金融数据的加载方案。

1.2 研究现状

近年来，很多学者和科研人员对海量数据的加载技术进行了广泛的研究，但这些加载技术研究大多集中于如何优化单个数据库中数据加载的性能，尤其在构建索引方面[5, 6]。文献[7]中提出了一种通用的批量数据加载技术，该加载算法并不依赖于底层的数据库。Robert等人[8]提出一种基于UB-Tree的批量数据加载模型，同时类似基于索引结构的批量加载已有很多研究，譬如R-Tree[9, 10, 11]、GridFiles[12, 13]、以及Quad-Tree[14]等。然而，上述的研究工作都仅仅针对单个数据库的批量加载进行优化，其加载性能仍不能满足海量数据的加载需求。

对此，各个数据库厂商也相继设计和研发了不同的批量数据加载的工具，譬如Oracle SQL*Loader[15]、DB2 LOAD[16]、以及SQL SERVER Data Transformation Services（DTS）[17]等。这些加载工具都是针对各自的数据库系统设计，尽管支持批量数据加载，但仍无法单独实现海量数据加载。

因此，一些学者提出一种基于分布式集群结构的数据加载技术，这种分布式结构的数据加载研究目前还处于起步阶段。Dora等人[18]针对海量天文数据的存

储和查询，提出一种优化数据存储框架SkyLoader。该研究主要针对海量天文数据的加载，根据特定的应用和表结构，并行化数据加载，将加载速率从2GB/h提高到13.3GB/h。文献[19]也研究了类似的问题，采用SQL Server和服务器集群构建SDSS SkyServer[20]数据仓库来存储和管理数据，其并行加载速率也达到5GB/h。随着分布式计算框架的出现，一些学者开始将数据加载过程中数据处理工作转移到分布式计算框架中，文献[21]针对并行数据仓库，提出了一种基于Hadoop的分布式数据加载方法。

这些加载策略均只适用于特定领域产生的数据，数据结构相对稳定，加载策略比较单一。因此，设计和实现高效的加载方法时，主要考虑加载算法、高效的数据结构和优化的并行机制等，而无需考虑加载策略的管理和同步问题。

1.3 本文工作

本文主要研究海量金融数据的快速加载，并以交通银行历史库系统作为研究对象，设计和实现一种多任务并发调度加载的数据加载方案，具体的研究工作如下：

1. 本文通过研究和分析历史库系统的业务和数据特点，给出了历史库系统所面临的海量数据存储和加载问题，并在现有的数据存储架构下，分析数据存储和数据加载的优缺点。交通银行采用分布式数据库OceanBase来搭建底层数据存储管理系统，从而解决海量数据存储的问题。本文通过研究和分析数据加载特点和存储架构，给出了以OceanBase 作为底层存储系统的新历史库系统所面临的海量数据加载的挑战。

2. 本文通过研究和分析OceanBase现有的数据加载的实现及其适用场景，得出现有的加载工具并不适用于历史库系统的数据加载的结论。根据数据加载需求和OceanBase特点，本文设计和实现了两种数据加载方法：基于SQL INSERT的并发加载和直接更新内存表的加载，这两种加载方法均支持将文本数据以增量或更新方式加载到数据库中。最后，本文通过实验测试两种加载方法的加载性能，并选取一种适用于历史库系统的数据加载方法。

3. 本文通过分析和研究数据加载任务的特点，分析并给出并行加载的可能性，并提出了一个多任务并行调度的数据加载方案。根据加载任务特点，结合已有的并行调度技术，本文为历史库系统的并行数据加载设计和实现两种并行调度策略：基于表级的任务调度策略和基于细粒度的两阶段调度策略，并通过实验测试两种调度策略的加载效率，并分析两种调度策略的适用场景。

1.4 本文结构

依据本文的主要研究工作，本文结构安排如下：

第二章，主要介绍本文研究工作的相关技术和概念，包括批量加载技术和多任务并发调度技术等。由于本文研究的工作主要建立在分布式数据库OceanBase上，在本章最后一节里我们将简单介绍OceanBase的实现架构和基本内容。

第三章，主要对本文的研究问题进行描述。本章通过研究分析历史库系统的数据存储和数据加载特点，提出了以分布式数据库OceanBase作为底层存储系统的新历史库系统所面临的海量数据加载的挑战，并给出了两种解决思路。

第四章，主要针对分布式数据库OceanBase设计和实现两种数据加载技术。本章中首先介绍了OceanBase现有的数据加载工具的实现及使用场景，然后详细介绍了本文的两种数据加载技术的设计和实现。

第五章，通过对数据加载的特点，提出一个多任务并行加载的数据加载方案。通过分析和研究加载任务特点，结合现有的调度技术，设计和实现两种并行调度，用于实现多任务并行加载的任务调度。

第六章对本文的研究工作进行总结。

第二章　基本概念和相关技术

2.1　批量数据加载技术

随着互联网的快速发展，企业系统功能不断地增强和更新，数据量也不断地增长，不同系统间需要大量的数据共享和迁移，这就需要在不同数据库之间进行数据加载。数据加载是指当数据库版本升级、不同数据库之间转换以及不同运行环境变更时，将源数据库中的数据及数据结构定义迁移到目标数据库中，且使之在新环境中正常运行。根据不同的应用场景，数据加载可以分三种类型：

- 版本升级的数据加载。
- 跨平台的数据加载，譬如从Windows到Unix。
- 不同数据库间（包括异构数据库间）的数据加载，譬如从DB2转入DB2或Sybase。

对于少量数据的加载操作可以采用SQL语句直接插入，但是对于海量数据的加载，就需要考虑数据库事务处理的优化，通常采用批量数据加载来降低事务处理量。批量数据加载的基本思想：将需要加载的数据切分成若干分块，然后依次地将每一个分块提交给DBMS，并由DBMS进行批量加载操作。由于数据并不是逐条地插入，这样不仅优化了事务处理量，而且大大地提高了数据加载的处理速度。

不同的应用对应的数据加载目的和需求也不相同，因此，根据应用目的不同，可以将批量加载划分为三类：初始化加载、定期加载和实时加载，也可按照加载需求同样将其分为三类：增量加载、更新加载和全量加载。对于不同的应用需求，可以采用不同的方法和技术来实现批量数据加载，从而满足不同系统的性能和实时性需求。文献[22]给出了四种不同的批量加载的实现方法和技术，下面将主要介绍四种加载技术的实现方法以及对应的加载工具。

- **数据文件转储形式**

采用文件形式的批量加载是将源数据库中的数据导出到数据文件，然后利用目标数据库的批量加载工具进行批量数据加载。这种方法适用于初始化数据加载和定期数据加载，即使源数据库与目标数据库不处在同一个局域网时，仍可以采用此种方法。采用文件存储实现数据批量加载是最常用、最简捷的方法，各个大型数据库系统都支持文件数据的批量加载，如Oracle、DB2等。

• 采用管道通信方式

管道通信（Communication Pipeline），即发送进程以字符流形式将大量数据导入管道，接收进程可从管道接收数据，二者利用管道进行通信。管道通信的数据加载是指在数据库间建立数据管道，通过管道通信来完成数据库间的数据传输和加载。这种加载方法主要针对局域网内数据库间的定期加载，同时也适用于异构数据库间的数据加载。

• 采用触发器方式

触发器（trigger），也称事件-条件-动作规则（event-condition-action rule），是指当表中数据发生修改时触发的一系列要执行的SQL语句集合[23]。触发器是一种特殊的存储过程，与一般存储过程不同，它的执行不是由程序调用，也不需要手动执行，而是由事件（event）来触发。由于数据加载操作可以通过执行SQL 语句完成，因此也可以采用触发器的方式实现实时数据加载。通常触发器是由数据库系统自行管理，因此，触发器方式加载数据既能保证数据的有效性和实时性，也能保证数据库的安全性。

• 采用Replication Server方式

服务器复制（Replication Server）是一种用于数据库备份的技术，但是同时也可作为一种批量数据加载的技术[24]。Replication Server是指源数据库到一个或多个目标数据库间的事务复制，其中源数据库和目标数据库可以是不同类型，如DB2、Oracle、Sybase等中任意一种。这种方式支持将数据从源数据库复制到另一个数据库，同时也支持多个源数据库的数据合并到一个目标数据库。

通常，Replication Server采用联机异步复制方式来完成两个数据库间的数据复制，适用于多种应用场景，主要有几个特点：

- 可靠性。Replication Server主要采用事务复制机制来实现数据复制，即在数据库之间进行事务传递，而非简单的数据传递。这种方式可以通过回滚的机制来确保数据传输的可靠性。
- 独立性。Replication Server是专门为数据库间的数据复制而设计的，并允许在数据传输过程中选择传输路由，进而充分利用网络，提高数据加载性能。
- 异构性。Replication Server对源数据库和目标数据库类型并没有限制，因此可以在异构数据库间进行数据复制操作。

- 本地自治性。数据复制是通过事务复制机制完成的，而整个复制过程的控制则由各个数据库自治决定。

由此可见，Replication Server主要适用于实时数据加载场景，当源数据库中主表的数据发生修改后，对应的额修改事务会在几秒内复制到目标数据库，并在目标数据库中执行。

2.2 并行任务调度技术

为了使加载任务可以更充分、更有效地利用加载服务器和数据库系统的计算资源（CPU、内存和网络），使得加载任务尽可能地并行执行，并在减少执行时间和提高加载效率的同时，实现加载的负载均衡，选择合适的加载任务调度策略及相应的实现算法就显得十分重要。

通常，一个任务调度系统可以大致分为三个模块：任务模型、并行计算模型和任务调度方法。任务模型是指任务间相互关系的一种形式化表示，主要包含了任务调度所需的所有信息。并行计算模型主要给出各个处理器的处理能力等一些信息。而调度算法则通过对任务模型和计算模型分析，得出调度结果[25]。图2.1显示了并行任务调度系统的结构图。

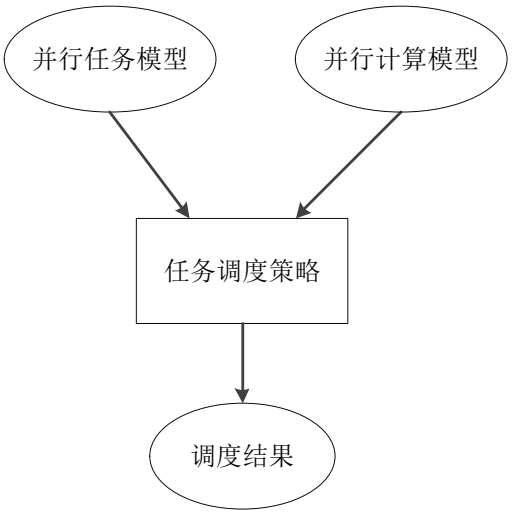

图 2.1: 任务调度系统的架构示意图

下面将从并行任务模型和任务调度方法来对任务调度技术进行分类和介绍。

2.2.1 任务调度模型

由于任务模型是任务调度的前提和基础，因此研究并行任务的调度问题首先要根据并行任务的实际情况来建立合适的数学模型。随着任务调度的研究和发展，树结构、Fork-Join等结构相继被用于构建任务调度模型，但这些结构都只针对一些特殊的调度场景。要想研究更广泛而多样化的任务调度问题，图结构将是一个很好的选择。目前几个典型的任务调度模型都是建立在图的基础上，这种基于图结构的任务模型称为任务图。

• 任务交互图

任务交互图（Task Interaction Graph，TIG）模型是基于所有任务可以同时执行且相互独立，并且任务间不存在执行上的时序关系的假设提出的。在任务交互图中，每个节点表示并行任务，而节点间的无向边则表示任务间的交互。通常，该模型适用于分布式系统中松散耦合任务的静态调度，具体来说，任务交互图适用于数据并行的应用，即各个处理器对不同的数据集执行相同的处理过程。

• 任务优先图

任务优先图（Task Priority Graph，TPG）模型假设任务在开始执行前必须接收到所有需要的输入数据，同时任务完成后将产生的输出数据传递给下一个任务。在任务优先图中，每个节点表示并行任务，而节点间的有向边则表示任务间的时序关系以及任务间的数据传输量。任务优先图充分地考虑了任务间的时序和依赖关系，即任务优先级，能够对大多数并行应用精确建模。

• 时序任务交互图

虽然TPG模型中已经考虑并行任务间的时序关系，但是有一些并行应用需要在计算和通信之间频繁切换，即相关任务并行执行过程中需要进行通信，而不只是在任务的开始和结束时通信。而TPG 模型只能反映不相关任务的并行化，并不能体现出相关任务间的并行化。基于此，Roig等人[26]对任务优先图TPG进行改进，提出了时序任务交互图（Temporal Task Interaction Graph，TTIG）模型。在时序任务交互图中有向边不仅代表通信量的权值，同时也有表示相关任务的并行度参数。因此，TTIG模型允许相关任务在并行执行过程中进行通信，其并行度由相关任务间的并行度参数来决定。

• 有向无环图

严格意义上来讲，有向无环图（Directed Acyclic Graph，DAG）只是一种广泛应用的图结构，并不能说是一种并行任务调度模型。由于上面介绍的任务优先图TPG模型就是建立在有向无环图DAG上，因此在很多情况下两个模型是等同的，并且很多文献中都采用DAG代替TPG模型。

2.2.2 相关任务调度策略

假设有p个节点参与执行一个划分为n个子任务的作业，把这n个子任务按照某种策略分配到这p个节点上执行的过程称为任务调度。

有效的任务调度策略可以充分利用计算资源，使任务具有较高的并行度，进而提高并行任务的整体执行效率。文献[25]根据任务特点、调度时间，对任务调度技术进行划分：

- **抢占式调度和非抢占式调度**

非抢占式调度是指任务执行过程中不允许被中断，直到任务运行完成才释放计算资源。而抢占式任务调度则是在保证任务最终正确执行完成的情况下，允许任务在执行过程中被中断，而由其他优先级高的任务抢占处理器的计算资源。

- **迁移式调度和非迁移式调度**

非迁移式调度是指任务在执行过程中不可以迁移到其他处理器执行。而迁移式调度则允许任务在执行过程中，根据不同处理器的负载情况迁移到其他处理器上执行。显然，后者能够更好地实现负载均衡，但实现过程也相对复杂。

- **集中式调度和分布式调度**

集中式调度是指所有的任务都由一个专门的调度处理器负责调度，而其他处理器只负责接收和执行任务。这种调度方法实现相对简单，但频繁的通信会导致较大的任务调度开销，同时单一的调度处理器也会影响调度系统的可用性。而分布式调度则将所有的调度工作分布到多个节点调度，避免单节点调度的瓶颈，同时具有良好的可扩展性，一般适用于动态调度。

- **静态调度和动态调度**

静态任务调度通常在调度前精确获取任务和处理器相关信息，然后调度节点评估已有的信息，将任务均匀地分配到不同处理器上执行。而动态任务调度则按照任务执行状态以及处理器的负载情况，动态地将任务分配到各个处理器上。由于动态调度需要适时监控和获取处理器负载及任务执行状态，因此该算法实现复杂，且调度开销较大。

还有一种介于静态调度和动态调度二者之间的调度算法，称为混合任务调度，其主要是在初始阶段采用静态调度策略调度一部分任务，余下的任务则是在程序运行过程中采用动态调度策略将任务分配到各个处理节点上。

本文只研究静态任务调度，对动态任务调度和混合任务调度不作讨论。

2.2.3 静态任务调度技术和算法

下面讨论不同的静态任务调度技术及其典型算法：

- **表（List）调度**

目前有很多调度算法都是基于表调度算法[27, 28]，表调度算法的基本思想：首先计算每一个任务的优先级，并将其按照优先级排序存储于列表；然后依次将列表中任务分配到最早可执行（或者执行代价最小）的处理器。通常，表调度算法也分成两类：静态表调度和动态表调度，二者区别在于，前者的任务优先级一旦确定后不允许在调度过程中更改，后者则充分地考虑调度过程中任务优先级的动态变化，每次调度后都会重新计算余下任务的优先级。表调度算法适用于并行计算系统，其核心在于任务优先级的衡量，常用的衡量策略有HLF（Highest Level First）、LP（Longest Path）、CP（Critical Path）等[29, 30]。与其它类型的调度算法相比，表调度算法能够以较低的调度开销获得较好的运行时间，其典型算法有HLFET （Highest Level First With Estimated Time）算法、ETF（Earlier Time First）算法、DLS（Dynamic Level Scheduling）算法等[31, 32]。

- **任务复制调度**

通常具有依赖关系的两个任务分配到不同处理器上时，任务间的通信需要在两个服务器间进行。而任务复制调度算法目的就是降低任务间的通信，其核心思想是在不同的处理器上冗余地复制某个或者某些任务的前驱，以达到减少处理器间的通信开销。该调度算法实际上通过复制前驱任务，避免任务间数据传输，从而减少处理器等待时间。除了任务复制外，该算法的另一个核心技术是如何选择复制策略，譬如，有的只复制其前驱任务，有的则会递归地复制其所有前驱任务。不同的复制策略形成了不同调度算法，如CPFD（Critical Path Fast Duplication）算法、DSH（Duplication Scheduling Heuristic）算法、BTDH（Bottom-up Top-down Duplication Heuristic）算法等[33, 34, 35]。

- **任务聚簇调度**

任务聚簇调度算法是将一个给定的任务图映射成多个任务簇，然后不断地合并簇，直到任务簇的数量与处理器的数量相等为止，并将这些任务簇分配到各个处理器上。如果两个任务被映射到相同的任务簇中，则代表它们将在同一个服务器上执行。同时，在每个处理器的内部，还需要确定任务的执行顺序。该调度方法需要考虑任务簇聚类合并的条件，其中基于动态关键路径（Dynamic Critical Path）的任务聚类是一种比较流行的算法[36, 37, 38]。对于不同的应用场景，也有其他典型算法：EZ（Edge-Zeroing）算法[39]和DSC （Dominant Sequence Clustering）算法[40]等。

- **非确定性调度**

非确定性调度技术又称为随机搜索调度技术，主要是基于之前的调度结果，然后采用某种随机策略对其修改生成新的调度结果。相比之下，随机搜索调度技术的调度结果比较好，但其调度过程实现相对复杂，且调度时间一般高于其他调度算法。非确定性调度算法主要包括遗传算法、模拟退火法以及局部搜索技术等，

其中遗传算法应用最广泛[41, 42, 43, 44]。由于本文不考虑该调度算法，这里不作详细介绍。

虽然上述内容对调度算法进行了明确分类，但是实际应用中很多调度算法都是基于两种调度技术的结合，如Ricardo等人[45]提出一种将表调度融入到遗传算法的调度算法。很多基于任务复制和任务聚簇的算法中都融合了表调度的思想，而本文采用一种将表调度思想融合到任务聚簇调度算法中的任务调度策略。

2.3 分布式数据库OceanBase架构

OceanBase[3]是一个支持海量数据的高性能分布式数据库管理系统，实现了数千亿条记录、数百TB数据上的跨行跨表事务。阿里巴巴研发的开源分布式无共享关系型数据库OceanBase，主要采用“OceanBase 数据库+主流PC服务器”取代“商业数据库+ 高端服务器+高端存储”，广泛应用于淘宝、天猫和支付宝线上业务，极大地降低了软件和设备的成本，其良好的伸缩性和自动故障恢复不仅很好地支撑了业务，而且也大大地降低了运行维护的人力成本。

作为一个分布式数据库，OceanBase主要包含事务处理引擎、分布式存储引擎和分布式查询引擎等组件。图2.2所示是单集群OceanBase整体架构图[46]。

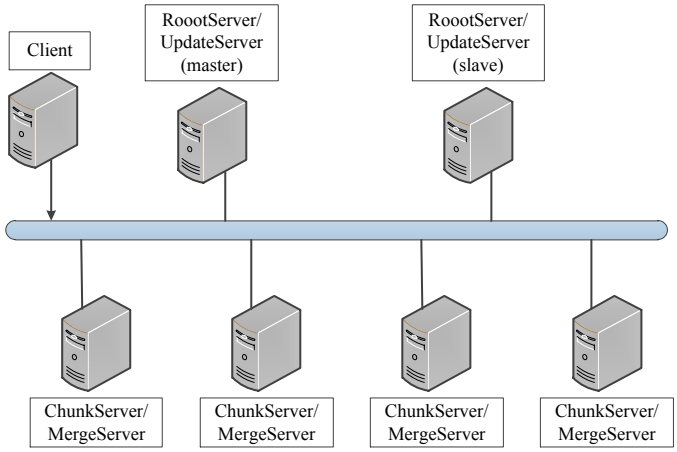

图 2.2: OceanBase单集群架构

OceanBase主要由4个核心模块构成：主控服务器（RootServer）、更新服务器（UpdateServer）、基线数据服务器（ChunkServer）、以及合并服务器（MergeServer）[47, 48]，其中：

- **主控服务器（RootServer）：** RootServer是OceanBase的控制中心。该服务器主要管理集群中所有服务器的上线/下线，以及数据分块（SSTable）和副本

的分布，并通过负载控制来保证数据分布均匀。

- **更新服务器（UpdateServer）**：更新服务器，顾名思义，用来存储数据库的增量更新数据。同时，UpdateServer也是OceanBase数据库中唯一支持事务处理模块，主要负责执行事务和写日志等操作，其功能相当于传统关系型数据库的事务处理引擎。

- **基线数据服务器（ChunkServer）**：ChunkServer主要存储OceanBase基线数据，并提供读访问。为了防止ChunkServer发生故障导致服务不可用甚至数据丢失，基线数据通常采用多副本冗余存储到多个不同的ChunkServer上。

- **合并服务器（MergeServer）**：MergeServer提供用户访问OceanBase的接口，其主要负责SQL解析、执行计划生成等操作。同时，MergeServer将执行计划分别发送相应的服务器上，并将查询结果合并后返回给用户。

如图2.2所示，OceanBase集群部署时，通常将UpdateServer进程和RootServer进程部署于同一物理服务器，而MergeServer进程和ChunkServer进程也部署于一台服务器。同时，为了保证高可靠性和高可用性，通常每个OceanBase集群分别配置主备两个RootServer和UpdateServer，主备之间可以采用不同的同步策略，但同一时刻只允许主UpdateServer提供写服务，而备UpdateServer仅提供读服务。在OceanBase中，数据存储分成两个部分：基线数据和更新数据，并分别由不同的服务器存储和管理。当用户请求读取数据时，需要将不同ChunkServer上的基线数据与UpdateServer上的更新数据合并后返回给用户。同时，由于更新数据存储于UpdateServer 内存，随着写操作的不断执行，更新数据量不断增加导致内存不断膨胀。为了避免内存负载过大而影响数据库正常服务，Oceanbase提出一种特有的操作——定期合并，即当UpdateServer内存占用达到一定阈值或者系统到达预设时间，就会将其内存中更新数据与当前基线数据进行融合，生成新的基线数据。在此架构下，OceanBase可以随时增加/减少服务器来应对业务和数据量的扩张，同时OceanBase也采用多机备份机制，可以随时进行自动、快速的故障恢复，保证了数据库的高可用性。

尽管，OceanBase定位为全功能的关系型数据库，但这并不代表其可以完美地替代传统关系型数据库。由于OceanBase还处于开发阶段，目前的设计和实现暂时摒弃了一些传统数据库的功能，如二级索引、游标、存储过程等。同时，OceanBase不支持复杂的SQL语句和查询优化，对某些OLAP应用的数据加载的支持也不完善。因此，OceanBase还需要进一步的开发和完善。

2.4 本章小结

本章主要介绍了本文所涉及到的相关技术以及现有的相关研究工作，包括数据批量加载的特点以及相关实现方法的介绍与分析，和并行任务调度技术的介绍和分类。首先，介绍了批量加载的意义，并给出了4 种常用的批量加载的实现方式。然后，对并行任务调度进行分类介绍，并对现有的调度技术进行介绍和分析。由于本文针对OceanBase进行海量数据加载，并在最后对OceanBase 系统架构及功能进行简单介绍。下面我们先对本文所要研究的问题和面临的挑战做出完整的描述。

第三章　问题描述

随着用户使用和业务处理的增长，企业级数据库的数据量也不断地增加，尤其是金融行业。目前，绝大多数的金融企业都采用“高性能服务器+商业数据库+高端存储”来构建底层数据存储系统，这种架构下的数据存储方式缺乏良好的可扩展性，难以应对海量数据的存储和查询。为了满足良好的存储可扩展性以及高并发访问需求，交通银行率先选择分布式无共享关系型数据库OceanBase，以“OceanBase数据库+主流PC服务器”取代“DB2数据库+IBM小型机+EMC高端存储”，解决了数据存储问题的同时也降低了成本开销，但同时也带来了一些新的问题和挑战。

本章首先对交通银行历史库系统的基本情况加以简单介绍，并且分析了历史库系统的数据特点，根据此特点提出了历史库系统在海量历史交易数据的存储和加载上所面临的严峻挑战。然后，分别介绍了历史库系统针对海量数据存储和加载的解决方案，分析基于分布式数据库OceanBase的历史库系统面临的数据加载问题。最后，针对现有的数据加载问题，我们提出了两种解决思路。

3.1　交通银行历史库系统概述

交通银行（全称：交通银行股份有限公司）作为中国五大国有控股商业银行之一，是中国境内主要综合金融服务提供商之一。作为一个国有控股商业银行，拥有着几亿的用户，这些用户每天的日常生活中都会通过交通银行进行交易和查询。据统计，2013年元旦假期间，电子银行渠道共处理交易940万笔，电话银行自助语音（VIR）共受理用户查询53万笔。面对如此大量的交易和查询，交通银行根据不同业务需求构建不同的业务系统，通常可以分为两大类：核心业务系统和历史库系统。所谓核心业务系统，即线上交易系统，主要通过业务处理和交易结算来处理银行各类业务的核心交易系统。整个核心业务系统根据业务类型，又分为不同业务子系统，主要有主机应用系统（GEMS）、主机账务系统（GAPS）、电子网银系统（EBANK）、远程银行（EPIC）、贷记卡（DJK）等。在银行业务中，各个子系统之间并不是完全独立的，用户查询和交易都会涉及到不同的子系统，

譬如用户通过交通银行的电子网银进行一笔资金交易，整个交易过程可能会涉及到电子网银系统、核心账务系统、贷记卡系统等，并在对应的系统中产生一条新的交易记录。

而历史库系统则是面向银行各类业务的历史交易查询系统，用户和企业可以通过查询历史库系统获得各类业务的详细历史交易信息。历史库系统主要存储用户所有的历史交易明细，每天的交易数据会定期地加载到历史库中，以供用户查询历史记录。历史库系统的数据加载是指将核心业务系统每天产生的所有交易记录，在银行当天停止营业到第二天营业前加载到历史库系统，以便用户在第二天可以查到前一天或者更久之前的历史交易明细。通常情况下，银行的历史库系统中会保留用户3-5年的历史交易记录。

由上述介绍分析可知，历史库系统的数据存储和加载主要有以下几个特点：

1. 数据来源多样性。历史库系统的数据通常来源于核心业务系统的历史交易记录，而核心业务系统主要处理各种不同业务的交易。由上述可知，核心业务系统是由多个不同的业务子系统组成，不同子系统产生的数据格式也是多样化的，并且每个系统数据推送时间也不相同。

2. 数据量大。历史库系统通常保存3-5年的用户历史交易信息，数量达到几百TB甚至PB级别，而核心业务系统每天产生新的交易数据也有几百GB甚至TB。通过统计交通银行历史库的每天加载数据可知，两个分行每天产生的历史交易信息达到100GB。

3. 加载效率要求高。通常，各个核心业务系统产生的交易数据会在银行每天下班开始陆续从源系统导出，然后推送到历史库系统进行数据加载，并要求在银行第二天营业前完成数据加载。一般情况下，历史数据加载过程持续在第一天的18:00 到第二天的8:00。

如此多样化的海量历史数据的存储和加载，给交通银行历史库系统提出了严峻的挑战。在接下来的章节里，我们将介绍历史库系统的数据存储和加载的实现，分析现有的实现方法存在的问题，并提供新的解决方案。

3.2 历史库的数据存储实现

目前，绝大多数的金融企业都采用传统的商业数据库来存储和管理数据，其数据存储层都采用“IBM小型机+AIX操作系统+DB2/Oracle数据库+EMC存储阵列”的实现方式。这种实现方式通常利用高端存储解决方案来完成海量数据的存储，同时利用高端服务器的高性能和商业数据库的高可用性和高可靠性来提供数

据的管理和查询。随着信息化时代的到来，金融行业数据库的数据量不断地增加，其数量级达到数百TB甚至PB级别，同时还需支持数十万TPS和数百万QPS的访问量。无论是数据量还是访问量，即使采用高端的小型机甚至大型机，单个关系型数据库系统都无法承受。对于海量数据的存储和查询，金融行业的一种常见的解决方法是根据业务特点对数据库进行水平拆分，通常的做法是根据不同的业务特点划分成不同的子系统，并将不同的子系统的数据分别存储于不同的数据库中。如果某个子系统的数据量仍然较大时，还可以根据子系统的特点采用分库分表方式进行细粒度切分。

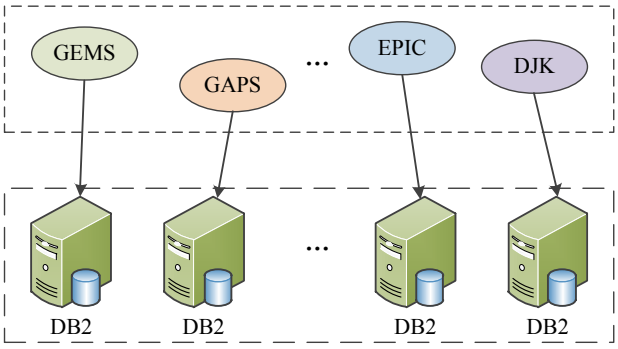

图 3.1: 基于DB2数据库的存储架构

在此之前，交通银行也采用上述解决方案来构建历史库系统的存储架构。由于IBM DB2数据库以其特有的数据库分区技术和数据压缩技术，为企业不断增长的数据提供了出众的数据存储解决方案，因此交通银行的历史库系统选用DB2作为底层的数据库管理系统，并通过水平拆分的方式分别存储和管理不同子系统的数据。图3.1给出了基于DB2数据库的历史库系统的存储框架，其中数据存储层是由多个相互独立的DB2数据库服务器组成，每个服务器上负责一个或者多个不同子系统。当数据量和访问量继续增长时，这种存储方式则需要继续切分数据库或者数据表，通过添加更多的DB2数据库服务器来存储和管理更多的数据。通常，这种方法需要数据库管理员（DBA）大量的手动干预，同时也带来巨大的硬件开销和人员维护成本。

为了避免海量数据存储带来的巨大开销，一些学者提出了一种分布式数据存储的解决方案。该方法借鉴分布式表格系统（Google BigTable）的实现思想，采用廉价的PC服务器搭建数据存储平台，将海量数据切分成固定大小的数据分块（SSTable），分别存储于不同服务器上，并通过多副本冗余和数据迁移等技术保证数据的高可用性。鉴于此，交通银行也考虑采用该存储策略来解决历史库系统的海量数据存储问题，并分别对一些主流的分布式数据库进行调研和分析。最终，

交通银行选用分布式数据库OceanBase作为历史库系统的底层数据库管理平台，并将所有的子系统全部存储于同一个OceanBase集群，如图3.2所示。

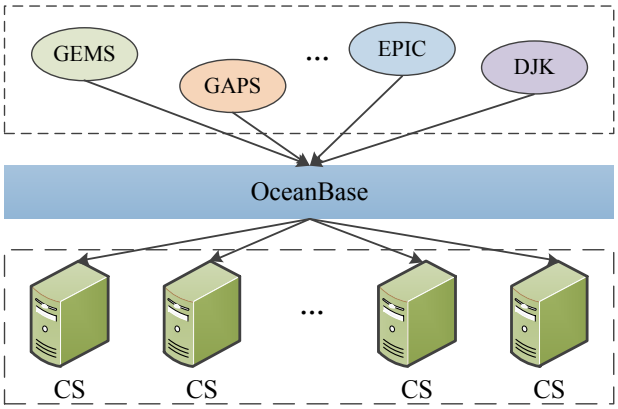

图 3.2: 基于OceanBase数据库的存储架构

如2.3节介绍，分布式数据库OceanBase也是采用上述方法来实现海量数据的分布式存储。OceanBase主要采用主流廉价的PC服务器提供服务，并利用分布式存储技术实现海量数据的存储和查询，其良好的可扩展性为海量数据的存储提供了保障。当数据库中的数据量不断地增长时，OceanBase只需要水平地增加基线数据服务器ChunkServer，即可满足数据量快速增长带来的存储需求。同时，这种方法不需要数据管理员过多的操作和干预，而且基线数据服务器通常采用廉价的PC服务器，也可以减少硬件开销。因此，分布式数据库OceanBase构建底层数据存储系统，可以很好地解决历史库系统的海量数据存储问题。

3.3 历史库的数据加载实现

从3.1节中可知，历史库系统的数据来源于线上核心系统，而每天核心交易系统产生的数据都将通过网络传输到历史库系统。然而，通过网络将大量数据从一台数据库服务器迁移到另一台数据库服务器非常困难。对于不同数据库间的海量数据迁移，各大数据库厂商提供了很多解决方案以供数据库管理员使用。图3.3给出了4种常见的数据迁移解决方案，包括：本地导出和本地加载策略、本地导出和远程加载策略、远程导出和本地加载策略、使用管道导出和本地加载策略。其中：

- **本地导出和本地加载策略**

  本地导出和本地加载策略是指在源数据库服务器上将源数据库中的数据导出到本地文件，然后通过网络将数据文件传输到目标数据库服务器上，最后

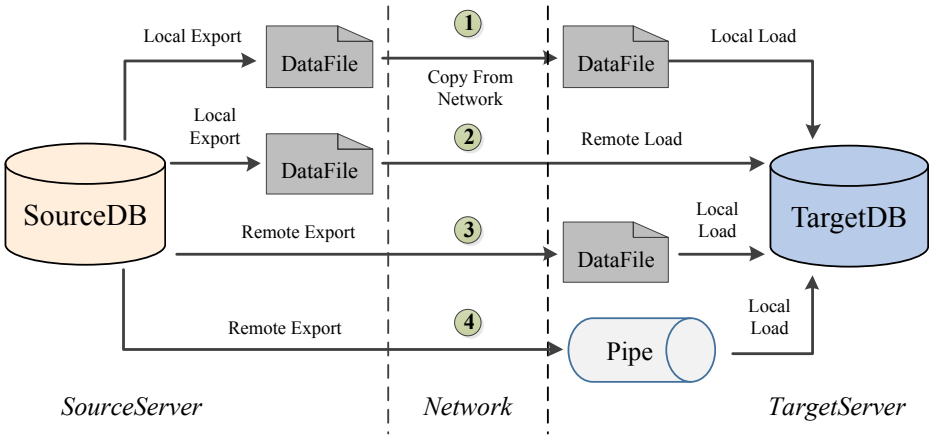

图 3.3: 4种常见的数据迁移策略

在目标数据库服务器上将数据文件加载到目标数据库中，如图3.3中方案1所示。这种方法的优点在于操作简单，适用于任何类型数据库间的数据迁移。但是，这种方法需要使用源服务器和目标服务器上的存储空间来保存数据文件。

- **本地导出和远程加载策略**

本地导出和远程加载策略，如图3.3方案2，是指在源数据库服务器上将数据导出到本地文件，并执行远程加载将数据文件远程导入到目标数据库中。这种方法的所有操作都在源数据库服务器上执行，并且只需要在源数据库服务器上保存数据文件。但是这种方法的加载过程主要使用源数据库服务器的系统资源，当源数据库提供实时应用时，这种方法大多数情况下并非理想方法。

- **远程导出和本地加载策略**

与前两种解决方案类似，远程导出和本地加载策略是指在目标数据库服务器上执行远程导出，将源数据库中数据导入本地文件，然后在将本地数据文件加载到目标数据库中。该数据迁移方案的主要流程见图3.3方案3。这种迁移策略不需要考虑源数据库，所有操作都在目标数据库上执行，同时也只占用目标服务器的存储空间。但是远程导出过程会增加目标服务器的负载，也会降低数据加载的效率。

- **使用管道导出和本地加载策略**

使用管道导出和本地加载策略是指在目标服务器上执行远程导出，将源数据

库的数据从操作系统管道一端导入，然后从管道另一端导出并加载到目标数据库中，其主要操作流程见图3.3中方案4。该数据迁移方法主要是利用管道通信技术来克服使用文件通信的问题，其实现相对复杂。如果源服务器与目标服务器间的管道断开，则需要启动导出与加载刷新来继续完成数据加载。

然而，交通银行的核心业务系统和历史库系统间的数据迁移过程采用上述的第一种解决方案。原因是，当源数据库或者目标数据库发生改变时，只要新数据库系统支持文件形式的数据导出或加载，只需更改一方的导出或者加载方法，整个数据迁移过程仍可继续执行。由于核心业务系统基本上不会变更，因此我们只需要考虑历史库系统的数据加载实现。在基于DB2数据库构建的历史库系统中，核心业务系统中的各个子系统将每天的交易导出到数据文件中，然后分别推送到历史库系统中对应的子系统所在的数据库服务器上，最后再由各个子系统完成数据加载。根据上一节的介绍可知，不同子系统所在的DB2数据库服务器是相互独立的，因此，彼此之间的数据加载过程也是相互独立的，并且可以并行执行。同时，DB2数据库提供了丰富的数据加载选择，主要有IMPORT、LOAD和DB2MOVE等数据加载工具，为历史库系统的数据加载提供了有利的保障。因此，基于DB2数据库的历史库系统通过在各个子系统间并行执行数据加载，来完成完成每天的历史交易数据的加载。

为了解决海量数据存储的问题，新历史库系统采用分布式数据库OceanBase替换DB2数据库，并将所有的子系统部署在同一个OceanBase集群进行统一管理。同时，这种新的存储架构也带来了新的问题和挑战。如何将每天海量的交易数据在指定时间内加载到分布式数据库OceanBase中成为新历史库系统迫切需要解决的问题。在基于OceanBase的历史库系统中，所有的子系统的数据加载都针对同一数据库，因此，数据加载并行执行时需要更多地考虑数据库的负载。同时，由于OceanBase数据库尚仍处于进一步完善阶段，其对于数据加载的支持也不完善。基于此，为了保证新历史库系统在指定时间内完成海量数据加载，我们提出了两种解决思路：

1. 首先，设计并实现一个针对OceanBase数据库的数据加载工具，支持将文本数据快速直接地导入数据库中。

2. 然后，分析历史库系统的数据加载流程，设计和实现一种并行加载策略，并通过任务调度技术将海量文本数据并行地加载到OceanBase数据库中。

再接下来的两章里，我们将重点讨论研究和设计OceanBase数据加载技术以及基于OceanBase历史库系统的海量文本数据加载的解决方案。

3.4 本章小结

本章主要对本文的研究问题进行描述。首先，我们介绍了交通银行历史库系统的应用背景，并根据其应用特点和数据特点，分析了历史库系统的数据存储和加载面临的挑战。然后，我们分别介绍了基于DB2数据库和基于分布式数据库OceanBase的数据存储层的实现，进一步说明基于OceanBase的存储架构可以很好地解决历史库系统的海量数据存储问题。最后，我们介绍了历史库系统的数据加载策略，并提出基于OceanBase的历史库系统所面临的数据加载挑战，同时给出了两种数据加载的解决方案。

第四章　OceanBase数据加载技术实现

在上一章中，我们介绍了交通银行历史库系统的特点，分析了在当前系统架构下的历史库系统所面临的海量数据存储的问题。新的历史库系统选用分布式数据库OceanBase解决了海量数据存储的问题，但是同样带来了海量数据加载的问题。

本章首先介绍了分布式数据库OceanBase已有的海量数据加载方法，并分析该方法的优缺点及适用场景。然后根据交通银行历史库系统的数据加载需求，设计和实现两种OceanBase批量数据加载方法。最后，通过实验对两种不同的加载方法进行性能测试，对比两种方法的加载时间，从而选出最优的数据加载方法。

4.1　ChunkServer旁路数据导入

很多企业都有一些在线数据分析处理（OLAP，又称联机分析处理）应用，这些应用往往需要定期（例如每天或每个月）导入大量历史数据，对数据加载性能要求很高，譬如交通银行历史库系统。为了快速将海量数据导入数据库中，OceanBase专门开发了旁路导入功能，即直接将数据导入到基线数据服务器ChunkServer中的方法，又称ChunkServer旁路导入。本节将介绍ChunkServer旁路导入的基本思想及其实现技术，并分析该加载方法的优缺点以及适用场景。

4.1.1　基本思想

ChunkServer旁路导入主要利用Hadoop将数据文件处理成OceanBase物理存储格式，然后再分别将生成的数据文件分发到不同基线数据服务器ChunkServer上。在OceanBase数据库中，数据是按照主键全局有序排列的，因此，旁路导入的第一步就是利用MapReduce对所有的数据进行排序。由于OceanBase中的数据是按照主键范围划分成若干份存储于基线数据服务器ChunkServer上，并且每个范围都对应一个SSTable数据文件，通常一个SSTable数据文件大小为256MB。因此，需要对排序的数据进行抽样获取数据分布，然后根据数据分布情况将数据划

分成一个个有序的范围，同时利用MapReduce为每个范围生成一个SSTable数据文件。接下来，再将生成好的SSTable文件并行拷贝到OceanBase集群中所有的基线数据服务器ChunkServer上。最后，主控服务器RootServer通知每个基线数据服务器ChunkServer并行加载本地SSTable文件。在OceanBase中，每个SSTable文件相当于一个子表，当基线数据服务器ChunkServer完成本地加载后会向主控服务器RootServer汇报，RootServer则将汇报的子表信息更新到RootTable中。

例4.1.1. 假设*OceanBase*集群有*4*台基线数据服务器*ChunkServer*：CS_1、CS_2、CS_3、CS_4。数据排序后划分为*4*个*range*：$r_1(0,25]$、$r_2(25,50]$、$r_3(50,80]$、$r_4(80,100]$，其对应的*SSTable*文件分别为sst_1、sst_2、sst_3和sst_4。假设当前*OceanBase*设置副本数为*3*，那么拷贝*SSTable*文件后，可能的分布情况为：

$$CS_1 : sst_1, sst_2, sst_3$$
$$CS_2 : sst_1, sst_2, sst_4$$
$$CS_3 : sst_2, sst_3, sst_4$$
$$CS_4 : sst_1, sst_3, sst_4$$

然后，每个*ChunkServer*分别加载对应的*SSTable*文件，所有*SSTable*加载完成后向*RootServer*汇报。*RootServer*最终会将这些分布信息保存到*RootTable*中，如下：

$$r_1 : CS_1, CS_2, CS_4$$
$$r_2 : CS_1, CS_2, CS_3$$
$$r_3 : CS_1, CS_3, CS_4$$
$$r_4 : CS_2, CS_3, CS_4$$

4.1.2 ChunkServer旁路导入实现

图4.1显示了ChunkServer旁路导入的整体架构。旁路导入主要由5部分组成：Hadoop集群、OceanBase集群、ImportServer、ProxyServer和Importcli。其中，Hadoop[49]是一个根据MapReduce和GFS（Google File System）实现的可运行于廉价PC服务器组成的集群上的分布式计算系统，能够以一种可靠、高效、可伸缩的方式对大量数据进行处理。OceanBase的详细介绍见第2.3节。而ImportServer、ProxySever和Importcli才是旁路导入的核心组件，主要负责发起和控制整个旁路导入的过程。下面将介绍三个核心组件的功能以及旁路导入的工作流程。

如图4.1所示，旁路导入的整个工作流程主要6个步骤，如下：

1. 由Importcli向ImportServer发起导入命令。

2. ImportServer接受命令生成mapreduce作业，然后将作业发到Hadoop上执行生成相应的SSTable数据文件。

3. 数据文件生成完毕后，ImportServer发送命令让RootServer准备加载数据文件。

4. RootServer收到加载请求后，首先通过ProxyServer获取生成的SSTable Range。

5. RootServer将Range分发给不同ChunkServer，并通知ChunkServer根据Range拉取对应的SSTable数据文件。

6. 当所有的SSTable都加载完成后，RootServer会更改表的table_id，使得新导入的数据生效。

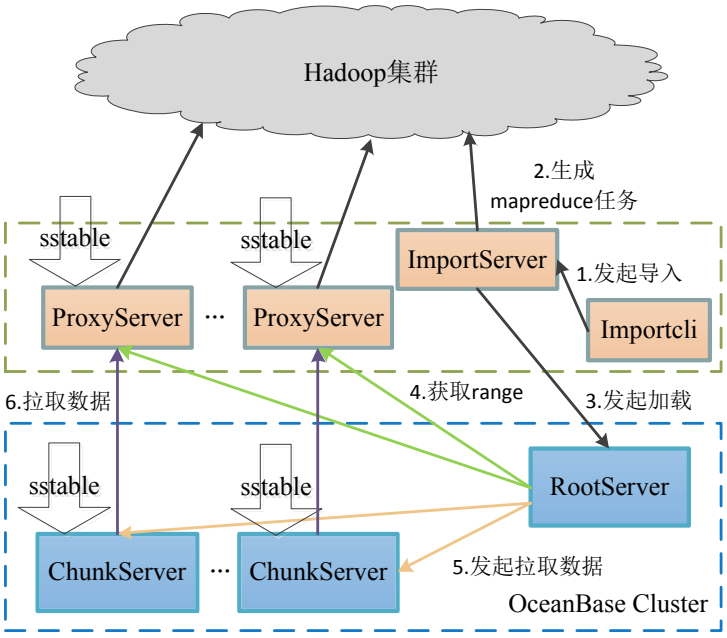

图 4.1: ChunkServer旁路导入流程示意图

由上述可知，ImportServer和ProxyServer负责隔离Hadoop集群和OceanBase集群，其中ImportServer负责数据加载的控制，而ProxyServer则起到数据中转的作用。通常，我们将Importcli和ImportServer部署在主控服务器上，而ProxyServer则被部署在每一台基线数据服务器上。

旁路导入主要利用Hadoop的高性能优势代替OceanBase完成生成SSTable工作，直接将数据加载到基线数据服务器ChunkServer上。该方法可以降低更新服

务器UpdateServer的压力，避免主键重复检查、事务处理等操作所带来的开销，但也给旁路导入带来一些问题及限制。由于旁路导入的数据不经由更新服务器UpdateServer处理，因此所有的插入操作均不产生任何日志信息。一旦系统出现故障，数据库无法通过回放日志来修复数据。并且旁路导入过程不对数据进行主键重复检查，因此旁路导入不能进行增量加载或者更新加载。目前，旁路导入仅支持两种应用场景：建表初始化时全量数据的加载和每天的全量加载。而对于交通银行历史库系统来说，每天只需要加载当天的交易数据，属于增量加载和更新加载，ChunkServer旁路导入的方法并不适用于历史库数据的加载，因此，我们需要设计和实现一种支持增量加载和更新加载的方法。

下面，我们将介绍两种不同加载方法的设计和实现，这两种加载方法均满足历史库系统的加载需求。

4.2 基于SQL INSERT加载技术

数据加载的最简单、最直观的方法就是执行SQL INSERT语句，该方法通过执行SQL语句将数据逐条插入数据库。传统商业数据库都支持这种数据加载的方式，譬如DB2 IMPORT就是利用该方式将数据文件中的数据逐条插入数据库中，而对于分布式数据库OceanBase来说，执行SQL INSERT语句加载数据的方法也同样适用。通常，这种方法适合加载少量数据，对于大批量的文本数据，这些加载工具通常利用多线程并发执行来提高数据加载效率。然而，执行SQL INSERT方式会占用大量CPU处理时间，而传统数据库都部署在单个服务器节点上，高并发的数据加载会增加数据库服务器的负载，进而会给其他应用访问数据库带来一定的延迟。对于分布式数据库OceanBase，为了避免高并发带来的高负载问题，我们提出一种通过负载控制实现并行数据加载的方法。

4.2.1 基本思想

在分布式数据库OceanBase中，传统数据库的读写事务所需的操作步骤被切分到不同的服务器上执行，如2.3节中所描述，合并服务器MergeServer负责SQL解析、逻辑计划和物理计划生成等操作；更新服务器UpdateServer 则负责执行读写事务、存储更新增量、记录日志等操作。对于OceanBase数据库，采用SQL INSERT方式插入数据时需要将SQL发到MergeServer执行，而多线程高并发的插入则会导致MergeServer负载变高。通常，OceanBase集群中会部署多个MergeServer，并且每个MergeServer 都能够独立地对外提供服务。因此，我们可以将高并发INSERT SQL分配到多个MergeServer执行，进而减低单个MergeServer的负载。

基于此，我们提出一种基于负载控制的并行数据加载方法，通过将INSERT SQL分发到多个MergeServer执行来降低单个节点负载，并根据负载控制确保每个节点负载均衡。该方法的主要思想：首先获取OceanBase集群中所有MergeServer信息，对每个MergeServer构建连接池并统一管理；然后监控每个连接池中连接数量，获取MergeServer当前的负载；最后根据所有的MergeServer的负载情况，将INSERT SQL均衡地分发到不同MergeServer上执行。

实现该方法的关键技术在于：获取集群信息、构建线程池、以及负载控制。我们采用OceanBase的Java客户端和淘宝开源的数据库连接池Druid来完成这些功能，图4.2给出基于SQL INSERT并行数据加载的流程，其中：

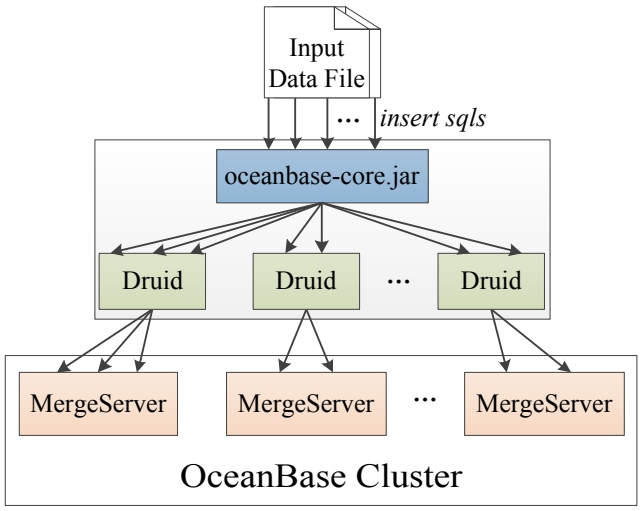

图 4.2: 基于SQL INSERT加载流程示意图

- **oceanbase-core.jar模块**

  oceanbase-core.jar模块根据系统内部表完成相关初始化任务，包括获取主备集群角色、流量分配、获取主备集群中各个MergeServer的信息和连接池的初始化、以及获取每个集群中MergeServer的负载策略。该模块还会定时检测集群变更并使新变更信息生效，以及输出各个MergeServer连接池状态。

- **Druid模块**

  Druid模块是阿里巴巴开发的开源数据库连接池。每个Druid数据源对应一个MergeServer，用于管理和维护该MergeServer 的所有连接。Druid是Java语言中最好的数据库连接池，其能够提供强大的数据库连接管理和性能监控等功能。在这里，我们利用Druid来管理数据库连接，关于Druid详细介绍可参考GitHub开源主页[50]。

4.2.2 详细设计

由上述可知，实现该方法的关键技术在于获取集群的MergeServer信息和选择负载策略。这两部分的功能均在oceanbase-core.jar模块中实现，下面分别对其实现进行详细介绍。

获取集群信息是通过与OceanBase集群内部表进行交互，主要是“__all_cluster”和“__all_server”两个内部表。表4.1给出两个内部表决定负载均衡的核心字段说明。

表 4.1: __all_cluster和__all_server核心字段

(a) __all_cluster

字段	说明
cluster_id	主备集群标志
cluster_vip	集群入口lms(Listen MergeServer)访问地址
cluster_port	lms访问端口
cluster_role	集群角色（“1”为主集群，“2”为备集群）
cluster_flow_percent	集群流量分配占比（%）
read_strategy	负载策略（0-轮询，1-一致性，2-随机）

(b) __all_server

字段	说明
cluster_id	指定集群，与“__all_cluster”中cluster_id相关联
svr_type	服务器类型，这里只查询值为“mergeserver”的记录
svr_ip	服务器访问地址
svr_port	服务器访问端口

而获取的方式采用执行SQL查询的方式，即分别执行SELECT SQL获取“__all_cluster”和“__all_server”两张表的核心字段。因此，负载监控过程与内部表交互流程如下：

1. 根据集群入口lms连接到OceanBases数据库。
2. 访问“__all_cluster”表，执行“select cluster_id, cluster_role, cluster_flow_percent, cluster_vip, cluster_port, read_strategy from __all_cluster”获取集群流量分配和每个集群的负载策略。
3. 根据查询得到的“cluster_id”，继续访问“__all_server”表，执行“select svr_ip, svr_port from __all_server where svr_type = ’mergeserver’ and cluster_id =

?”获取所有MergeServer信息。

根据上述的信息，SQL INSERT数据加载方法可以通过一系列的负载控制实现插入语句的并发执行。该方法首先通过上述方式获取OceanBase集群信息，并根据MergeServer信息（ip，port）初始化数据库连接池（Druid）；然后根据集群流量分配和每个集群的负载策略将并发SQL INSERT语句分发到各个MergeServer执行，其负载控制过程见算法1。

Algorithm 1 基于负载控制的加载过程

输入： insert sql集合：$Q = \{Q_i | i = 1, 2, ..., n\}$;

```
init(db_conn_pool);
for all Q_i ∈ Q do
    cluster_id =alloCluster(Q_i);
    mode =readStrategy(cluster_id);
    ms =alloMergeServer(mode);
    execute(Q_i, ms, db_conn_pool);
end for
```

这种方法可以充分利用集群资源来提高数据加载效率，同时也能够确保数据库不会因为某一台MergeServer的负载过高而影响业务正常查询效率。

4.3 直接更新内存表加载技术

4.3.1 基本思想

对于任何数据库系统来说，采用SQL INSERT方式插入一条记录大致需要经过SQL解析、逻辑计划和物理计划生成、查询优化、事务处理、记录日志等步骤。在传统的集中式数据库中，这些操作都是在单节点数据库服务器上执行的。如2.3节中所描述，在分布式数据库OceanBase中，这些操作分别被切分到不同类型服务器（MergeServer、ChunkServer和UpdateServer）上执行，而这些不同类型的服务器之间主要通过网络进行通信。如图4.3所示，在OceanBase数据库中，执行一条插入SQL语句至少要经历2次网络传输。

因此，上一节中介绍的基于SQL INSERT加载方法，每插入一条记录就需要经历至少2次网络传输。采用该方法加载大量数据时，频繁的网络传输将限制数据加载的性能和效率。而OceanBase中新插入的数据都会写到UpdateServer 上，MergeServer只在整个数据插入过程中充当一个“中转站”的角色。为了减少网络

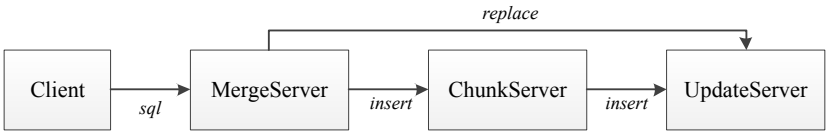

图 4.3: insert/replace执行流程

传输次数，我们提出一种将数据直接加载到UpdateServer上的方法，这样可以节省数据传输到“中转站”MergeServer的开销，进而提高数据加载的效率。

4.3.2 详细设计

根据2.3节中介绍可知，更新服务器UpdateServer是分布式数据库OceanBase的事务处理引擎，同时也是OceanBase集群中唯一能够接收写入的模块。当用户向OceanBase数据库插入一条记录时，UpdateServer接收到插入操作后将记录处理并存放到内存数据结构中，成为内存表MemTable。内存表MemTable是内存事务引擎用于存储数据和处理查询的核心数据结构，其数据以行为单位存储。实际上，MemTable底层是一个高性能内存B+树，而B+树中每个叶子节点对应一条记录，其中叶子节点的Key为数据行的主键，Value为行操作链表的指针。在MemTable中，每条记录的增删改操作会按照时间顺序构成一个操作链表，并存于B+树叶子节点中。因此，MemTable可以通过数据行的主键索引提供查询功能，包括B+树范围索引和Hash单行索引。

如图4.4所示，MemTable内存结构主要包括两部分：索引结构和行操作链表。其中，索引结构上面提到的B+树结构，支持插入、更新、随机读取以及范围查找等操作；而行操作链表则保存某一行各个列（每个行和列确定一个单元，称为cell）的修改操作。

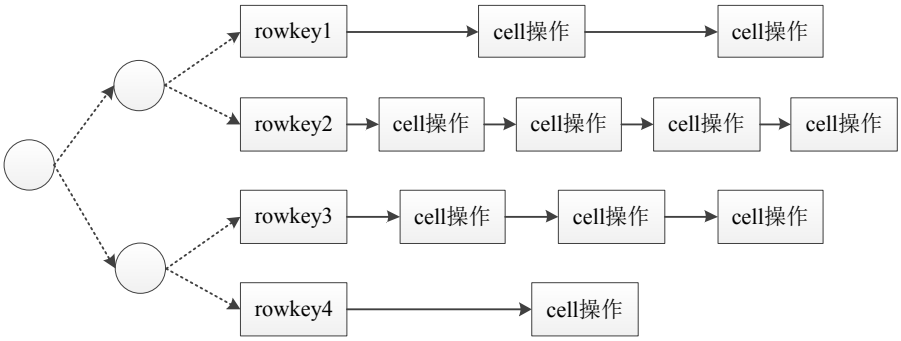

图 4.4: MemTable内存结构

因此，为了能够直接将数据加载到UpdateServer中，我们提出了一种直接更新内存表（Direct Update MemTable，DUMT）数据加载方法。DUMT加载方法的基本思想：数据加载程序根据内存表B+树数据节点数据结构，将每条插入的记录直接构造成一个数据节点，然后将该节点直接发到UpdateServer上更新内存表B+树。当UpdateServer更新内存表MemTable时，首先遍历B+树查找更新记录行是否已存在B+树中，如果存在的话，就将更新节点的cell操作链表直接挂到已有操作链表后面；否则，将更新节点插入B+树中。

由于客户端和UpdateServer之间通过网络传输进行数据加载，为了充分利用网络传输提高数据加载效率，我们将加载任务切分成多任务并发执行，每个任务独立负责将记录行构建成B+树叶子节点，然后并发提交给UpdateServer更新MemTable。同时，考虑事务处理和网络传输，提出一种批量加载方法，即每个加载任务处理一批数据行，这些数据行作为一个事务一起发给UpdateServer执行，这样可以提高网络传输效率和减少UpdateServer事务处理量。

但是，对于事务处理引擎UpdateServer来说，并发批量数据加载就是对B+树的并行批量插入过程。对于B+树的并发操作，UpdateServer采用细粒度锁来处理并发写入，同时采用Copy-on-Write技术来实现读写互不阻塞[51]。目前，UpdateServer提供两种细粒度锁：S锁（Shared lock，共享锁，也称读锁）和E锁（Exclusive lock，互斥锁，也称写锁）。而与传统的B+树实现不同，尽管数据都存储在叶子节点，但为了简化并发处理的逻辑，内存表的B+树中并没有将叶子节点串联起来[52]。如下图4.5所示，插入Key-Value到B+树的过程主要有以下3个步骤：

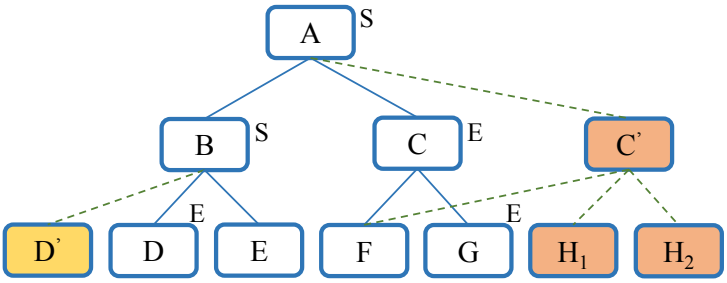

图 4.5: B+Tree插入数据示意图

1. 从B+树的根节点开始遍历Key-Value要插入的位置，并在遍历过程中对路径上的所有节点加S锁。

2. 将叶子节点的S锁升级为E锁，复制当前叶子节点，将Key-Value插入复制的叶子节点。如果叶子节点不发生分裂，则修改其父节点的指针指向拷贝的叶

子节点，如4.5图中叶子节点D的插入过程。

3. 如果叶子节点发生分裂，首先将其父节点的S锁升级为E锁，并复制父节点；然后将分裂后的叶子节点分别插入复制的父节点，如果父节点还需分裂则递归地执行此操作。见图4.5中叶子节点G的插入过程。

由此可知，共享锁以自顶向下的方式获取，而互斥锁则是随着节点修改自底向上加锁。为了避免出现死锁情况，每个事务采用try-lock方式来获取锁，如果检测到所冲突导致加锁失败，该事务将已经加锁成功的锁释放，然后再重新申请。因此，高并发批量插入数据过程会产生更多的锁冲突。尽管，UpdateServer已经通过两阶段并发控制和锁冲突调度技术来解决事务冲突，但大量的锁冲突会增加UpdateServer负载，同时也会降低加载效率。因此，我们希望在数据加载过程中尽量保证并发事务处理的数据不插入B+树的同一节点上。

有关树结构的批量更新和插入，已经有很多人在这方面进行了详细的研究和实现，尤其B树、B+树以及R树等[53, 54, 55, 56]。作为一种常见的数据库索引结构，B树或B+树更新和插入操作更多地涉及并发控制和加锁机制[57, 58, 59, 60]。文献[61]中提出一种基于排序的批量更新B树的算法，并给出一种新算法使其可以并发批量执行更新操作。而文献[56]提出一种应用于内存数据库的高效B+树索引结构，并采用辅助树方法结合树合并算法来有效地处理小范围数据的突发插入，以减少树再平衡开销。由于B树或B+树实现比较灵活，不同的系统中B树和B+树的实现也不一样，包括树的构建、节点分裂等策略的实现都可能不一样，因此，不同的系统需要根据自身索引实现机制来实现并发批量更新。

对于DUMT数据加载方法，我们提出一种naive改进方法，尽可能解决并发批量加载带来的冲突问题。该方法也是将数据排序后执行并行加载，其主要思想：首先将所有数据按照主键进行排序，然后将排序后的数据按照主键范围（Range）切分若干份数据分块（Chunk），不同并发线程分别处理不同数据分块。由于不同加载线程处理不同主键范围的数据，且主键范围间互不重叠覆盖，因此，可以降低并发插入事务间发生冲突的概率。

图4.6给出了直接更新内存表（DUMT）数据加载流程。

实现直接更新内存表（DUMT）加载过程大致需要4个步骤，包括数据排序和分块、解析数据文件、构建B+树数据节点和更新内存表MemTable。其中：

1. **数据排序和分块**

   根据表的数据模式读取记录，将记录按照主键排序，并将排序后的数据按照主键范围（Range）切分成若干份数据分块（Chunk）。

2. **处理Chunk数据**

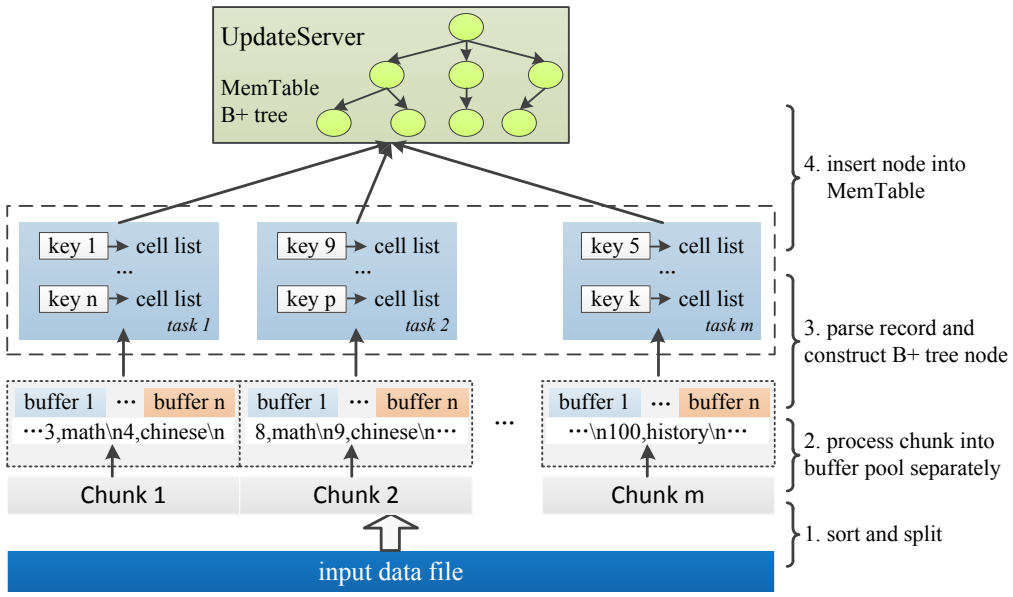

图 4.6: 直接更新内存表（DUMT）加载流程示意图

通常，数据文件以纯文本形式存储数据，并采用字符分隔值（Character-Separated Values，CSV）格式存储每行记录。处理Chunk数据是指根据行分隔符解析出记录，依次添加到固定大小的缓冲区，并对这些缓冲区统一管理。

3. **构建B+树数据节点**

在构建B+树叶子节点时，首先从一个缓冲区中依次获取并解析每行记录，获取到每个字段值。然后，根据表的数据模式，检查每个字段值的有效性和合法性。最后，构造成一个B+树数据节点，数据节点是由主键（key）和列操作链表（cell list）构成。

4. **更新内存表MemTable**

将第二步构建造好完成一批数据节点直接发送到UpdateServer并完成内存表MemTable的更新。

由于UpdateServer中B+树的数据并行插入过程是否发生冲突与B+树的结构有关，而目前OceanBase并不支持获取当前B+树结构，因此本文提出的改进方法只能尽量降低事务冲突的概率，并不能保证一定会减少事务冲突。

4.4 实验准备与结果分析

4.4.1 实验准备

4.4.1.1 数据集

本实验中，实验数据采用交通银行主机业务应用系统（GEMS）的历史数据。主机业务应用系统（GEMS）主要负责账务处理，在历史库系统中，主机业务应用系统主要有60多张表，共涉及60多个交易查询。实验选用gems_pdtdcrd表作为数据加载测试表，gems_pdtdcrd表共73个字段，主要涉及的数据类型有INT、VARCHAR、DECIMAL和DATE四种基本数据类型，每种数据类型的字段数量如表4.2所示。每条记录最多占用1178个字节（Byte），通过对实际的数据文件中记录分析，每条记录平均占用650多个字节（Byte）。最终，实验数据选用主机业务应用系统的某一天历史交易数据，其中gems_pdtdcrd表有4.3GB数据，共计700多万条交易记录。

表 4.2: gems_pdtdcrd表中数据类型的分布

数据类型	字段数量	所占比例（%）
INT	3	4
VARCHAR	52	71
DECIMAL	16	22
DATE	2	3

4.4.1.2 实验设置

本节所提出的所有数据加载方法最终在OceanBase（版本号：0.4.2.21）数据库上实现。由于目前还未有其他分布式数据库应用于金融行业，尤其是银行，所以无法在实验中为本文的实现找到一个对比测试组，只是测试了本文中不同加载的方法的效率。

为了进行本次实验，我们共准备了7台服务器，其硬件参数见表4.3。我们选取6台服务器节点组成的OceanBase单集群，其中一台服务器上部署主控服务器RootServer、更新服务器UpdateServer和一个监听合并服务器Listener MergeServer（lms，第一种加载方法需要通过该服务器访问和监听OceanBase集群），其硬件参数说明如表4.3(a)；余下的5服务器都用于部署基线数据服务器ChunkServer和合并服务器MergeServer，其硬件参数信息见表4.3(b)。

表 4.3: 实验环境说明

(a) UpdateServer配置

组件	说明	备注
CPU	Intel(R) Xeon(R) E5-2680*2 32核	16 核/CPU
主频	2.70GHz	
内存	378GB	主机内存
以太网	BCM5719 Gigabit Ethernet	千兆网卡

(b) MergeServer配置

组件	说明	备注
CPU	Intel(R) Xeon(R) E5-2630*2 24核	12核/CPU
主频	2.30GHz	
内存	94GB	主机内存
以太网	BCM5719 Gigabit Ethernet	千兆网卡

同时，为了使数据加载程序不对OceanBase集群产生影响，我们将所有加载程序单独部署到一台加载服务器，其硬件配置与MergeServer相同，见表4.3(b)。

4.4.2 实验结果与分析

在本节中，我们将对两种支持增量加载和更新加载的加载方法进行性能测试，并根据其加载时间和计算资源占用量来选择合适的加载方法用于交通银行历史库系统的数据加载。

4.4.2.1 SQL INSERT加载性能测试

图4.7展示了SQL INSERT数据加载在不同并发度和不同MergeServer数量下的加载时间。从图中可以发现，当使用一个MergeServer进行数据加载时，随着线程数的增加，加载效率也提高。但当并发度达到一定时，加载效率并不能继续增长。其原因如4.2节中介绍，高并发加载会增加单个MergeServer的负载，当并发线程数达到120时MergeServer的CPU使用率达到90%（见图4.8中1 MS的CPU使用率），因此仅提高并发数无法线性提升加载效率。

鉴于此，SQL INSERT通过负载均衡控制将高并发的数据加载SQL分发给多个MergeServer服务器执行，保证高并发加载的同时也降低了MergeServer负载。图4.8显示了不同数量的MergeServer下并发加载时MergeServer的CPU使用

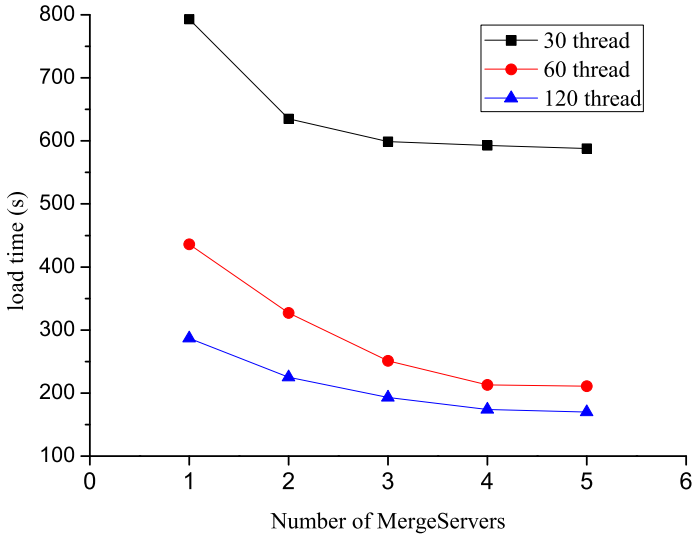

图 4.7: SQL INSERT不同并发度下加载时间

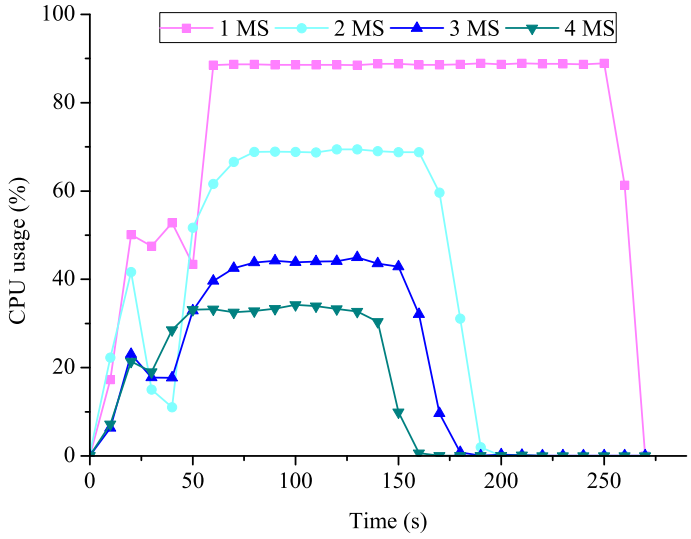

图 4.8: 加载时MergeServer的CPU负载

率，并发线程数为120。从图中可以看出，当MergeServer数量从1增加到4时，MergeServer的CPU使用率从90%降到30%。同时，随着MergeServer数量的增加，数据加载时间逐渐减少，但MergeServer增加到一定数据量（MS = 4，5）时，数据加载时间也趋近极限，如图4.7所示。其原因在于，MergeServer数量增加，UpdateServer接收并发事务量也增加，同时UpdateServer的网络负载也增加。图4.9给出了不同数量的MergeServer下并发加载时UpdateServer的网络负载，并发线程数为120。从图中可以发现，MergeServer数量从1增加到4，UpdateServer的网络I/O从40MB/s提升到100MB/s；而MergeServer数量为5时，其加载时间基本上没有变化，原因是UpdateServer的网络I/O已经趋近饱和。此时，可以通过提高并发度来提高数据加载效率，但120个并发线程下UpdateServer的网络I/O已经达到100MB/s，如果继续增加并发线程数，则会影响UpdateServer的正常服务。

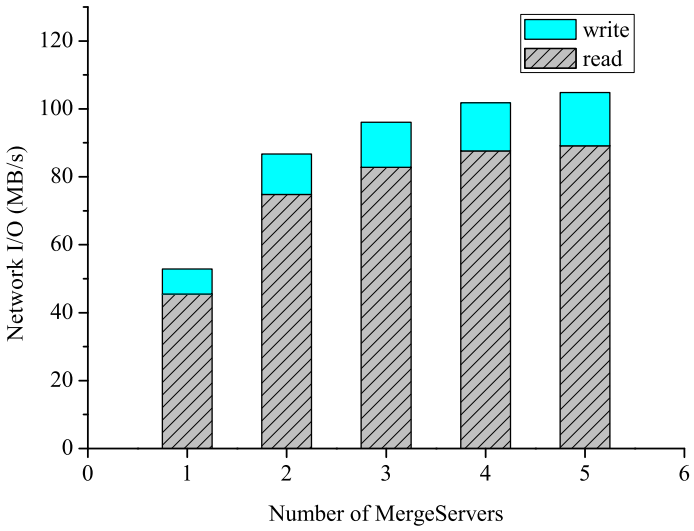

图 4.9: 加载时UpdateServer网络负载

4.4.2.2 DUMT加载性能测试

在本小节中，我们评估了直接更新内存表（DUMT）加载方法的数据加载性能，同时给出DUMT优化前后的性能比较。在第4.3节中，我们介绍了影响DUMT数据加载性能的两个因素：多线程并发度和单次导入的记录数。在这个实验中，我们将基于这两个影响因素来测试DUMT的加载性能，分析实验结果，并给出最优的并发度以及批量插入记录数。

实验中，并发度对应DUMT的并发线程数（thread number），批量插入数据量对应每次从数据文件读取并插入的数据大小（buffer size）。我们分别让buffer size = 128KB，256KB，512KB，然后将线程数（thread number）从8到32依次变化。如图4.10所示，给出了不同线程数和不同buffer size下DUMT加载4.3GB测试数据所需的时间。

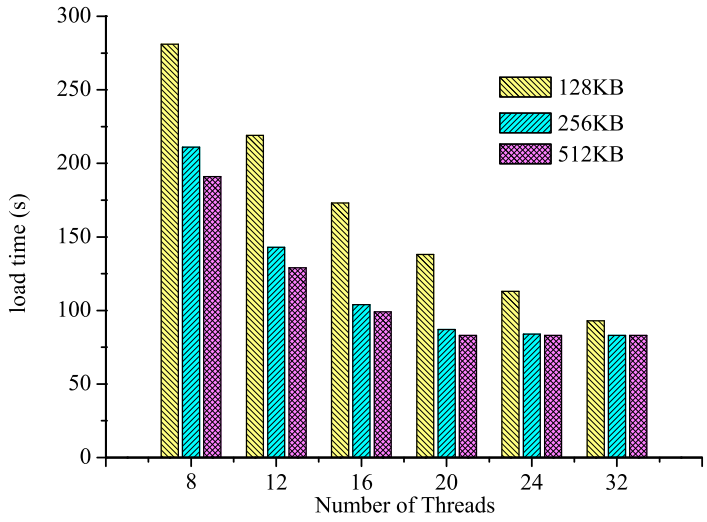

图 4.10: DUMT不同并发度下加载时间

从图中可以发现，随着并发线程数的增加，数据加载效率递增，但并发线程数较高时，线性增加并发线程数，数据加载效率提升程度反而降低，譬如线程数从12增加到24所带来的性能提升低于线程数从8增加到16。其原因在于，随着并发线程数的增加，UpdateServer中并发事务发生冲突的概率也随之增加，导致UpdateServer上处理数据插入事务的时间增加。

在2.1节中介绍了采用数据批量加载可以减少事务处理量，进而减少事务处理的开销。在DUMT批量数据加载中，按照buffer size从数据文件中读取记录，并将整个buffer记录作为一个事务提交到UpdateServer插入内存表MemTable。在4.4.1节中，我们介绍了测试表gems_pdtdcrd结构，并根据实际数据量和记录数计算出平均每条记录占650B。那么，buffer size分别为128KB、256KB和512KB 时，代表批量插入记录数分别为200、400和800。从测试结果可以看出，buffer size 越大，数据加载效率越高。然而，当数据加载并发度较高时，批量加载记录数越大，并行插入事务间发生冲突的概率越大。从图中可以看出，当线程数达到20时，buffer size为256KB和512KB的加载时间基本相同。同时，由于OceanBase设计网络框架

时将网络传输数据包的大小限制为2MB，而每条记录在内存构造B+树数据节点时会有内存膨胀，因此为了保证数据地顺利加载，通常建议将buffer size设置为256KB。

无论buffer size设置为256KB或512KB，当并发线程达到一定数量时，数据加载时间也趋近平衡。其原因是高并发的数据加载已经达到加载服务器的最大网络传输率（120MB/s），而数据加载的瓶颈已经不在UpdateServer事务处理开销。此时，在保证数据加载效率最大化时，也需要考虑降低加载并发度，从而降低UpdateServer负载。因此，采用DUMT方法进行数据加载时，我们将并发线程数设置为20，buffer size 设置为256KB。

4.4.2.3 性能对比

上述两小节分别对两种不同的数据加载方法的加载性能进行测试。如图4.11所示，在测试环境中加载4.3GB数据到OceanBase数据库中，分别采用两种数据加载方法的最短加载时间。

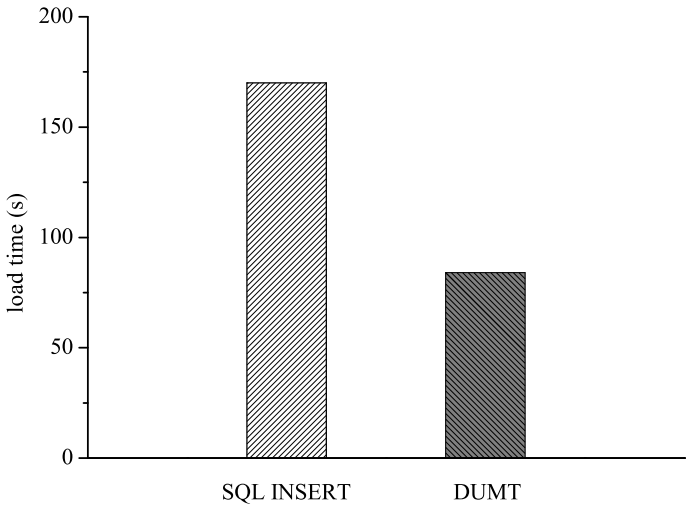

图 4.11: 两种加载方法性能对比

从图中可以看出，DUMT加载性能要明显优于SQL INSERT，且DUMT并不占用OceanBase集群中MergeServer的计算资源。SQL INSERT加载方法将加载负载均衡到多个MergeServer，需要大量的网络开销，而DUMT加载方法直接将数据加载到UpdateServer中，无需更多的网络传输。因此，对于交通银行历史库系统的数据加载选用直接更新内存表（DUMT）加载方法。

4.5 本章小结

本章主要介绍了三种不同的OceanBase数据加载技术和实现方法。首先，本章介绍了OceanBase现有的数据加载工具——ChunkServer旁路导入，通过对其实现方法的研究和分析，得出该方法的优缺点及适用场景。接下来，我们设计和实现一种基于SQL INSERT方式的并发数据加载方法，这种方法满足交通银行历史库系统的数据加载需求且实现原理简单，但其加载性能无法满足需求。然后，通过研究和分析OceanBase架构及实现特点，我们提出一种直接更新内存表（DUMT）数据加载方法。该方法支持并行批量数据加载，并且通过减少网络传输来提升数据加载性能。最后，通过实验对两种加载方法的加载效率进行测试和对比，我们给出了适合历史库系统的数据加载方法。

第五章　多任务并行加载设计与实现

在第三章中我们简单介绍了交通银行历史库系统，并且历史库系统已经采用分布式数据库OceanBase作为数据存储和管理平台，从而解决了业务快速增长和数据量膨胀所带来的海量数据存储和查询问题。同时作为新历史库系统的数据库，OceanBase也将面临着每天海量数据加载的问题。在第四章中，我们针对OceanBase设计和实现了两种加载技术，用以支持数据的增量加载和更新加载，从而满足历史库系统的数据加载需求。然而，历史库系统包含多个子系统，每个子系统中涉及若干张用户表。基于此，我们提出一种多任务并行调度加载的策略，用以满足交通银行历史库系统的海量文本数据的加载需求。

本章首先通过对历史库数据加载过程和数据加载的性能瓶颈进行分析，评估并行加载的可行性，从而提出一种基于多任务并行调度的加载策略。然后，详细介绍多任务并行调度加载策略的具体实现，主要从任务划分和任务调度两个方面进行介绍。最后，对并行调度加载策略进行实验测试，并给出不同任务划分以及不同任务调度策略的加载性能对比。

5.1　多任务并行调度加载设计

5.1.1　可行性分析

在3.1节中我们简单介绍了交通银行历史库系统的构成和其主要功能。历史库系统是由若干个子系统构成，主要包括主机业务应用系统（GEMS）、主机账务系统（GAPS）、企业级用户管理系统（EUIF）、电子网银系统（EBANK）等24个子系统。而历史库系统的数据通常来源于核心业务系统的每天交易记录，因此，核心业务系统中各个子系统需要每天在指定时间将当天的所有交易数据导出，并推送到历史库系统进行历史数据加载。

如3.3节中介绍，在基于DB2数据库构建的历史库系统中，不同子系统分别存储于不同DB2数据库服务器中，而这些DB2数据库服务器是相互独立的。一般情况下，每个子系统的数据推送时间并不确定，同一时刻可能会有多个子系统的数

据需要加载。由于不同子系统之间的数据加载在不同的服务器上执行，因此不同子系统间的数据加载可以同时并发执行。即使采用分布式数据库OceanBase来构建新历史库系统的底层存储，不同子系统间的数据加载仍然可以并行执行。

然而，在新历史库系统中，由于所有子系统都存储在同一个OceanBase集群中，因此，并行执行数据加载时需要更多地考虑数据库的负载。为了保证数据加载程序不占用数据库服务器的计算资源，新历史库系统将数据加载过程运行于数据库之外的其他服务器上，我们称之为加载服务器（Load Server）。对于新历史库系统的每日的数据加载，我们采用直接更新内存表（DUMT）方法将核心系统每日推送的文本数据加载到OceanBase中。但通过4.4 节中测试结果可知，当批量加载的线程数达到一定值时，数据加载达到加载服务器的网络瓶颈（加载服务器通常配置一块千兆网卡，最大网络传输率为120MB/s）。而在分布式数据库OceanBase架构下，更新服务器UpdateServer是OceanBase集群中唯一能接收写入的模块，为了解决UpdateServer单节点瓶颈和提升事务吞吐量，通常UpdateServer所在的服务器需要配置多块千兆网卡或者万兆网卡，譬如交通银行历史库系统的生产环境中UpdateServer就配置万兆网卡。因此，当单台加载服务器达到网络传输瓶颈时，而更新服务器UpdateServer并没有达到网络传输限制，仍可以接收更多的加载数据。

由上述可知，在不影响UpdateServer正常提供服务的前提下，通过增加加载服务器来提高数据加载效率的方法是可行的。因此，在多加载服务器情况下，我们提出一种多任务并行调度加载策略。该方法通过任务调度充分地利用加载服务器和数据库系统的资源，使数据加载任务尽可能地并行执行，从而从整体上提高数据加载效率。

5.1.2 多任务并行调度加载设计

图5.1给出了多任务并行调度加载的整体架构。其中：

- 加载服务器（Load Server）负责执行数据加载，其中最主要的过程是采用直接更新内存表（DUMT）方法将文本数据加载到更新服务器UpdateServer中。除此之外，加载服务器可能还会执行一些其他任务，譬如数据预处理、索引维护、加工表更新等。

- 调度服务器（Schedule Server）主要负责管理和调度所有加载任务。调度服务器根据不同的调度策略，将不同子系统的加载任务调度到不同加载服务器上，使加载任务能够并行执行，进而最小化数据加载时间。

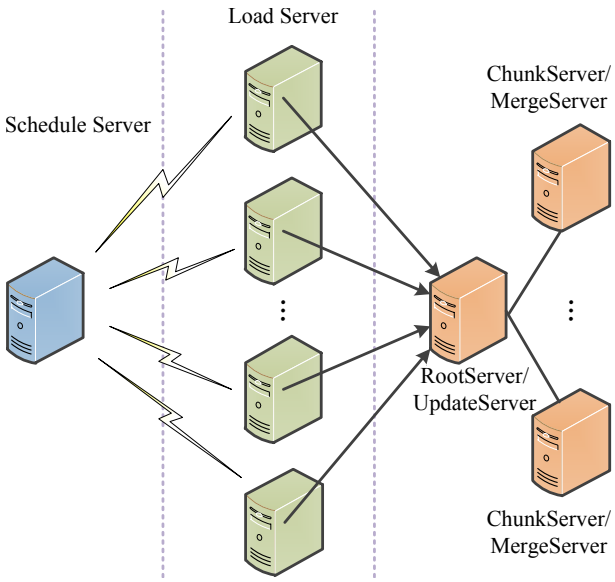

图 5.1: 多任务并行调度加载架构示意图

那么，实现多任务并行调度就需要考虑两点：任务划分和调度策略。任务划分是指将大任务（或任务集合）按照某种划分粒度划分成多个独立的子任务，进而对这些子任务进行并行调度，譬如历史库系统的数据加载任务就可以按照不同的子系统进行划分。而任务调度则是对划分后的子任务集合进行调度，按照某种调度策略将子任务分配到不同加载服务器执行，充分利用加载服务器的计算资源，进而减少数据加载的开销。下面，我们将从这两个方面来介绍交通银行历史库系统多任务并行调度加载的实现。

5.1.3 任务并行度

对于海量数据加载服务来说，整个加载系统全局的资源协调是一个很重要的环节。在多任务的计算资源需求下，通过控制任务的并行度来实现多任务的协调调度，并考虑有限资源在时间和空间上合理搭配，充分利用多个加载服务器的计算资源，以期达到更好的加载效率[62]。

定义5.1.1. 假设有$n(n>1)$个加载服务器，分别记为：$S_1, S_2, ..., S_n$。对于每一个加载服务器$S_i(i=1,...,n)$，假设在服务器S_i上可并发执行的任务个数的最大值为P_i，则整个加载系统的最大任务并行度P为：

$$P = \sum_{i=1}^{n} P_i$$

一个合适的任务并行度不仅可以充分利用加载服务器和数据库系统的计算资源，同时还可以进一步提高整体的海量数据加载效率，因此确定一个任务并行度对整个并行加载系统的加载能力是至关重要的。由于任务并行度与实际运行环境有密切关系，无法通过计算准确地给出精确的合适任务并行度。譬如，假设UpdateServer配置一块万兆网卡，而加载服务器采用千兆网卡，那么理论上10台加载服务器同时加载才会达到UpdateServer的网络传输瓶颈，而实际上少于10台的加载服务器同时工作时就已经达到UpdateServer瓶颈。因此，我们通过大量的实验测试，在确定的实际环境中测出最佳的任务并行度。

5.2 任务模型及任务划分

5.2.1 任务模型

任务是指完成某一种特定功能的应用实例，是构成多任务系统的基本单元，也是多任务并行调度中最小的调度单位。根据任务间是否存在依赖关系可分为：

- **独立任务：**任务间不存在数据或控制依赖关系，任务可以按照任意次序执行。
- **优先约束任务：**任务间存在某种时序或者控制依赖关系，必须按照某种次序执行。

通过对交通银行的数据加载过程分析得知，历史库数据加载任务可以在子系统层面划分为多个独立的加载过程，每个子系统内不同表间的数据加载也是相互独立，而一个表的加载任务也可细化成多个不同的子加载任务。在当前历史库系统的数据加载过程中，一个表的加载任务通常可以划分为以下几类子加载任务：

1. **数据文件预处理**

   预处理负责将来自不同的子系统的数据格式处理成统一格式，主要包括统一列分隔符和行分隔符、去除字段值前后空格、添加新字段值等。

2. **排序**

   由于直接更新内存表（DUMT）方法进行批量加载，因此加载前需要对数据进行排序。这里我们将排序工作从批量加载过程中分离出来并作为一个单独子任务，以便获取更高的任务并行度。

3. **批量加载**

使用批量加载工具将文本数据加载到目标表中，这里的批量加载工具就是采用直接更新内存表（DUMT）技术实现的。

4. **维护索引**

索引维护主要根据每天新增数据更新表的索引。由于目前OceanBase版本不支持二级索引，我们需要自己维护索引表。

5. **更新加工表**

历史库中有一些表的数据并不是来源于源数据库，而是通过对当前数据库中某些表的数据进行加工处理后得到的，这些表被称为加工表。而更新加工表是指当加工表涉及的源表数据得到更新时，那么也需要对加工表进行必要的更新和维护。

其中，维护索引和更新加工表两个子任务并不是存在于所有表的加载过程，例如某张表没有索引和加工表，那么该表的加载过程只包含前三个子任务。实际上，这些不同的子任务间并不是相互独立的，而是存在某种优先约束关系。譬如，前三个子任务间存在数据依赖关系，而后两个子任务又需要第三个任务完成后才能执行。通过对加载过程分析，不同子任务间存在两种优先约束关系:数据依赖和控制依赖。因此，一个表的加载任务由一组具有优先约束的子任务组成。根据第2.2.1节中任务模型介绍，对于由一组具有优先约束的任务构成的任务集合，通常采用任务优先图TPG或有向无环图DAG进行建模。本文采用有向无环图DAG模型对历史库系统的一个表加载任务进行建模。

任务DAG模型可以表示为：$D=(V,E,w,c)$，其中$V=\{v_i|i=1,2,3,...,|V|\}$表示一组优先约束的任务集合，$|V|$表示任务数目;$E=\{e_{ij}|e_{ij}$表示任务$v_i$到$v_j$的有向边}是一组消息的集合，表示了任务间的通信和优先约束关系[63]。

在有向无环图D中，每个节点$v_i \in V$代表执行任务，其权值w_i代表任务执行时间，而$e_{ij}=(v_i,v_j) \in E$则表示任务v_i和v_j之间的依赖关系。不同的依赖关系，传递的消息也不同，其有向边的权值也不相同。假设两个任务间存在数据依赖关系，任务间需要传递数据，此时有向边e_{ij}的权值c_{ij}则表示任务v_i和v_j之间的传递数据所需要的通信时间；反之，若任务间存在控制依赖，那么有向边e_{ij}没有权值。

当任务v_i和v_j之间存在数据依赖关系，若两个任务在不同服务器上运行，c_{ij}则表示任务v_i产生的输出数据在两台服务器间传输所需的网络开销；若两个任务被分配到同一台服务器上运行，那么$c_{ij}=0$ 表示两个任务间的数据不需要经过网络传输（同一台服务器间的数据传输时间忽略不计）。

因此，每个表的数据加载所需时间为：

$$T = \sum_{i=1}^{|V|} w_i + \sum_{j=1}^{|E|} c_j$$

例5.2.1. 假设*test*表有一个索引表*test_index*和一个加工表*test_etl*，如上所述，*test*表的加工过程共包含*5*个子加载任务$\{T_1, T_2, T_3, T_4, T_5\}$，其加载任务模型如图*5.2*所示。

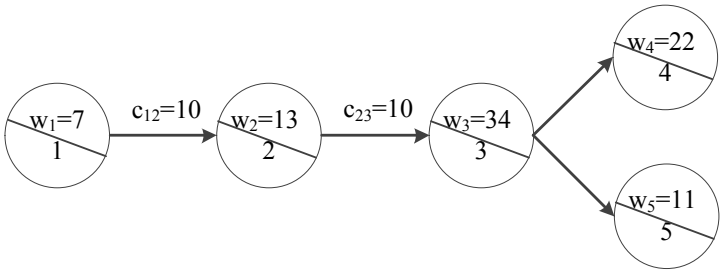

图 5.2: test表的DAG任务模型

其中，T_1、T_2和T_3分别对应数据预处理、排序和批量加载三个子任务，对应的处理时间分别为：*7,13,34*；T_4是对索引表*test_index*的维护，T_5负责更新加工表*test_etl*，对应处理时间分别为*22* 和*11*。而c_{12}和c_{23}表示T_1、T_2和T_3均不在同一节点上运行时，数据文件通过网络传输的开销。那么，*test*表的最大加载时间$T_{max} = 7 + 10 + 13 + 10 + 34 + 22 + 11 = 107$

如果将T_1、T_2和T_3运行与同一节点，且T_4和T_5并发执行，*test*表的最小加载时间$T_{min} = 7 + 13 + 34 + 22 = 76$

5.2.2 划分粒度

由于历史库系统的数据加载可分为多个不同子系统的数据加载过程，不同子系统间数据加载互不影响，同时同一系统中不同表之间的数据加载又是相互独立的，因此整个历史库的加载任务是由多个独立的加载任务组成。根据上一小节中介绍，每个表的加载任务可以构建成一个有向无环图DAG，那么整个历史库系统的加载任务模型是由多个独立的有向无环图构成，如图5.3所示。

因此，对历史库系统的加载任务的划分问题转换成基于有向无环图DAG的粒度划分。通常来说，无论是任务的粒度还是整个DAG的粒度，它都是用来表示任务的计算代价和不同任务间的通信延迟之间的关系。但是对于如何定义这种粒度，很多文献中分别给出了不同的表述[64, 65, 66, 67]。但是上述这些文献中任务的

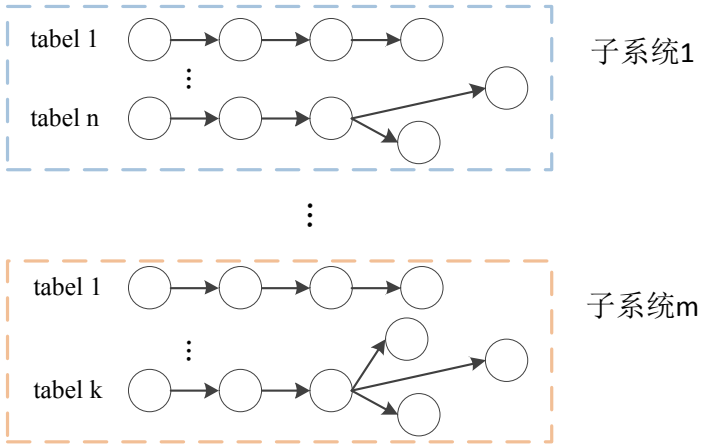

图 5.3: 历史库系统的数据加载任务模型

粒度定义都只是考虑了DAG中单个任务的计算代价，并不适用于有向边带有权值的DAG的调度。因此，一些学者针对带边权值的有向无环图DAG重新定义任务粒度,其中文献[68]考虑了任务计算代价和任务间通信代价来定义任务的粒度，并通过这种新的任务粒度来决定DAG中任务的串行化和并行化的均衡问题。

鉴于此，历史库系统可以按照子系统级、表级、甚至更小的加载子任务粒度来切分数据加载任务。如果采用子系统级的划分粒度，这种切分方法与原历史库系统的划分策略类似，即不同子系统的数据加载任务在不同的加载服务器上执行。由于不同的子系统的数据量不同，这种划分粒度无法充分利用加载服务器的计算资源。因此，本文根据加载任务特点提出两种细粒度的划分策略：表级划分和子任务划分，并分别对两种划分粒度设计和实现不同的任务调度算法，进而分析和比较两种划分粒度及其对应的调度算法。

5.3 多任务并行调度实现

任务调度是整个并行加载中一个非常重要的环节，它对于充分利用各个加载服务器的计算资源，从整体上提高加载效率等有至关重要的意义。

在2.2.2节中，我们对相关任务调度策略进行简单分类和介绍。对于静态调度，所有任务的运行时间、通信关系都是已知，且每个任务在执行前已经被分配到固定的处理器。而动态调度可以根据任务执行过程中的负载变化，对任务进行迁移或重新分配。虽然动态调度技术可以获得较好的负载均衡的效果，但其调度开销很大，且实现也很复杂。所以，当静态调度可以获得较好的效果时，应当尽量选用静态调度技术。本节也采用静态调度技术来设计和实现并行加载任务调度。

5.3.1 任务管理

对于任务的管理，本文借鉴文献[62]中提出的任务状态定义，给出了前去任务和后继任务的定义，同时提出一种适合本文的任务管理策略。

在有向无环图DAG模型中，当某个任务的前驱任务都已执行成功，并且其资源需求也得到满足，则称该任务是可调度执行的。

定义5.3.1. 假设任务T_A依赖于任务集合$T_B = \{T_{B_i} | i = 1, 2, ..., n\}$。当且仅当$T_B$中所有任务全都执行成功时，则称$T_A$的状态是可执行的（*Executable*）；反之，T_A的状态为等待（*Waiting*）。因此，我们称任务集合T_B中每个任务T_{B_i}是T_A的前驱任务，或者T_A是任务集合T_B 中每个任务T_{B_i}的后继任务。

因此管理加载任务时，为了保证任务的优先级，我们选择最常用的优先级队列（Priority Queue）方式来存储和管理任务。同时，为了控制任务之间执行顺序，根据任务的执行状态将任务分为两类：可执行任务和等待任务，并分别存储于可执行任务队列和等待任务队列。每个加载服务器都采用这两种任务队列管理加载任务，并根据任务的状态（可执行状态和等待状态）将加载任务在两个队列间进行调度。其任务管理调度的基本原则：当加载服务器接收到一个或一批新任务时，该任务将被添加到等待任务队列中，只有当该任务的状态为可执行时，才允许将该任务从等待任务队列移到可执行任务队列中。算法2给出了任务队列管理调度算法的伪代码。

5.3.2 任务调度

通过第2.2.2节中的介绍，结合对历史库系统的数据加载任务特点的分析，可以看出历史库系统的加载任务调度属于非抢占式调度和非迁移式调度。因此，我们将采用集中式分配、多机调度执行的方式来对多加载任务并行调度。集中式分配是指整个调度系统中有一个服务器作为专门的任务分配调度器，所有的加载任务都被提交到这个中心分配服务器上，经过其分析和计算，然后将任务划分后分配到各个加载服务器上；而多机调度是指不同的加载服务器采用上一小节中介绍的方法对其所分配的任务进行管理，并根据算法2对不同任务队列的任务进行管理和调度。这种调度方式实现起来相对比较简单，并且避免了过多的通信开销，同时将任务的调度分布到多个节点上，具有良好的可扩展性。

对于调度技术的选择，我们采用基于表调度和任务聚簇调度二者结合的调度方法。在集中式调度服务器上，采用任务聚簇的调度方法，将任务按照划分粒度进行聚簇，然后将各个任务簇分发到不同的加载服务器上执行；而加载服务器则

Algorithm 2 任务队列管理调度算法

输入：可执行任务队列：Q_e；等待任务队列：Q_w；

while Q_e不为空或Q_w不为空 **do**
  if Q_e不为空 **then**
    根据任务优先级，从Q_e中取出可执行任务T_A；
    execute(T_A)，任务T_A执行结束且结果正确；
    for all $T_i \in Q_e$ **do**
      if T_i是T_A相关任务 **then**
        $T_i \rightarrow B$，其中B是任务T_A的相关任务集合；
      end if
    end for
    for all $B_i \in B$ **do**
      重新计算B_i的状态；
      if B_i的状态为可执行的 **then**
        将B_i从Q_w中移到Q_e中；
      end if
    end for
  end if
end while

采用表调度方法，将其分到的任务簇中的任务按照优先级排序存放于队列中，然后由加载服务器根据任务状态和资源使用情况来进行任务选择调度。

接下来，我们将分别介绍两种不同的划分粒度下任务调度的实现。

5.3.2.1 基于表级任务调度策略

由于不同表间的数据加载过程互相独立，我们可以将一张表的加载任务分配到任意加载服务器执行，而不影响其他加载任务的执行。因此，基于表级任务调度是指在加载任务按照表级粒度划分后，将不同表的加载任务调度到合适的加载服务器上执行。相比于原历史库系统中子系统间的并行加载，该方法充分利用了加载服务器的计算资源，使得加载任务能够并行化执行，进而提高数据加载效率。该调度过程的主要步骤包括：第一步将数据加载任务按照表级粒度划分，得到由不同表的加载任务组成的任务集合；第二步对任务集合中加载任务按照优先级排序；第三步，采用任务聚簇调度算法将任务集聚类成多个任务簇，任务簇的数量取决于加载服务器数量；最后，将不同任务簇分别分配到不同加载服务器，而各

个加载服务器则自主地调度任务簇里的加载任务。

在这里，假设不同子系统的加载任务已经按照表级粒度划分，算法3给出了上述基于任务聚簇调度算法的伪代码。实现该调度算法的关键技术在于优先级衡量策略和任务聚类约束。如2.2.3节中介绍，优先级衡量策略有很多种，这里我们采用加载数据量来衡量不同表的加载任务优先级，即加载数据量越大的表，其加载任务的优先级越高。而任务聚类约束也同样采用加载数据量来衡量，即任务聚类时需要尽可能地保证各个任务簇的加载数据量相等，充分利用加载服务器的计算资源的同时也可保证负载均衡。

Algorithm 3 基于任务聚簇的调度算法

输入： 加载任务集合：$T = \{T_i | i = 1, 2, ...n$，其中，$T_i$为一个表的加载任务$\}$；
加载服务器数量（任务聚簇数）：$N(N > 1)$；

输出： N个优先级队列：$PQ = \{PQ_i | i = 1, ..., N\}$，代表$N$个任务聚簇；

1: $sort(T)$，对任务集合T中每一个任务T_i按照优先级排序；
2: 创建N个优先级队列$PQ = \{PQ_i | i = 1, ..., N\}$；
3: **for all** $T_i \in T$ **do**
4: $PQ_{min} = minPriority(PQ)$，从$N$个优先级任务队列中找出优先级最小的队列；
5: $PQ_{min} \leftarrow T_i$，将T_i添加到任务队列PQ_{min}中；
6: **end for**
7: **return** PQ

对于任务簇中的任务调度，加载服务器主要采用表调度算法，将每张表的整个加载过程作为最小的调度单元，并按照优先级存储于等待队列中，具体调度过程见算法2。由于每张表的数据加载过程至少包含三个步骤，其中批量加载过程采用直接更新内存表（DUMT）加载方法。根据4.4节实验结果可知，DUMT数据加载会占用加载服务器的所有网络带宽，为了避免加载任务间发生资源使用冲突，通常加载服务器上的加载任务并行度只能为1。

在5.2.1节中，我们将一张表的数据加载过程构建成一个DAG模型，其加载所需时间为：

$$T = \sum_{i=1}^{|V|} w_i + \sum_{j=1}^{|E|} c_j$$

然而，采用基于表级任务调度策略将一张表的所有加载过程分配到同一个节点上执行，避免存在数据依赖的子任务间的数据传输。此时，每张表的数据加载所需

时间为：

$$T = \sum_{i=1}^{|V|} w_i$$

因此，采用基于表级任务调度策略可以降低每张表的数据加载时间，从而可以提升整体的数据加载效率。

5.3.2.2 两阶段任务调度策略

在上一小节介绍的基于表级任务调度策略中，每个加载任务就是一张表的整个加载过程，包括数据预处理、排序、批量加载以及维护索引和更新加工表等。通过对每个加载子任务的特点分析，可以看出数据预处理和排序两个阶段主要使用CPU资源；批量加载过程则占用加载服务器的所有网络带宽（通常，我们尽可能地提高批量加载的线程数以最大化利用网络传输速率）；而维护索引和更新加工表则需要占用一部分CPU计算和网络传输资源。如图5.4所示给出了不同加载子任务所需的主要计算资源。

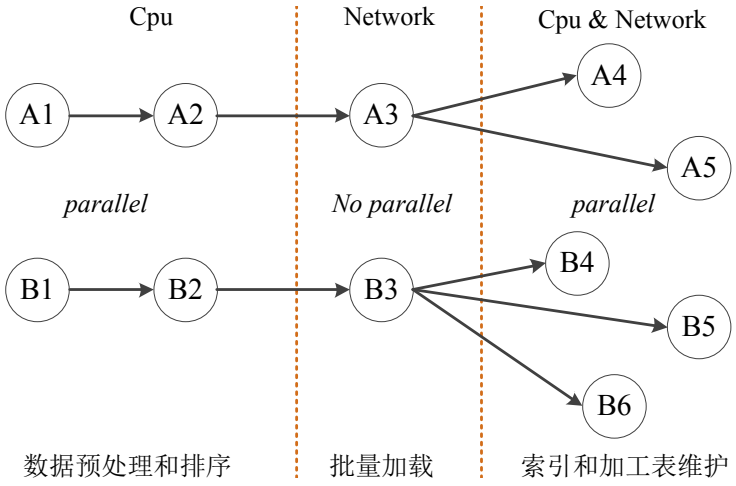

图 5.4: 不同加载子任务所需计算资源

由于不同加载子任务所需的计算资源不同，不同加载子任务间可以并行执行，譬如A表执行批量加载过程A_3时，可以并发执行B 表的数据预处理B_1或排序B_2加载子任务。因此，基于表级任务调度策略并不能充分利用加载服务器的计算资源，我们提出一种细粒度的任务划分策略，将一张表的加载任务按照不同处理过程所需资源类型划分成三类子任务：数据预处理和排序、批量加载、索引和加工表维护。然后通过资源合理搭配和分时使用的原则，使得不同表的加载子任务并行执行，进而充分利用加载服务器的计算资源。同时，考虑子任务间的数据依赖

Algorithm 4 基于任务聚簇的调度算法

输入： 加载任务集合：$T = \{T_i | i = 1, 2, ...n,$其中，$T_i$为一个表的加载任务$\}$；
加载服务器数量（任务聚簇数）：$N(N > 1)$；

输出： N个$PHASE_1$任务簇：$PQ1 = \{PQ1_i | i = 1, ..., N\}$；
N个$PHASE_2$任务簇：$PQ2 = \{PQ2_i | i = 1, ..., N\}$；

创建两个任务队列$PHASE_1Q$和$PHASE_2Q$，分别存储不同阶段的子任务；
for all $T_i \in T$ **do**
    $split(T_i) \Rightarrow (Phase_1T_i, Phase_2T_i)$，将任务$T_i$划分成两个子任务；
    $(PHASE_1Q, PHASE_2Q) \leftarrow (Phase_1T_i, Phase_2T_i)$
end for
$sort(PHASE_1Q), sort(PHASE_2Q)$，分别对不同任务队列进行优先级排序；
创建两个任务簇集合$PQ1$和$PQ2$，其中每个任务簇对应一个优先级队列；
for all $T_j \in PHASE_1Q$ **do**
    $PQ1_{min} = minPriority(PQ1)$，从$PQ1$中选出优先级最小的任务簇；
    $PQ1_{min} \leftarrow T_j$，将T_j添加到任务簇$PQ1_{min}$中；
end for
for all $T_k \in PHASE_2Q$ **do**
    $PQ2_{min} = minPriority(PQ2)$，从$PQ2$中选出优先级最小的任务簇；
    $PQ2_{min} \leftarrow T_k$，将T_k添加到任务簇$PQ2_{min}$中；
end for
return $PQ1, PQ2$

和控制依赖，我们将数据预处理、排序和批量加载划分为第一阶段的子任务，而索引维护和加工表更新作为第二阶段的子任务。由于批量加载过程（如图5.4中的A_3和B_3）需要占用加载服务器的所有网络带宽，无论A 表还是B表的索引维护和加工表更新的操作都无法与之并行执行。因此，我们提出一种两阶段调度（Two-phase Schedule）策略，即将所有表的加载任务切分成细粒度的加载子任务，并分成两个阶段分别调度。

实现两阶段任务调度策略，主要包括两个部分：两阶段任务划分和两阶段任务调度。两阶段任务划分主要基于不同子任务间的依赖关系以及并发性，将所有表的加载任务划分为两个阶段，并采用任务聚簇的方法对不同阶段的任务集进行聚类，算法4中给出了两阶段任务聚簇算法的伪代码。

而两阶段任务调度则是将不同阶段的任务簇分成两个阶段进行调度，而不同阶段的任务簇则由加载服务器分别进行不同的调度。其主要过程如下：

1. **PHASE 1：** 将$PQ1$中N个第一阶段的任务簇分别分配到N个加载服务器上执行。

2. **PHASE 2：** 在PHASE 1中所有任务完成且完全正确后，再将$PQ2$中N个第二阶段的任务簇分别分配到N个加载服务器上执行。

对于两个阶段任务簇的多机调度阶段，加载服务器也采用基于表调度算法来存储和管理不同阶段的任务簇，并根据优先级（加载数据量）来决定加载任务执行顺序。不同于表级的任务调度过程，我们将第一阶段的加载任务按照所需计算资源分成两类：处理器密集型（CPU-intensive）和网络密集型（Network-intensive），其中数据预处理和排序属于处理器密集型，而批量数据加载属于网络密集型。由上面介绍这两类任务可以并行执行，我们将其分别存储在不同的准任务队列中，在任务调度时最大化不同类型加载子任务间的并行度，充分利用加载服务器的计算资源，进而减少加载时间。

与基于表级任务调度类似，两阶段任务调度策略也避免存在数据依赖的子任务间的数据传输，同时充分地利用计算资源将加载子任务并行化，譬如不同表间的预处理和排序并行化，索引维护和加工表更新的并行化等。根据5.2.1小节中数据加载时间的计算公式，采用细粒度的两阶段任务调度后，每张表的数据加载所需的平均时间为：

$$T = w_3 + \max_{4<i<|V|}\{w_i\}$$

其中，w_3表示批量加载的时间，$\max_{4<i<|V|}\{w_i\}$表示索引维护和加工表更新子任务中最长的处理时间。

由此可见，两阶段任务调度的加载效率要优于基于表级任务调度。接下来，我们将通过实际环境的测试，对比两种不同调度策略的性能。

5.4 实验准备与结果

5.4.1 实验准备

5.4.1.1 数据集

本实验中，实验数据采用交通银行历史库系统的历史数据。如第三章中介绍，历史库系统主要负责各类业务的历史交易查询，其中包括主机账务系统（GAPS）、主机应用系统（GEMS）、远程银行（EPIC）等24个子系统，共计300多张数据表，同时还包括100多个索引表和7张加工表。每天核心业务系统将各个子系统的数据推送到历史库系统加载，由于各个系统的负责业务不同，涉及的表数量以及数据

加载量也不同。实验数据选用历史库系统的某一天历史交易数据，数据量大概有100GB（这里的数据量并不包括索引和加工表的更新量），不同子系统的数据加载量的比重如表5.1所示。

表 5.1: 不同子系统的数据加载量占比

数据类型	说明	表数量	数据量（GB）	所占比例（%）
GEMS	主机应用系统	62	30	30
DJK	贷记卡系统	30	20	20
GAPS	主机账务系统	7	7.5	7.5
EDWH	新一代数据仓库系统	25	6.5	6.5
BPS	综合积分系统	10	6	6
others	其余20个系统	168	30	30

5.4.1.2 实验设置

本节所提出的多任务并行加载策略最终在新历史库系统的生产环境中应用，而新历史库系统部署在分布式数据库OceanBase（版本号：0.4.2.21）集群上。生产环境中数据加载采用直接更新内存表（DUMT）方法进行批量加载，我们在多台加载服务器上同时并发执行DUMT 批量加载过程，通过大量的数据测试，给出生产环境中最合适的加载服务器数量。然后，分别测试不同调度策略下的并行加载性能，并评估不同的并行调度策略的优劣。

为了进行本次实验，我们测试案例部署于交通银行历史库系统的生产环境中进行测试，其硬件参数见表5.2。历史库系统的生产环境中部署了一主一备两组OceanBase集群，其中主集群中包括2台更新服务器UpdateServer和20 台基线数据服务器ChunkServer，这里我们只考虑更新服务器UpdateServer配置，见表5.2(a)。并且为了测试多机并发加载的性能，我们还准备了若干台数据加载服务器LoadServer，其硬件配置见表5.2(b)

5.4.2 实验结果与分析

5.4.2.1 并行度测试

在5.1.3小节中，我们给出任务并行度的定义，同时也分析了合适的并行度可以进一步提高数据加载能力。由于，任务并行度与实际运行环境密切相关，所以无法通过计算得到合适的并行度。因此，我们需要在历史库系统的实际生产环境下，通过测试来确定加载系统的并行度，进而观察不同并行度对整个加载过程以

表 5.2: 实验环境说明

(a) UpdateServer配置

组件	说明	备注
CPU	Intel(R) Xeon(R) CPU E5-2680v2*2 40核	20核/CPU
主频	2.60GHz	
内存	500GB	主机内存
以太网	Intel 82599EB 10-Gigabit Ethernet	万兆网卡

(b) LoadServer配置

组件	说明	备注
CPU	Intel(R) Xeon(R) CPU E7-2850*4 16核	4核/CPU
主频	2.00GHz	
内存	50GB	主机内存
以太网	VMware VMXNET3 Ethernet	千兆网卡

及数据库的影响。本实验主要通过设置不同数量的加载服务器，来观察加载性能的变化以及OceanBase的负载变化。

试验中数据加载采用直接更新内存表（DUMT）方法，并将线程数设置为20，buffer size设置为256KB，图5.5给出了不同数量的加载服务器下DUMT数据加载性能。从图中可以看出，随着加载服务器的数量增加，数据加载效率不断提高，但加载速率提升幅度却逐渐降低。当加载服务器数量增加到5时，加载速率达到峰值，加载速率接近130MB/s。通过对数据库OceanBase的负载（主要是UpdateServer的CPU和网络负载）监控和分析，我们发现并行度小于4时，数据库资源并没有得到充分利用，此时增加加载服务器数量可以获得加载速率明显的提升。当并行度增加到4或者5时，尽管加载速率仍有提升，但提升幅度却变小，原因是UpdateServer的CPU使用率已经达到80% ∼ 90%，数据库负载变高导致处理速度降低。继续增加并行度到6时，加载速率基本不变，此时UpdateServer的CPU使用率接近100%。

在实验中，我们还针对不同的加载场景下的加载效率进行测试，这两种场景分别为：无索引表和加工表的数据加载、有索引表和加工表的数据加载。从图中可以看出，前者的加载效率优于后者，并且二者均在并行度为5时数据加载效率达到峰值，分别为125MB/s 和98MB/s。通过实验结果的对比和加载特点的分析，可以看出索引维护和加工表更新过程对数据加载效率有很大影响，因此上一节中提到的两阶段调度策略可以充分地使这两类子加载任务并行化执行，进而提

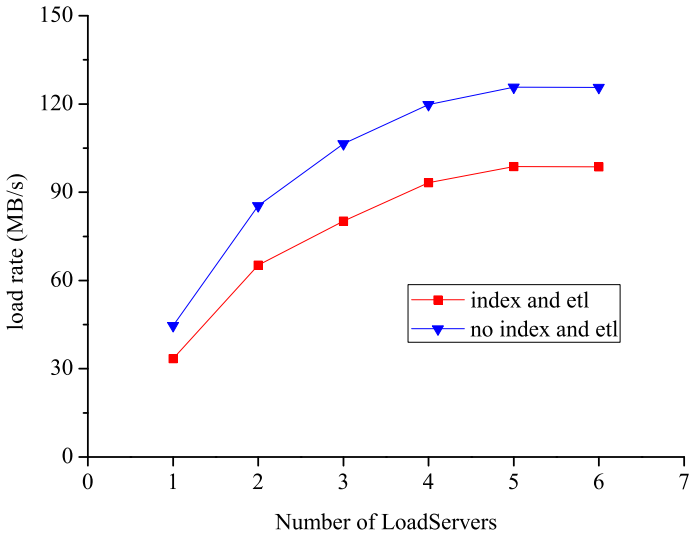

图 5.5: 不同并行度下加载效率

高整体的加载效率。同时，最佳的并行度选择只取决于实际运行环境，而与加载场景无关。因此，采用直接更新内存表（DUMT）数据加载方法时，我们可以设置4或5台加载服务器并行加载，如果数据加载时应用访问量较少的话，建议将并行度设置为5，此时加载速率最高可以达到130MB/s。

5.4.2.2 不同调度策略性能测试

在5.3节中，我们针对历史库系统的数据加载任务提出了两种不同的任务调度策略：基于表级任务调度（table schedule）策略和两阶段任务调度（Two-phase schedule）策略。前者将一张表的整个数据加载过程作为单独任务进行调度，而后者则根据数据加载的特点将加载任务切分成多个子任务，并分成两个阶段进行调度。因此，本实验主要测试两种不同的调度策略下并行数据加载效率。由于两阶段任务调度策略会根据不同阶段的子任务特点进行不同的优化调度，为了测试两阶段调度策略在不同阶段的调度效果，我们分别在两种不同的加载场景下测试两种调度策略的加载效率，并通过实验结果的对比和分析，得出不同调度策略的优劣及适用场景。

图5.6给出了两种调度策略在不同的数据加载场景下的加载时间。对于两阶段任务调度策略来说，无索引和加工表的加载过程只包括数据预处理、排序和批量加载，相当于第一阶段的加载任务；而有索引和加工表的数据加载过程不仅包括第一阶段的加载任务，而且还有索引维护和加工表更新过程，相当于包含两个阶

段的加载任务。从图中可以看出，在两种不同的加载场景下，两阶段任务调度策略的加载效率都要高于基于表级任务调度策略。

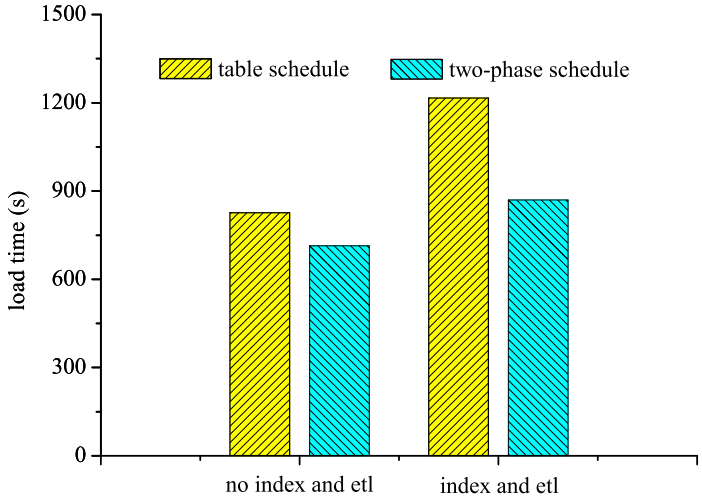

图 5.6: 两种调度策略性能对比

在无索引和加工表的加载场景下，两阶段任务调度只需进行第一阶段任务调度，其加载时间比基于表级任务调度少了100s。其原因在于，两阶段任务调度过程中将第一阶段的加载任务切分成两种类型的加载子任务，并通过资源合理搭配和分时使用的原则，尽可能地使不同加载子任务并行执行，从而减少整体的数据加载时间。而对于有索引表和加工表的加载场景，两阶段任务调度的加载时间比基于表级任务调度策略少了350s。尽管两种调度策略都增加了索引维护和加工表更新过程，但是采用两阶段调度策略所增加的加载开销要远小于基于表级任务调度策略。其原因是，基于表级任务调度将整个表的加载过程作为并发调度单元，并未考虑索引维护和加工表更新加载任务的并行化执行；而两阶段任务调度则将这两类任务划分到第二阶段调度，将所有的加载任务分配到多个加载服务器上并行执行。相比之下，两阶段任务调度策略可以充分地利用加载服务器的资源，使索引维护和加工表更新过程最大并行化执行，从而提高整体的数据加载效率。根据上一小节的实验结果得知，索引维护和加工表更新过程对整体的加载性能有很大的影响。因此，对于有索引和加工表的数据加载场景，采用两阶段任务调度策略可以减少数据加载时间。

5.5 本章小结

本章主要设计和实现一种多任务并行调度的数据加载方法，以满足历史库系统的海量数据加载需求。首先，本章分析和评估并行执行数据加载到OceanBase数据库的可行性，并设计一种多任务并发调度的数据加载方法，以满足历史库系统的数据加载需求。接下来，分别从任务模型和任务调度两个方面来介绍多任务并行调度加载的实现。根据历史库系统的数据加载特点，本章采用有向无环图DAG对加载任务进行建模，因此，数据加载任务调度问题转换成基于DAG任务调度问题。然后，通过研究加载任务的特点，我们分别提出了基于表级任务调度策略和两阶段任务调度策略，这两种策略分别基于不同的任务划分粒度，充分考虑资源利用和任务并行化，提高整体的数据加载效率。最后，我们通过实验对比了两种不同调度策略的加载效率。实验结果显示，两阶段任务调度策略的加载性能要优于基于表级任务调度策略，并且当加载任务中包含大量索引维护和加工表更新操作时，两阶段调度策略会获得更好的效率提升。

第六章　总 结

随着互联网的快速发展，金融行业的业务系统不断地增强和更新，企业数据库的数据量也成倍地增长，海量数据的存储和管理给金融行业的各个系统的数据存储层带来了严峻的挑战。基于此，一些金融企业，譬如银行，开始尝试采用分布式数据库代替传统商数据库（DB2、Oracle）来提供底层的数据存储和管理，利用其良好的可扩展性和容错性来解决海量数据存储的问题。同时，企业内部不同的系统之间需要大量的信息共享和迁移，这就需要在不同数据库间进行数据加载，因此如何将海量数据快速加载到分布式数据库系统中也成为一个新的挑战。本文主要以交通银行历史库系统作为研究对象，分析和研究历史库系统的数据存储和加载特点，设计和实现一种海量数据并行加载方法用以解决历史库系统的数据加载问题。本文的主要研究内容包括以下几个部分：

首先，本文简单介绍交通银行历史库系统，研究和分析了历史库系统的数据存储层面临的海量数据存储和加载的问题。为此，新历史库系统采用分布式数据库OceanBase实现数据存储和管理，从而解决海量数据存储和查询的问题。但同时，如何把海量数据的快速加载到OceanBase中，将成为新历史库系统面临的最大挑战。基于此，我们提出了两种解决思路：实现批量加载工具和并行加载策略。

其次，介绍了OceanBase数据库现有的数据加载工具的特点和实现过程，并对其优缺点进行分析，得出现有加载工具并不适用于历史库系统的数据加载。而本文则根据历史库系统的数据加载需求和分布式数据库OceanBase的特点，设计和实现了两种不同的数据加载方法，同时支持将文本数据以增量加载或更新加载方式导入数据库中。同时，在相同的测试环境下对两种加载方法进行测试和分析，选取一种适用于历史库系统的数据加载方法。

最后，本文为历史库系统的数据加载过程设计一个多任务并行加载的数据加载方案。在设计过程中，分析和研究了数据加载任务的特点，给出并行加载的可行性。同时，针对数据加载过程，采用有向无环图DAG对加载任务进行建模，并对DAG任务模型提出两种并行调度策略。通过测试结果分析两种调度策略的优劣势，并给出一个合适的加载调度策略。

综上所述，本文以交通银行历史库系统为研究对象，通过研究历史库系统的数据特点分析海量数据在数据存储和数据加载中的主要挑战，并重点研究了历史库系统中的海量数据加载问题。针对分布式数据库OceanBase，设计和实现了批量数据加载方法，并提出一种多任务并行调度加载策略来实现海量数据的快速加载。

参考文献

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

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

[3] Tao Code. OceanBase. `http://code.taobao.org/p/OceanBase/`. [Online; accessed 10-April-2015].

[4] Sina Finance. 首个去IOE银行系统. `http://finance.sina.cn/2015-04-02/detail-icczmvun8094792.d.html?spm=0.0.0.0.ajcey0&wm=3175_0001&lwfrom=UID10983917_dynamic_page&_network=wifi&_network=wifi`. [Online; accessed 10-April-2015].

[5] Sihem Amer-Yahia, Sophie Cluet, and Claude Delobel. Bulk-loading techniques for object databases and an application to relational data. In VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA, pages 534–545, 1998.

[6] Sihem Amer-Yahia and Sophie Cluet. A declarative approach to optimize bulk loading into databases. ACM Trans. Database Syst., 29(2):233–281, 2004.

[7] Jochen Van den Bercken and Bernhard Seeger. An evaluation of generic bulk loading techniques. In VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy, pages 461–470, 2001.

[8] Robert Fenk, Akihiko Kawakami, Volker Markl, Rudolf Bayer, and Shunji Osaki. Bulk loading a data warehouse built upon a ub-tree. In 2000 International Database Engineering and Applications Symposium, IDEAS 2000, September 18-20, 2000, Yokohoma, Japan, Proccedings, pages 179–187, 2000.

[9] Yván J. García, Mario A. López, and Scott T. Leutenegger. A greedy algorithm for bulk loading r-trees. In ACM-GIS '98, Proceedings of the 6th international symposium on Advances in Geographic Information Systems, November 6-7, 1998, Washington, DC, USA, pages 163–164, 1998.

[10] Taewon Lee and Sukho Lee. OMT: overlap minimizing top-down bulk loading algorithm for r-tree. In The 15th Conference on Advanced Information Systems Engineering (CAiSE '03), Klagenfurt/Velden, Austria, 16-20 June, 2003, CAiSE Forum, Short Paper Proceedings, Information Systems for a Connected Society, 2003.

[11] Bin Lin and Jianwen Su. On bulk loading tpr-tree. In 5th IEEE International Conference on Mobile Data Management (MDM 2004), 19-22 January 2004, Berkeley, CA, USA, pages 114–124, 2004.

[12] Scott T. Leutenegger and David M. Nicol. Efficient bulk-loading of gridfiles. IEEE Trans. Knowl. Data Eng., 9(3):410–420, 1997.

[13] Hee-Sun Won, Sang-Wook Kim, and Ju-Won Song. An efficient algorithm for bulk-loading of the multilevel grid file. In Proceedings of the 6th Joint Conference on Information Science, March 8-13, 2002, Research Triangle Park, North Carolina, USA, pages 382–386, 2002.

[14] Gísli R. Hjaltason, Hanan Samet, and Yoram J. Sussmann. Speeding up bulk-loading of quadtrees. In GIS '97. Proceedings of the 5th International Workshop on Advances in Geographic Information Systems, November 13-14, 1997, Las Vegas, Nevada, USA, pages 50–53, 1997.

[15] Bert Scalzo, Donald K. Burleson, and Steve Callan. Advanced Oracle Utilities. Rampant Techpress, 2010.

[16] IBM Knowledge Center. LOAD Command. `http://www-01.ibm.com/support/knowledgecenter`. [Online; accessed 10-April-2015].

[17] Microsoft. Microsoft SQL SERVER: Data Transformation Services(DTS). https://msdn.microsoft.com/en-us/library/aa933484(SQL.80).aspx. [Online; accessed 10-April-2015].

[18] Y. Dora Cai, Ruth A. Aydt, and Robert Brunner. Optimized data loading for a multi-terabyte sky survey repository. In Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom, page 42, 2005.

[19] Alexander S. Szalay, Jim Gray, Ani Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The SDSS skyserver: public access to the sloan digital sky server data. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002, pages 570–581, 2002.

[20] SkyServer Support team. Sloan Digital Sky Survey - SkyServer. http://www.skyserver.org/. [Online; accessed 10-April-2015].

[21] Yu Xu, Pekka Kostamaa, Yan Qi, Jian Wen, and Kevin Keliang Zhao. A hadoop based distributed loading approach to parallel data warehouses. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, pages 1091–1100, 2011.

[22] Liu Bin and Li Jianzhong. Practising bulk data copy in sybase database. Computer Engineering and Applications, 36(14):200–202, 2003.

[23] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. DATABASE SYSTEM Implementation(Second Edition). China Machine Press, Stanford University, 2008.

[24] Isug. Introduction to Replication Server. http://www.isug.com/Sybase_FAQ/REP/section1.html. [Online; accessed 10-April-2015].

[25] 田甜. 异构环境中并行计算模型与任务调度的研究. 硕士论文, 曲阜师范大学, 2010.

[26] Concepció Roig, Ana Ripoll, Miquel A. Senar, Fernando Guirado, and Emilio Luque. A new model for static mapping of parallel applications with task and data parallelism. In 16th International Parallel and Distributed Processing Symposium

(IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings, 2002.

[27] Hesham El-Rewini and Ted G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. J. Parallel Distrib. Comput., 9(2):138–153, 1990.

[28] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. Task scheduling in parallel and distributed systems. Prentice Hall series in innovative technology. Prentice Hall, 1994.

[29] Thomas L. Adam, K. Mani Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. Commun. ACM, 17(12):685–690, 1974.

[30] Min-You Wu and Daniel Gajski. Hypertool: A programming aid for message-passing systems. IEEE Trans. Parallel Distrib. Syst., 1(3):330–343, 1990.

[31] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. SIAM J. Comput., 18(2):244–257, 1989.

[32] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans. Parallel Distrib. Syst., 4(2):175–187, 1993.

[33] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. IEEE Trans. Parallel Distrib. Syst., 9(9):872–892, 1998.

[34] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv., 31(4):406–471, 1999.

[35] Yeh-Ching Chung and Sanjay Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In Proceedings Supercomputing '92, Minneapolis, MN, USA, November 16-20, 1992, pages 512–521, 1992.

[36] Yuan Tian, Yaoyao Gu, Eylem Ekici, and Füsun Özgüner. Dynamic critical-path task mapping and scheduling for collaborative in-network processing in multi-hop wireless sensor networks. In 2006 International Conference on Parallel Processing Workshops (ICPP Workshops 2006), 14-18 August 2006, Columbus, Ohio, USA, pages 215–222, 2006.

[37] Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, and Chao-Chin Wu. Dynamic critical path duplication task scheduling algorithm for distributed heterogeneous computing systems. In 12th International Conference on Parallel and Distributed Systems (ICPADS 2006), 12-15 July 2006, Minneapolis, Minnesota, USA, pages 365–374, 2006.

[38] Junzhou Luo, Fang Dong, Jiuxin Cao, and Aibo Song. A novel task scheduling algorithm based on dynamic critical path and effective duplication for pervasive computing environment. Wireless Communications and Mobile Computing, 10(10):1283–1302, 2010.

[39] Vivek Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press, Cambridge, 1989.

[40] Tao Yang and Apostolos Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst., 5(9):951–967, 1994.

[41] Edwin S. H. Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst., 5(2):113–120, 1994.

[42] Shu-Chen Cheng, Der-Fang Shiau, Yueh-Min Huang, and Yen-Ting Lin. Dynamic hard-real-time scheduling using genetic algorithm for multiprocessor task with resource and timing constraints. Expert Syst. Appl., 36(1):852–860, 2009.

[43] Poonam Panwar, A. K. Lal, and Jugminder Singh. A genetic algorithm based technique for efficient scheduling of tasks on multiprocessor system. In Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011 - Volume 2, pages 911–919, 2011.

[44] Mohsen Ebrahimi Moghaddam and Mohammad Reza Bonyadi. An immune-based genetic algorithm with reduced search space coding for multiprocessor task scheduling problem. International Journal of Parallel Programming, 40(2):225–257, 2012.

[45] Ricardo C. Corrêa, Afonso Ferreira, and Pascal Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996., pages 462–469, 1996.

[46] 杨传辉. 大规模分布式存储系统原理解析与架构实现. 机械工业出版社, 2013.

[47] 阳振坤. Oceanbase关系数据库架构. 华东师范大学学报（自然科学版）, 6(5):141–148, 2014.

[48] 周欢，樊秋实，胡华梁. Oceanbase一致性与可用性分析. 华东师范大学学报（自然科学版）, 6(5):103–116, 2014.

[49] Apache. Apache Hadoop. `http://hadoop.apache.org/`. [Online; accessed 10-April-2015].

[50] Alibaba. Druid. `https://github.com/alibaba/druid/wiki`. [Online; accessed 10-April-2015].

[51] Ohad Rodeh. B-trees, shadowing, and clones. TOS, 3(4), 2008.

[52] LI Kai and HAN Fu-sheng. Memory transaction engine of oceanbase. Joural of East China Normal University(Natural Science), 6(5):149–163, 2014.

[53] Li Chen, Rupesh Choubey, and Elke A. Rundensteiner. Bulk-insertions info r-trees using the small-tree-large-tree approach. In ACM-GIS '98, Proceedings of the 6th international symposium on Advances in Geographic Information Systems, November 6-7, 1998, Washington, DC, USA, pages 161–162, 1998.

[54] Rupesh Choubey, Li Chen, and Elke A. Rundensteiner. GBI: A generalized r-tree bulk-insertion strategy. In Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, China, July 20-23, 1999, Proceedings, pages 91–108, 1999.

[55] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. PVLDB, 4(11):795–806, 2011.

[56] Pei-Lun Suei, Victor C. S. Lee, Shi-Wu Lo, and Tei-Wei Kuo. An efficient b^+-tree design for main-memory database systems with strong access locality. Inf. Sci., 232:325–345, 2013.

[57] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. Acta Inf., 9:1–21, 1977.

[58] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[59] Dennis Shasha. Review - efficient locking for concurrent operations on b-trees. ACM SIGMOD Digital Review, 1, 1999.

[60] Goetz Graefe. A survey of b-tree locking techniques. ACM Trans. Database Syst., 35(3), 2010.

[61] Kerttu Pollari-Malmi, Eljas Soisalon-Soininen, and Tatu Ylönen. Concurrency control in b-trees with batch updates. IEEE Trans. Knowl. Data Eng., 8(6):975–984, 1996.

[62] 房友园. 面向海量文本数据的加载技术的研究与实现. 硕士论文, 国防科学技术大学, 2005.

[63] 华强胜. 基于DAG模型的高效并行任务调度算法研究. 硕士论文, 中南大学, 2004.

[64] Boontee Kruatrachue and Ted Lewis. Grain size determination for parallel processing. IEEE Software, 5(1):23–32, 1988.

[65] Carolyn McCreary and Helen Gill. Automatic determination of grain size for efficient parallel processing. Commun. ACM, 32(9):1073–1078, 1989.

[66] Yiqun Ge and David Y. Y. Yun. A method that determines optimal grain size and inherent parallelism concurrently. In 1996 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '96), June 12-14, 1996, Beijing, China, pages 200–206, 1996.

[67] Apostolos Gerasoulis, Sesh Venugopal, and Tao Yang. Clustering task graphs for message passing architectures. In ICS, pages 447–456, 1990.

[68] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. IEEE Trans. Parallel Distrib. Syst., 4(6):686–701, 1993.

致　　谢

转眼间，三年的研究生生涯即将结束。对比大学刚毕业的自己，在海量所的三年研究生生活让我在学习、工作、以及为人处世方面，都有很大的改变和提升。在这三年的时间里，实验室的老师、同学、朋友和家人都给予我很多关心和帮助。在毕业论文即将完成之际，我在此深深地感谢所有给予我关心帮助的人们。

首先，我要向我的导师周傲英教授致以衷心的感谢和崇高的敬意。周老师以他渊博的学术知识、严谨的科学态度和精益求精的工作作风给我留下深刻的印象，每次听他的报告都让我受益匪浅。周老师虽然工作繁忙，但是经常会在学术和生活上给予我们关心和帮助。周老师的谆谆教诲将会对我以后的生活和工作产生深远影响。

同时，我要特别感谢我的指导老师周敏奇老师，感谢他在我研究生阶段的学习和生活上的关心和支持。在读研期间，周老师不辞辛劳地指导我学习和论文，并给我提出了很多建议。周老师不仅传授知识，教会我们学习和思考的方法，还在生活和工作上交给我很多为人处世的道理。在我以后的生活和工作中，我会谨记周老师的教诲，也会永远感谢他的指导。

接下来，我还要感谢实验室的其他老师们。首先，我要感谢宫学庆老师、张蓉老师、张召老师、高明老师，感谢他们对我在交通银行一年的工作的帮助和支持。同时，我还要感谢王晓玲老师、金澈清老师、何晓丰老师等，感谢每一位老师给予我的关心和帮助，感谢他们在这三年里为我们提供了良好的学习环境和实践机会。感谢所有关心我、帮助我的老师们，您们的无私奉献和悉心指导使得我顺利毕业。

另外，我还要感谢身边的同学们。感谢王立、张磊、顾伶、董绍婵、张新洲以及其他CLAIMS组的成员，感谢他们对我在学习和学术上的帮助；感谢徐晨、马海欣、夏帆、于程程、朱涛等原110实验室的师兄师姐们，感谢他们在我研究生初期时给予的关心和指导；还要感谢翁海星、庞天泽、周欢、张晨东、刘骁、樊秋实等交行组的小伙伴们，感谢他们对我在交行工作的支持和帮助；感谢海量所的每一位同学，海量所就像一个大家庭，每个人都给我带来了欢乐，和他们在一起生活和学习是一件很开心的事情。感谢那些一直关心和支持我的同学和朋友们，感谢你们让我拥有一个充实和欢乐的研究生生涯。

此外，我还要特别感谢我的奶奶、我的父母、以及其他关心我的家人。从小到大，父母从来不关心我的学习和生活，但他们一直支持我的选择，并默默地付出着。没有他们为我创造的学习条件和生活环境，我也不会有机会完成研究生学业。在这里我要由衷地感谢我的父母，感谢他们的关心、支持和鼓励。

最后，向所有关心和帮助过我的人们致以由衷的感谢!

李永峰

二零一五年四月二十日

攻读硕士学位期间发表论文和科研情况

■ 已发表或录用的论文

[1] **李永峰**, 周敏奇, 胡华梁, 集群资源统一管理和调度技术综述, 华东师范大学学报（自然科学版）2014, 17-30. (CSCD)

■ 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