申请上海交通大学工程硕士学位论文

一种数据库复制数据流框架设计和实现

学校代码： 10248
作者姓名： 罗健
学　　号： 1100379114
第一导师： 胡飞
第二导师： Heping Shang
学科专业： 软件工程
答辩日期： 2015 年 7 月 24 日

上海交通大学软件学院

2015 年 7 月

A Dissertation Submitted to Shanghai Jiao Tong University

for Master Degree of Engineering

DESIGN AND IMPLEMENTATION OF DATA FLOW FOR DATABASE REPLICATION

University Code：	10248
Author：	Jian Luo
Student ID:	1100379114
Mentor 1：	Fei Hu
Mentor 2:	Heping Shang
Field：	Software Engineering
Date of Oral Defense：	2015/7/24

School of Software

Shanghai Jiaotong University

July 2015

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：罗健

日期：　2015 年 6 月 25 日

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：罗健　　指导教师签名：胡飞

日期：　2015 年 6 月 25 日　　日期：　2015 年 6 月 25 日

一种数据库复制数据流框架设计和实现

摘 要

在数据库管理(Database Management)领域，实时或者近实时的数据库数据复制(Real Time or Near Real Time Database Replication)始终是企业级数据管理领域的一个重要主题。大数据时代的来临对数据库管理领域的影响是深远和长久的。本论文的研究目标是设计一种高性能数据库复制数据流框架，基于该框架的数据库复制系统能够充分利用现有的软件、硬件的技术进步，从而提供高速的数据库数据复制服务。

本论文的主要工作和贡献如下：

1. 设计和实现了一种基于并发执行的子任务流水线架构，以此实现高性能的数据库复制系统。从系统开发角度，子任务是完成数据复制的分解步骤的过程描述，与具体的运行线程隔离，从运维角度，用户可以通过调节同时执行的子任务的个数来调控数据库复制的性能和计算资源占用的关系。
2. 研究分析了不同事务之间数据变更的相互依赖关系，将无依赖的变更并发复制到目标数据库中，充分利用了目标数据库自身的并发处理能力，从而提高了数据复制的性能。
3. 设计了一种可以计算一段时间内所有事务净变更的计算方法，从而减少了需要写入目标数据库的变更操作个数，提高了特定场景下的数据复制的性能。
4. 设计了一种简单的自动调节事务提交尺寸的算法，提高了数据库的事务吞吐量，并进行了单项性能测试。
5. 研究了特定数据库编程接口对数据复制性能的影响，包括预编译 SQL 接口和 BULK LOAD 接口，并对其性能进行了对比实验验证。
6. 通过记忆数据库对象的复制状态、机器结构的变更历史，实现了真正的数据库日志重播功能。

测试结果表明，本文提出的数据库复制数据流框架，能够提高数据库复制的吞吐性能，与基准系统的性能相比较，在不同的测试实验中均取得 2 倍至 10 倍的性能提升。

关键词 数据库复制，复制性能，数据流框架，并发处理

DESIGN AND IMPLEMENTATION OF DATA FLOW FOR DATABASE REPLICATION

ABSTRACT

Near Real Time Database Replication is an important component of enterprise database management system. There is a big and long-lasting impact from data explosion age. According to McKensey's estimation， total data volume in enterprise will increase by 44% every year. It also estimates that the total data volume stored in all enterprise information system will be increased by 44 times. In order to digest use information in such as a large volume of data， enterprise needs to utilize proper tooling and architecture. However， according to one of a survey by Information Week Reports in 2012， more than 56% enterprise does not treat "big data" differently. More than 90% of enterprises are still depending on traditional RDBMS to store and process data. This means， in the coming years， RDBMS will still play an important role in data management. So， database replication system needs to carry larger and larger data volumes between database systems.

The major contributions of this paper are:

1. Design and implement a task-based pipeline for better database replication performance. A task is a pure computing unit that can be executed by a thread in a thread pool. Task-based programming model splits the computing unit and executing thread so that each of them can be controller separately. Programmer can add a new task without considering in which thread the task will be running， system administrator can tune the size of thread pool to balance between utilization of CPU resources and performance of replication.
2. Propose a way to improve replication performance by analysis inter-transaction dependencies so that transactions that are not inter-related can be replicated

concurrently into target database system. It makes better use of concurrent execution capability of database system.

3. Design a method to compute "net-data-change" to reduce the number of data changes needed to be applied to target database. This helps improve replication performance in some situations.
4. Combine smaller transactions into larger transactions. Conducted an experiment to show that the commit rate of database system is limited by the IOPS of disk sub-system. Propose a simple algorithm to automatically tuning the size of commit transaction to approach the IOPS of disk sub-system of target database system.
5. Propose to use Prepared/Dynamic SQL statement API and BULK LOAD API. These APIs can help improve replication performance latency.
6. Propose a way to manage table metadata and replication states change history. With it, database change replay becomes easy to implement.

Keywords database replication， replication performance， replication performance test， data flow architecture

目 录

1 绪 论

1.1 研究背景及意义

在数据库管理领域，实时或者近似实时的数据库数据复制(Database Replication)技术始终是企业数据管理的一个重要主题。几乎所有的关系型数据库管理系统都提供了数据复制的功能。简单的说，数据库数据复制就是将一个数据库系统中的数据（包括表数据以及所有用户变更）无偏差的、迅速的复制到另外一个数据库系统中。总体而言，数据库数据复制的目的是为了提高数据库的可用性（availability），或者提高数据库的数据访问性能（data access performance）。 在实际应用中，数据库数据复制技术主要应用在实时容灾备份和恢复、实时商务智能、报表等常见企业数据库应用场景[1][2][3]。数据库数据复制的方式通常有两种：整表复制(full-table replication)和增量复制(data change replication)。整表复制是将一整张表中已经存在的数据行全部一次性复制到目标数据库系统中。增量复制则是复制一张表中自某一个时间点以来的全部的变更到目标数据库系统中。整表复制主要应用在对数据实时性要求不高的场合。增量复制主要应用在对数据实时性有一定要求的场合。本文主要研究增量复制的高性能复制技术。

随着大数据时代的来临，对数据库管理领域的影响是深远和长久的。根据麦肯锡的估计，企业所保有的数据容量每年提高 44%，2009 年至 2020 年间，企业系统中的数据的总容量将增长 44 倍。为了能够以实时或者近似实时的方式访问大数据中所蕴涵的信息，企业必须使用合适的工具和架构。然而，根据 Information Week Reports 在 2012 年的一次调查中，超过 56%的企业并不区分大数据和传统数据，90%的企业依然将传统的数据库系统作为数据处理的主要方法。这意味着，在相当长一段时间内，数据库系统依然会承担数据管理的主要角色，而数据库数据复制系统所需要承载的数据量也将会越来越大[4][5][6]。因此，研究如何提高数据库复制技术，提高系统处理大数据的能力是十分必要和迫切的[7][8]。

1.2 现有数据库复制产品简介

现有的数据库复制系统设计开发于上世纪 90 年代末，其最初设计目标是在一个公司内实现跨机构、跨地区的数据库近似实时数据复制和灾难备份。跨机构是指将源数据库中的表数据变更按照一定的规则分发复制到不同机构的数据中心的目标数据库中。跨地区是指将源数据库中的变更数据通过广域网传递到处于另外一个物理地址的数

据中心，然后复制到目标数据库中。可以看出，该系统的设计初衷是数据库的近似实时的灾难备份。当时的设计重点是如何解决传输线路的不稳定和目标数据库可能会随时暂停的问题。

为了解决传输线路的不稳定问题和目标数据库可能会随时暂停的问题，现有的复制产品采用了所谓先保存再发送的数据流程：即当一个数据变更进入复制服务器后会被首先保存到一个基于磁盘的入站队列，然后再进行接下来的处理。当一个变更事务被写入到目标数据库之前会被首先保存到一个基于磁盘的出站队列，然后再写入目标数据库。这样做的好处是，数据流不会因为链路中断或者目标数据库的中断而停下。例如，当一个数据变更通过卫星链路发送到处于不同物理位置的复制服务器时，如果发生了链路中断，则所有数据变更依然可以被写入到位于磁盘的出站队列中，从而保证从源数据库发送来的数据变更的处理数据流不会中断，当卫星链路恢复后，出站队列中的数据变更会被继续送往目标复制服务器。

该系统的实现采用了当时流行的多线程、流水线式的设计，如图 1-1 所示。

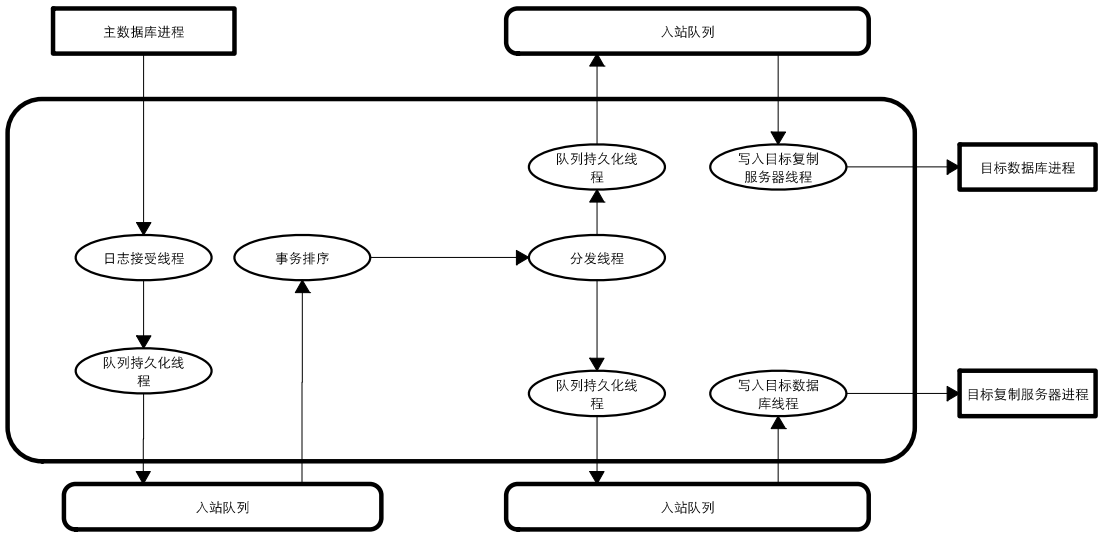

图 1-1 现有产品的数据库复制数据流

Fig.1-1 Dataflow diagram for existing product

图 1-1 是现有产品的数据复制数据流概念图。该架构采用了当时流行的多线程、流水线式的设计。图中展示了从一个主数据库复制到两个从数据库的数据流。椭圆形代表一个功能性线程。箭头代表数据流流向。正方形代表一个磁盘上的队列。可以看

出主要的线程主要有日志接受线程、事务排序线程、分发线程、写入（目标数据库或者目标复制服务器）线程。当用户在源数据库执行了一些数据变更后，源数据库会将这些变更事务写入数据库日志文件，这些日志变更信息会被日志扫描守护进程捕获并发送到复制服务器，所有的数据变更都会经过这 4 个主要线程的处理，并且会被两次保存到磁盘队列中，即入站队列和出站队列。

(1) 日志接收线程（Log Change Capture Thread）

日志接收负责接收由日志扫描守护进程发送来的数据库日志记录。并转换成日志记录对象。最后将日志记录对象提交给入站队列持久化线程进行持久化。

(2) 持久化线程(Log Change Persistent Thread)

分别有入站队列持久化线程和出站队列持久化线程。它们的主要职责是将变更对象或者变更事务写入磁盘队列中。

(3) 事务排序线程(Transaction Sorting Thread)

扫描变更事务，并按照事务体教的顺序进行排序。

(4) 分发线程(Distributor Thread)

将变更事务分发到目标数据库或者复制服务器所对应的出站队列中。

(5) 事务提交线程(Transaction Executor Thread)

将变更事务复制到目标数据库中。

通过使用标准化的测试数据产生器（详见第五节性能验证测试）产生大量数据库变更，测量现有数据库复制产品复制数据库变更所需要的时间，并采集其在复制过程中的系统行为，可以得出如下结论：

(1) 目标数据库的写入性能不是瓶颈

通过比较测量产生数据库变更所需要的时间和复制这些数据变更所需要的时间，发现复制变更所需要的时间较长。这意味着数据库的处理性能并不是瓶颈。

(2) 日志捕获性能不是瓶颈

通过测试日志捕获线程捕获全部测试数据变更所需要的时间，发现捕获日志时间比产生数据变更所需要的时间还要短。这意味着日志捕获性能并不是瓶颈。

(3) 并发事务提交算法过于简单

现有的并发事务提交算法是假设 2 个连续事务没有关联性：即按照事务提交的先

后次序提交 BEGIN TRAN 命令后，通过 2 个线程同时执行这 2 个事务的变更动作，最后再按照事务提交的先后次序提交 COMMIT TRAN 命令。但是，在线事务处理的事务往往具有关联性。面对具有关联性的 2 个事务，数据库可能会锁住这 2 个事务中的任何一个直到另外一个事务提交，从而导致死锁可能，并且随着并发提交线程个数的增多，死锁的概率也变大。即便没有死锁，这 2 个事务也降级为顺序执行提交，失去了并发提交的性能提升。

(4) 无区别对待全部事务类型，不能使用最有效的数据复制方法加快复制

比如，对于常见的数据导入操作，即连续的单表、大规模的插入操作，各个主流数据库系统都提供了更高效的编程接口。如果能够识别这些特定形式的事务类型，则可以采取最恰当的编程接口来复制数据。

(5) 出站队列设计带来的好处并不明显，并可能导致磁盘资源占用更多

出站队列的设计假设是：当目标数据库连接终端或者复制提交较慢时，保存尚未提交到的变更事务，从而可以尽快的将这些事务从入站队列中移出。但是实际情况是，如果目标数据库的复制不够快或者连接终端，出站队列的空间最终会被耗尽，直至入站队列也被耗尽。通过合理的设计，完全可以避免引入出站队列。

1.3 国内外研究概况

经过十几年的发展，数据库复制技术已经被广泛应用在不同的数据库管理场景。

数据库复制技术主要分为主动复制型和被动复制型两种，基于主动复制型的复制技术能够使主—从数据库始终保持一致，即当一个事务在主数据库提交时，该事务所包含的数据变更也同步提交到从数据库。这种复制技术的缺点是，事务在主数据库的执行时间会延长，这是由于事务执行的过程中需要将事务变更同步发送到从数据库。基于被动复制型的复制技术以异步的方式将主数据库的变更提交到从数据库中，这种复制技术对主数据库中的事务执行时间影响很小。如果主、从数据库的物理位置较远，就必须使用被动复制技术。

目前，大部分主流数据库复制产品都是基于被动复制型的复制技术。从获取变更的方法来看，已经发展出如下几种数据变更捕获技术：

（1）基于数据库触发器[9][10]

几乎所有主流的数据库系统都提供了触发器的功能，通过特定编写触发器可以获得用户对主数据库中某张表的全部变更历史。触发器编写简单，但是由于触发器捕获变更数据的过程为用户事务的一部分，对用户事务的性能有一定的影响。当用户的表

结构发生变化时，触发器就要进行响应更新，这给数据库管理带来一定的问题。

（2）基于数据库的恢复日志[11][12][13]

数据库的恢复日志记录了用户对数据库的所有变更历史信息，通过数据库的恢复日志，可以在不影响用户事务性能的前提下获取变更数据。读取和分析数据库的恢复日志，需要有能力理解恢复日志具体格式信息的能力。Sybase Replication Server[14][15]就是使用基于数据库恢复日志进行数据库数据变更捕获的典型产品。

（3）基于客户端驱动程序层[16][17]

用户对数据库的全部变更都会经过客户端的驱动程序，某些数据库提供了基于驱动程序的数据复制，当用户通过该驱动发出全部 SQL 语句，会被转发给从数据库进行复制执行。

（4） 变更跟踪表[18]

由数据库将用户对某张表格的全部变更历史写入到另外一张表格中。然后在数据库可以通过这张表格中的信息进行更新。

（5）基于行时间戳[19]

该技术利用数据库的特性，即一张表中所有的数据行都有一个时间戳，这个时间戳是该行最后一次被更新时的系统时间。复制程序通过不断查询该表中所有时间戳大于上一次查询时的系统时间的数据行得出需要复制的内容。

（6）基于影子表[20][21]

影子表(Shadow Table)的实现方法较为直观，既首先将需要跟踪变更的表的全部数据复制到另外一张结构相同的表中，然后在某一个适当的时间点，比较这两张表的内容以获取净变更的信息。该方法的优点是可以在任何关系型数据库上实现，适应性较好，缺点是实时性不可能很高，并且需要大量表空间用于保存影子表。

目前，基于数据库日志的数据复制技术基本上被国外数据库源厂商所垄断。主要有如下几个著名的数据复制软件产品：

(1) Sybase公司的Replication Server[22][23][24]

Replication Server 最初是 SYBASE 于 1994 年为美国银行开发的数据库复制系统。最初仅支持 Sybase ASE 数据库之间的数据复制。后来经过长期的开发完善，已经能够支持 ORACLE， DB2， MSSQL， Sybase ASE 数据库之间的数据复制。其是世界上

第一款通过读取源数据库的日志获取变更信息的数据库复制软件。被大部分 Sybase ASE 的客户应用在生产系统中，在国内也有较多的客户。

(2) 甲骨文公司（ORACLE）的Data Guard[25]

Oracle Data Guard是ORACLE公司为其数据库系统设计开发的实时双机热备软件解决方案。其通过读取ORACLE数据库的重做日志（Redo Log），实现主数据库和备数据库的事务同步。该方案只支持ORACLE数据库到ORACLE数据库的数据复制。改产品主要被用于ORACLE数据库的容灾备份。

(3) 甲骨文公司(ORACLE)的Golden Gate[26]

Oracle Golden Gate是ORACLE于2010年收购的近实时数据库复制软件产品。支持多种数据库（包括ORACLE， DB2， MS SQL， Sybase ASE）为源的数据复制。工作原理是通过读取源数据库的日志文件中的变更信息，然后将变更信息应用到目标数据库中。目前，该产品是ORACLE公司主推的ORACLE数据库复制系统。

(4) IBM公司的Info Sphere Replication Server[27]

IBM Info Sphere也是一款通过读取源数据库的日志获取变更信息的数据库复制软件产品。该产品的主要特点是与IBM DB2结合较为紧密，其变更信息的传送采用了著名的IBM MQ.

1.4 课题研究目标

本课题的研究目标是：如何通过一定的设计、技术和实现手段，优化并提高数据库复制系统的复制性能，使其复制能力能够超过 100GB 每小时，与此同时保证数据复制系统具有一定的伸缩性，能够在 1~8 个 CPU 内核的硬件资源配置范围内进行复制性能的伸缩。

1.5 主要研究内容

论文将研究如下技术以提高数据库复制的复制性能以及可伸缩性：

(1) 简化数据复制数据流程步骤，为数据继承和数据分析的应用场景进行优化

- 为 1 对 1 数据库复制应用场景进行优化
- 取消出站队列情况下的截断点管理

(2) 提高数据复制的处理和写入的并发度，实现超过 4 倍以上的数据复制性能提高

- 向目标数据库并发提交不同事务的设计和实现；
- 向目标数据库并发提交同一个事务的设计和实现；
- 减少向目标数据库提交的变更数量的设计和实现
- 不同的数据库的接口以及写入数据的方法对数据复制的性能影响；

充分利用现代 CPU 的多核心的并行处理能力，使得复制流水线的处理步骤能够尽可能并行化，减少顺序处理环节，增加并行处理环节，从而提高实现流水线的吞吐能力。比如对数据库日志记录的并发解释极大的提高了日志记录的处理性能。

1.6 论文结构安排

本论文共分 6 章。

第 1 章 绪论，简单介绍了本课题的研究背景、内容和目标，回顾了现有数据库复制产品的设计历史，对其设计和性能特点做了简单的介绍，此外还介绍了市场上其他数据库复制产品技术的情况。

第 2 章 主要回顾了基于数据库恢复日志技术的数据库复制系统的基本运行原理和相关实现技术，为后续章节的内容介绍做一定的基础内容铺垫。主要介绍了数据库恢复日志所包括的信息及其在数据复制中的作用，事务排序的概念及其对保证目标数据库的事务完整性和一致性的意义，还介绍了数据截断点的管理及其如何避免提交重复变更事务到目标数据库的办法。最后，由于使用了 JDK7 提供的 Fork-Join 框架，本章也对其原理和基本使用方法做了简单介绍。

第 3 章 从总体设计角度介绍了新的复制系统的复制数据流的设计，描述了简化后的数据库复制数据流的各个步骤的，在本章还描述了取消出站队列后的截断点管理原理。

第 4 章 详细介绍了提高数据库复制性能的关键技术。包括合并变更事务，使用并发提交不同的合并后变更事务的原理和算法，还探索了并发提交相同合并后的变更事务的原理和算法，还介绍了利用不同数据库的编程接口实现告诉数据写入的技术，此外，还对这些技术和算法进行了单项性能测试以研究其性能优点和缺点。

第 5 章 介绍了测试与验证方法以及测试数据。使用了 3 中不同的测试案例对新的

数据复制系统进行了端到端的性能测试，并对结果进行了探讨。

第 6 章 全文总结。

2 数据库复制原理和技术

本章主要介绍了基于数据库恢复日志的数据复制基本原理和技术，包括事务排序，即保证目标数据库的事务一致性的原理；如何维护日志截断点，即如何保证源数据库中的尚未被复制的日志不被意外截断；如何避免重复提交，即数据复制在中断后重新开始复制后，如何保证不向目标数据库提交已经复制的变更事务。此外，由于大量使用了 JDK7 中的 fork-join 框架[28]，在本章也会对其进行简单介绍。

2.1 基于数据库恢复日志的数据库复制的基本原理

恢复日志是所有关系型数据库系统的重要组成部分之一[29][30][31]。当数据库系统崩溃后，依靠恢复日志，数据库系统依然能够将数据库恢复到事务一致的状态。对每一个数据库的表数据的变更操作，数据库都会产生一条对应的日志记录并写入数据库日志文件中，当一个事务提交时，数据库系统会首先保证改事务所产生的事务日志全部被刷新落实到数据库日志文件中，而对表数据文件的更新则是异步地刷新落实到磁盘。

基于数据库恢复日志（Database Recovery Logging）的数据库复制技术是指利用数据库的恢复日志中的日志记录捕获用户对数据库所有变更操作历史，并将这些变更操作转换为目标数据库所接受的操作写入的技术。相比于其他复制技术，该技术的特点是对源数据库的应用程序完全透明，相比其他数据变更捕获技术，对源数据库的影响最小，可以获得用户对源数据的全部数据变更的最完整的信息，因此可以将用户变更事务完整、真实的复制到目标数据库中。缺点是实现和维护的技术难度较高，这是由于每一个数据库厂商的日志类型、格式和捕获方式完全不同，甚至相同数据库的不同版本也会有一定的差异，因此日志处理的步骤对每个数据库都是不一样的。

虽然每一个数据库系统的日志记录格式都不尽相同，但是从逻辑角度而言，日志记录都包括了日志头和日志体两大部分。在日志头的最开始是日志记录的长度信息，通过累加这个长度信息可以计算出下一条日志记录的存储位置进而读取到下一条日志记录。每一条日志记录都有一个唯一的逻辑序号用于唯一标示这条日志记录，通过这个逻辑序号，可以推算出这条日志记录在存储设备上的保存位置近而可以读取到这条日志记录。日志头中还有一个重要字段叫日志操作类型，这个类型决定了日志体中所保存日志数据的具体格式。日志类型多种多样，不同的数据库系统定义了不同种类的

日志类型，但是对于数据复制而言，只有一小部分类型的日志是有用的。例如和数据变更（DML）相关的日志类型：插入(INSERT)、更新(DELETE)、删除(DELETE)、全表清除（TRUNCATE TABLE）等。也包括一些和事务控制有关的操作，例如事务开始(BEGIN TRANSACTION)、事务提交(COMMIT TRANSACTION)、事务回滚(ROLLBACK TRANSACTION)等。在和数据变更相关的日志体中记录了这条日志记录所影响的表格在数据库字典中的编号以及所影响到的表的数据行在变更前后的各个列的值。在日志头部中还有另外一个字段和事务边界识别相关的字段，即：事务序号。数据库系统为所有的事务赋予了一个唯一的序号，当前日志记录里的事务序号表明这个日志记录是由哪一个事务所产生的。日志记录和用户的 SQL 语句并不一一对应。一条日志记录仅仅用于记录数据库系统对某一行用户的表数据的变更历史。例如，如果一条 DELETE SQL 删除了 1000 条表数据，则这条 SQL 会产生 1000 条操作类型为 DELETE 的日志记录。

数据变更类的日志记录中记录了被影响的数据行的各个字段的值，对于更新操作，日志记录中将记录更新前和更新后各个字段的值。对于删除操作，日志记录中将记录被删除数据行的各个字段的值。对于插入操作，日志记录中将记录被插入数据行的各个字段的值。为了节约磁盘空间，日志记录中将省略所有字段的名称以及类型。日志记录中仅仅包括每个字段值的起始偏移量。对于每一种给定的字段类型，其值在日志记录中的保存格式是确定的。数据库还有一类特变的数据变更，我们成为之DDL(Data Dictionary Language)变更，即对数据库对象（例如表、字段、约束、索引）的创建，删除，修改的变更。这类变更一般是由一组连续的数据类变更日志记录组成。这是由于，数据库中所有的数据对象的定义信息都是保存在一组称之为数据字典的表格中。对数据库对象的变更就是对这些数据字段表的变更。

由此可见，复制系统通过扫描数据库日志文件可以获取所有数据库的变更历史信息。由于日志记录都是以一定的二进制格式写入数据库的日志文件中，复制系统需要根据具体数据库的日志格式信息才能正确地读取每一条日志记录。每一个数据库系统都用自己独有的日志存储格式，为了能够方便的访问每一条日志记录中被复制所需要的信息，需要将日志记录的存储格式，转换成复制程序内部的数据对象中。这个过程用计算机的术语就是反序列化。这个过程类似于编译原理里的编译过程：将一个程序解析成可执行程序，只不过在这里的编译输入是日志记录，输出是变更对象。由于日志结构的复杂性，该过程是复制系统中占用 CPU 时间较多的步骤。

事务的特性之一是原子性（Atomicity），即一个事务对数据库的更改要么全部发生，要么完全没有发生[32]。不同事务的日志记录是交错地写入数据库的日志文件中。为了将各个事务各自包括的日志记录都抽取出来，必须对日志记录进行按事务为单位归

并。 只有这样才有可能按事务为单位将数据变更复制到目标数据库中。事务的另外一个特性是个隔离性(Isolation)，虽然数据库系统可以同时并发处理大量的事务请求，但每一个事务都感觉系统中只有自己一个活动的事务。一个事务所做出的变更，只有在这个事务被提交以后，才能被其他事务读取到。例如，在大多数情况，当某一个事务更新了一行被另外一个尚未提交的事务更新过或者删除的数据行时，该事务会被暂时挂起或者回滚。正是这一个又一个被提交的事务，将源数据库从一个状态改变为另外一个状态。为了保证目标数据库的状态变更与源数据库保持一致，我们必须保证所复制的事务在目标数据库的提交顺序，严格按照其在源数据库提交时刻的前后次序保持一致。我们称这个过程为事务排序，而这种保持目标数据库与源数据库在事务级别上的一致性为目标的复制技术，称为基于事务的数据库复制技术(Transaction Replication)。

但是在某些应用场合，用户并不严格关心目标数据库与源数据库能否保持事务级别的一致性，他们只需要确保目标数据库最终能够与源数据库保持事务一致性。例如，在向数据仓库的复制过程中，用户只需要数据仓库中有前一天的全部数据库。在这种情况下，事务排序则可以省略。我们只需要按照日志记录在日志文件中的原始顺序将数据变更复制到目标数据库即可。我们称这种复制记录为最终一致性复制（Eventual Consistent Replication）。

基于事务的复制和最终一致性复制各有优缺点。基于事务的复制必须经过事务合并和排序的步骤，有较高的复制延时，这是由于复制系统必须遇到一个事务的提交日志记录之后，才能开始复制这个事务所包括的第一条数据变更。但是在一般情况下，事务所包括的日志记录都不会太多，整个事务的执行时间跨度也不会很长，这种延时一般不会成为性能瓶颈。但是在一些批量处理的过程中，例如银行在夜间的对账过程，会涉及到一些包括操作非常多、执行时间特别长的事务，对于这种事务的复制，基于事务的复制技术会产生较大的延时。目前通常的解决办法是，如果某一个事务所包括的日志记录个数超过一个阀值，则不论是否遇到这个事务的提交命令，该事务的数据变更都会开始向目标数据库开始复制。

为了将一个日志变更写入目标数据，必须将日志变更转换为对某种数据库接口的调用[33]。一般有两类接口，一类是基于 SQL 语言的接口，即将日志记录转换为具体的 SQL 命令在目标数据库进行执行；另外一种是基于数据库存储层的接口，即将日志记录所包括的信息直接传入数据库的存储层接口，从而直接写入数据库的存储设备。基于 SQL 语言的接口是最直观、简单、通用的接口。所有的关系型数据库系统都支持 SQL，因此通过 SQL 接口可以很容易实现支持不同种数据库系统的功能需求。但是由于数据库系统在执行 SQL 语句时，都需要经过解析、正交化、权限检查、查询优化等

步骤最后才会被数据库系统的执行引擎执行，而数据库复制所涉及到的 SQL 命令种类是非有限(INSERT，DELETE，UPDATE，BEGIN，COMMIT，ROLLBACK，TRUNCATE 等)，如果数据库系统能够在存储层直接提供对这些操作的执行接口，则可以大幅减少 SQL 语句所引入的额外处理过程从而获得更高的处理性能。

为了将一条日志记录翻译成对应的 SQL 语句，必须获得日志记录所对应的表名、字段名、字段类型等信息。这些信息保存在数据库的数据字典中。如果源数据库系统在被复制的过程中没有对数据字典的变更，即对数据对象的结构的变更，则复制系统是否可以通过读取数据字典来获取这些表对象的结构信息。一旦遇到源数据库对象的结构变更，例如某一张表删除了一个字段，我们可以要求用户对这张表进行重新同步的操作。为了减少对用户的影响，现在的数据库复制系统一般都维护了一套基于版本的数据字典库，即保存全部的源数据库的表对象的结构信息及其历史变更版本。由于所有对表结构的变更操作最终就是对数据库数据字典表的事务变更，因此可以认为，数据字典里的信息被变更事务从一个版本变更为另外一个版本。

由于磁盘空间有限，为了避免数据库恢复日志无限制增长，数据库管理员需要定期将已经落实的部分恢复日志文件进行截断删除操作，该过程称为日志截断[34][35]。被截断的位置称为截断点。所有截断点前的恢复日志所占用的磁盘空间会被回收重用。由于复制系统依赖于恢复日志，必须确保那些尚未被读取和复制的日志记录不会被数据库系统截断。为此数据库系统提供了第 2 截断点，该截断点由复制系统根据复制进度不断移动。由数据库系统确保所有在第 2 截断点前的日志记录不会被截断。

为了保证已经提交到从数据库系统中的事务不被重复提交，复制系统需要记录最近已经成功提交的事务的序号。复制系统为每一个变更对象赋与了一个唯一的 ID，这个 ID 包括若干信息并且保证互相不重复。通过 ID 信息，可以在数据库日志文件中唯一定位一个变更对象对应的变更日志。一个事务的序号就是改事务的提交日志对象的 ID。最后提交事务的 ID 会被更新到目标数据库的一个表中。

2.2 源数据库、日志捕获和复制和目标数据库的隔离

源数据库的隔离，主要是指隔离不同源数据库系统在数据字典存取方式的不同。数据字典本身是所有数据库系统都具有的基本元数据集合，但是每一个数据库管理系统都采用了不同的方式组织和存储这些元数据。复制系统需要获取和保存所复制表格的数据字典信息。为了尽可能提高数据复制系统的通用性，必须将不同的数据库的数据字典的差异进行封装。

日志捕获的隔离是由于不同的数据库的恢复日志的物理格式和读取方式完全不同

，为了简化数据库复制逻辑的设计，必须将数据库日志捕获和数据库复制进行技术化的隔离。隔离的办法是设计一种独立于具体数据库之上的中间日志记录格式。中间日志记录具有足够的表达能力和紧凑的存储格式。我们可以为不同的源数据库系统设计开发不同的日志捕获程序，日志捕获程序将具体的日志记录转换翻译成中间的日志记录格式。

目标数据库的隔离主要指对同一种数据存储操作，不同的数据库管理操作可能提供了不同的操作接口和不同的表达命令。为了简化数据复制系统执行数据变更写入逻辑的复杂性，必须将不同数据库系统的数据存储操作方法进行封装。

从上面介绍可知，如果将源数据、日志捕获、目标数据库与数据复制系统的接口进行适当封装，数据库复制系统的实现逻辑可以最大限度的做到中立，即与具体数据库管理系统无关，为今后支持更多种类的数据库管理系统及其更高版本奠定了技术架构基础。图 2-1 体现了这种隔离的思想，数据库复制系统只需要与源数据库日志捕获接口，源数据库数据字典读取接口以及目标数据库存取接口即可以实现数据复制的功能。

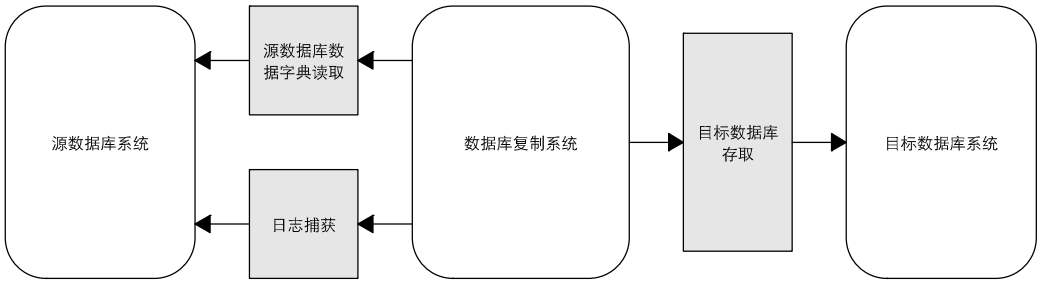

图 2-1 数据复制系统与源数据和目标数据库的隔离

Fig.2-1 Separate replication concerns from source and destination database systems

2.3 数据库复制的初始化原理

我们称基于数据库日志记录的复制为增量复制。然而在开始基于数据库日志记录的数据库复制之前，必须首先将源表中的所有数据行整体复制到目标表中，然后才能开始基于日志记录的数据复制操作，我们称这个步骤为复制初始化操作。

通常的做法是首先停止生产系统以便暂停一切对源数据库的变更操作，然后开始将所有需要复制的源表整体复制到对应的目标表中，然后获取源数据库系统当前最后

一条日志记录的序号，最后重启生产系统。数据库日志捕获程序则从之前获取的日志记录序号开始将新的日志记录发送给数据库复制程序进行增量复制[36][37]。

现在流行的办法是不暂停生产系统的前提下进行复制初始化操作进而切换到增量复制状态[38]。其设计原理是如图 2-2 所示：

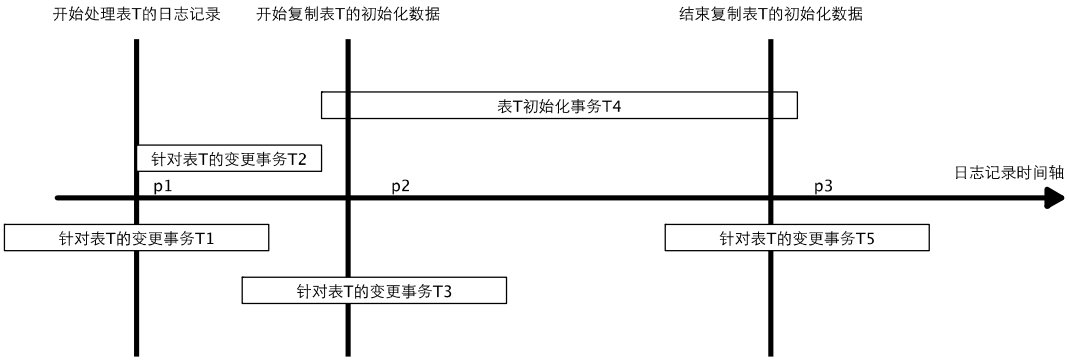

图 2-2 单表初始化

Fig.2-2 Replication initialization for a single table

由图 2-2 所示，我们将表 T 的初始化过程封装在事务 T4 中完成，并且指示日志捕获程序在 P1 时间点开始传送表 T 的日志记录到复制系统，由于表 T 尚未被初始化成功，数据库复制系统会将事务 T1~T4 暂时存放在一个独立的队列中，其他表的事务则按照正常流程继续复制。

当表 T 的初始化数据全部复制到目标表后，数据库复制系统开始复制 T1-T4 所包括的变更。可以容易发现，T1-T3 在 T4 开始之前提交，其变更后的数据行已经经过初始化过程复制到目标表，所以如果直接复制 T1-T3 的数据变更到目标表，我们会发现一些特别情况，即需要复制的行可能已经在目标表中，或者需要删除的行可能已经不在目标表中，或者需更新的行可能已经是更新后的状态。

在正常复制的过程中，上述情况皆为异常情况。但是在初始化这个特殊环节，我们对这些异常情况进行放行处理，即对 T1-T3 进行容错复制。事务 T4 本身并不包括对表 T 的更新操作，但是数据库复制系统利用这个事务作为一个标记信号，即从 T4 之后的对表 T 的日志记录都需要经过正常复制流程进行复制，即表 T 初始化完成。在这里需要指出一种特别的情况，即如果 T1-T4 尚未被处理，数据库复制系统就遇到了 T5，则数据复制数据流必须暂停直至 T1-T4 处理结束[39][40]。

2.4 Fork-Join 并发编程框架简介

Fork Join 框架属于 Java SE 7 的新功能。由 Doug Lea 设计并开发，属于对 JSR-166 标准的扩展。其核心思想是，一个任务如果可以被分解为多个子任务，当每个子任务结束时，通过组合每个任务的结果就能获得应用的最终结果。在这里，一个任务是一段可以被线程执行的代码和数据的组合。一个应用可以分解为超过系统 CPU 个数的任务集合，Fork Join 通过其内置的 Fork Join 任务调度器将任务按照用户程序所指定的逻辑顺序分配到一组固定数目的工作线程上执行。

基于Fork Join的算法就是我们常见的并行化版本的的分治算法，其基本形式如下：

```
解决（问题） {
    如果 问题很小
        直接解决这个问题
    否则
        将问题分解为更小、互相独立的子问题
        fork子任务，分别解决这些子问题
        join子任务，等待所有的子任务已经完成
        收集、合并各个子任务的结果
}
```

本设计中将全面采用 Fork Join 框架建立数据复制数据流的并发模型。

2.5 本章小结

本章介绍了基于数据库日志(Database Transaction Log)的数据库复制的相关概念，原理以及技术。此外还简单介绍了 Fork Join 的框架，该框架被用于本次设计的并发处理提供了便利基础。

3 总体设计概述

本章描述了优化后的数据库复制数据流的总体设计。首先介绍了新的数据复制系统的软、硬件环境和目标应用场景，然后介绍了总体架构、组件、模块的划分及其之间的相互关系，最后对几个重要的组件的内部数据处理流程进行了详细介绍。

3.1 系统软、硬件环境说明

本数据库复制系统将运行于主流操作系统之上，例如 Redhat Linux Enterprise 6，Microsoft Windows Server 2008， IBM AIX6 等。管理员通过基于 WEB 的界面对复制系统进行管理和监控。源数据库支持 ORACLE 11/12， Sybase ASE 15， Microsoft SQLServer， IBM DB2 LUW。目标数据库支持所有前述数据库系统，并且支持 SAP HANA1.0 和 Sybase IQ 16 数据仓库。

本数据库复制系统的目标应用场景是数据集成，即 DI(Data Integration)。其目标是将位于同一局域网的不同源数据库的数据表有选择的复制到一个集中的数据库或者数据仓库，例如 SAP HANA 1.0 和 Sybase IQ16。

根据复制数据量的不同，复制服务器所需的硬件配置会有所不同，但是，在本次设计开发中，我们采用的服务器为：

硬件配置

CPU: Intel® Xeon® CPU X7350 @ 2.93GHz (超线程打开，共 4 核，8 线程)

内存：64G

硬盘：146GB x 4 / 15000 转 / SAS 接口

网卡：10GBPS LAN

软件环境

操作系统：Redhat Linux Enterprise V6.0

主数据库: Sybase Adaptive Server Enterprise 15.7.1 ESD1

从数据库：Sybase Adaptive Server Enterprise 15.7.1 ESD1

3.2 目标应用场景

如图 3-1 所示，本数据库复制服务器主要被应用于数据集成的场景内。数据集成是一个具有很宽泛概念的数据管理活动。其主要目的是收集将企业中分散在结构化和非结构化数据，并对其进行一定的加工处理的过程。对于数据库而言，主要就是集成来自不同数据库表格的数据以满足数据分析的需要。集成后的数据必须定期与所涉及到的源数据库的数据保持同步关系，为了获得实时性的功能要求，数据库复制系统分担了将不同源数据库中指定表格的变更数据复制到目标数据库中的工作。

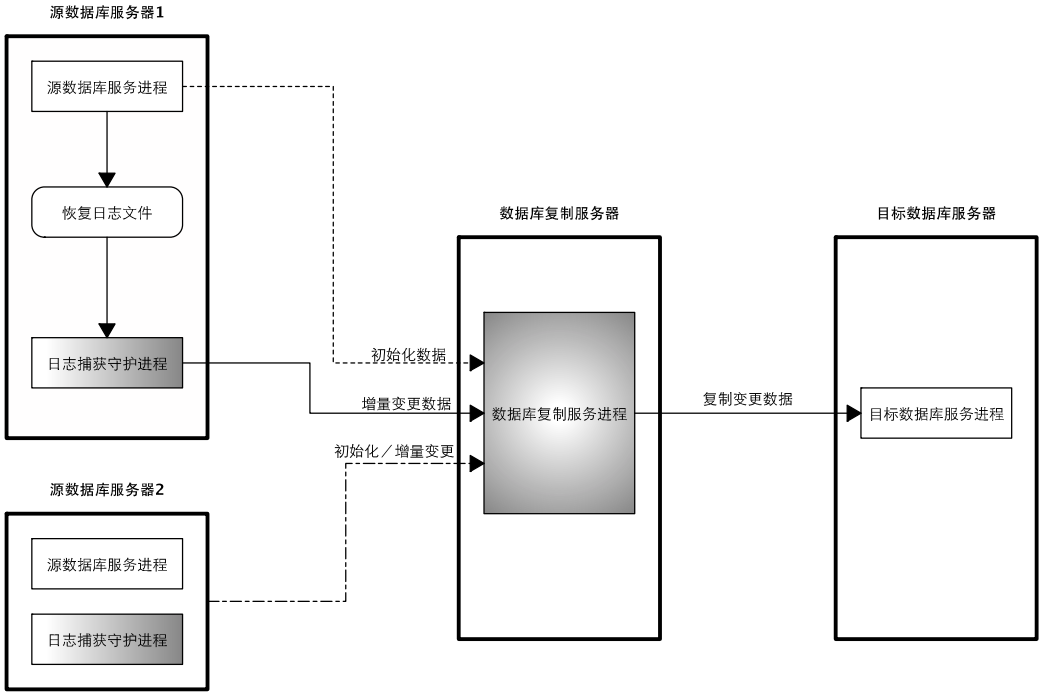

图 3-1 在数据集成场景下的数据库复制

Fig. 3-1 Database replication in the context of data integration

为了实现实时的数据集成过程，硬件上必须满足如下条件：

（1） 源数据库、复制服务器、目标数据库服务器之间的网络必须高速、稳定；这是实现数据流通的基本条件之一。

（2） 复制服务器必须独占单独的服务器硬件，并具有较高的配置以满足同时处理来自

不同源数据库的数据变更的能力：例如更多 CPU，更多内存；

3.3 总体流程概述

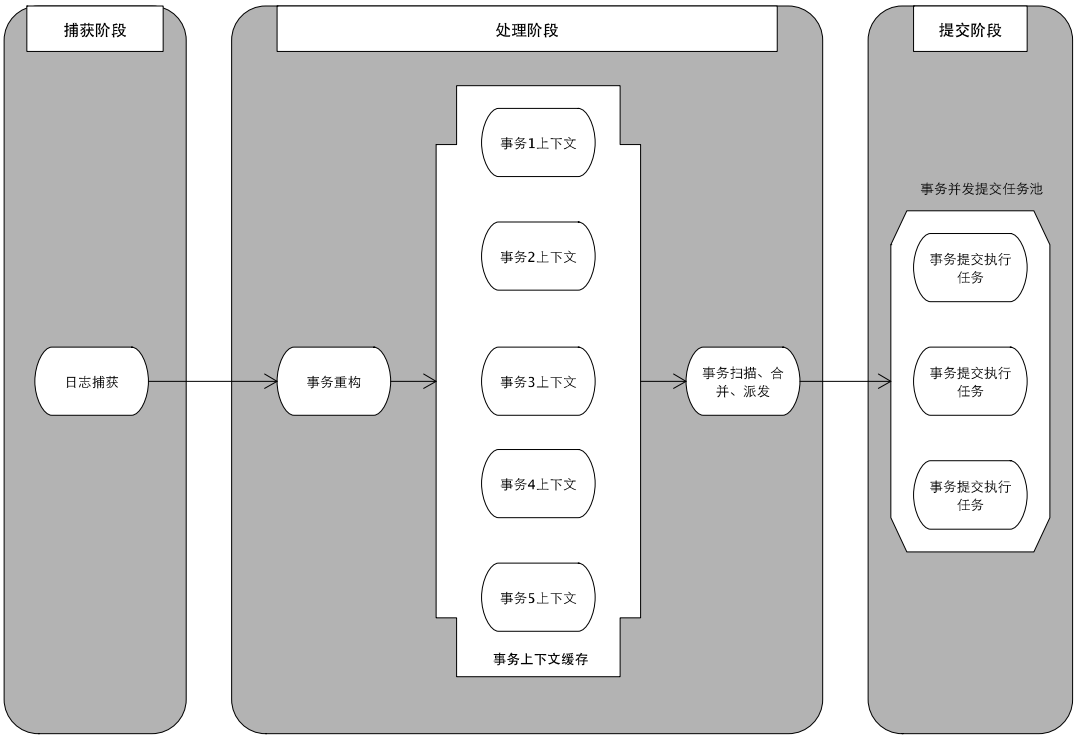

图 3-2 总体流程图

Fig. 3-2 Overview of new replication processing steps diagram

由图 3-2 可以看出，新的数据复制流程被分解为捕获阶段、事务处理阶段和事务提交阶段[41][42]。

3.4 日志抓取阶段

日志抓取的主要任务是负责从数据库恢复日志文件中或者相应接口获取数据库的原始变更日志并发送给日志处理阶段。由于每一个数据库关系系统的日志格式都互不相同，因此日志抓取会将每一条对复制有意义的日志记录翻译成中间日志记录，即数据库复制系统同一定义的一套中间日志记录的标准数据格式。

日志抓取主要由三个步骤组成：原始日志读取、原始日志解析转换、日志发送[43]

。如图 3-3 所示：

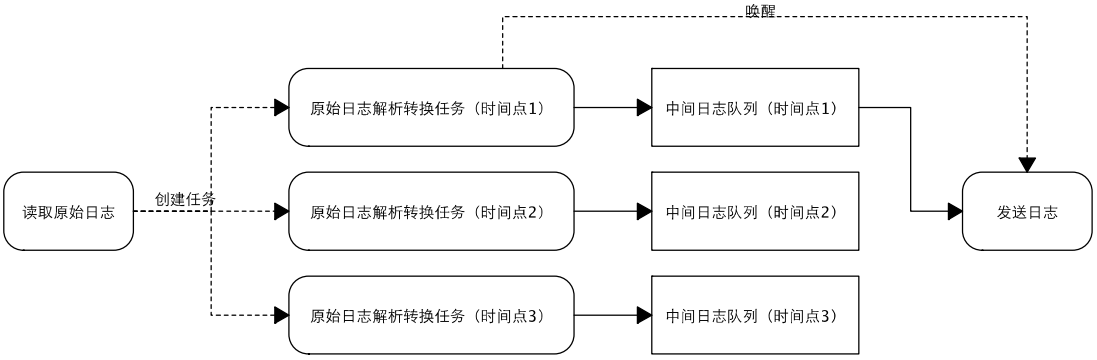

图 3-3 日志抓取进程内部流程

Fig. 3-3 Raw database log record capturing processing

原始日志读取的主要功能是获取数据库恢复日志的原始日志记录信息，并将读取到的原始日志记录分配给原始日志解析转换任务，提交任务到 ForkJoin Thread Pool。

该阶段主要将尚未处理的原始日志记录按顺序读取出来。读取的方式有：

- 对 IBM DB2 数据库，直接调用其日志读取管理 API；
- 对 ORACLE 数据库而言，直接调用其 XStream API 或者 LogMiner 存储过程；
- 对 SYBASE ASE 或者 MSSQL 而言，直接读取其数据库的日志设备或者文件

日志读取的处理步骤伪代码如下：

```
const int READBATCHSIZE = 1000;   // 每次读取日志记录个数上限，用户可以配置
lastLRSeq = getEndOfLogSeq();          // 获取恢复日志中最后一条日志的序号
while (!exit)
{
  if (LRSeq <= LastLRSeq)   // 如果当前还有尚未读取的日志记录
  {
      // 读取日志为 LRSeq 的原始日志记录
      rawLogRecs = readLogRecords(LRSeq， lastLRSeq， READBATCHSIZE);
```

```
            // 提交原始日志解析转换任务，仅处理 rawLogRecs 中的原始日志记录
            submitLogTranslateTask(rawLogRecs);
            // 将 rawLogRecs 入发送队列 sendQueue
            sendQueue.push(rawLogRecs);
        } else
        {
            sleep; //等待 1 秒
        }
    }
```

通过伪代码可以看出，每一个原始日志解析转换任务只会处理一定数量的原始日志记录。并且所有读取的原始日志记录都会存入发送队列(sendQueue)中，利用这个队列我们可以维护日志记录之间的先后次序[44]。

原始日志解析转换任务的主要功能是负责将原始日志记录转换为中间日志格式，并过滤大量与数据库复制无关的原始日志记录。数据库的日志类型会有超过几十种，但是只有如下的日志记录类型才能用于数据复制目的。

- 事务开始
- 事务结束
- 操作回滚
- INSERT 一行数据
- DELETE 一行数据
- UPDATE 一行数据
- INSERT 一块大字段数据
- TRUNCATE 某张表

对上述日志类型，我们仅需要如下日志信息：

（1） 所有日志的通用信息

- 日志的原始日志序号地址
- 事务序号

（2） DML 日志的信息

- 所影响的表对象序号
- 对于 INSERT 操作，需要给出被插入的数据行信息
- 对于 DELETE 操作，需要给出被删除的数据行信息
- 对于 UPDATE 操作，需要给出被更新数据行在更新前后的信息
- 对于 TRUNCATE 操作，则无需额外信息，仅需要表对象序号即可

3） Transaction 日志的信息

- 事务开始、结束时的时间戳，用于进行事务排序
- 对于多条日志回滚，需要知道被回滚的日志的序号范围
- 对于单挑日志回滚，无需知道额外信息

由于每一个数据库系统的原始日志格式、日志种类完全不同，即使是同一数据库系统的不同版本之间的日志格式也会有所不同。为了将上述不同日志的种类和信息封装起来，本设计使用了面向对象的设计办法将这种差异体现在日志记录类的继承体系中。图 3-4 的类图体现了 IBM DB2 的日志记录的部分类图[45]。

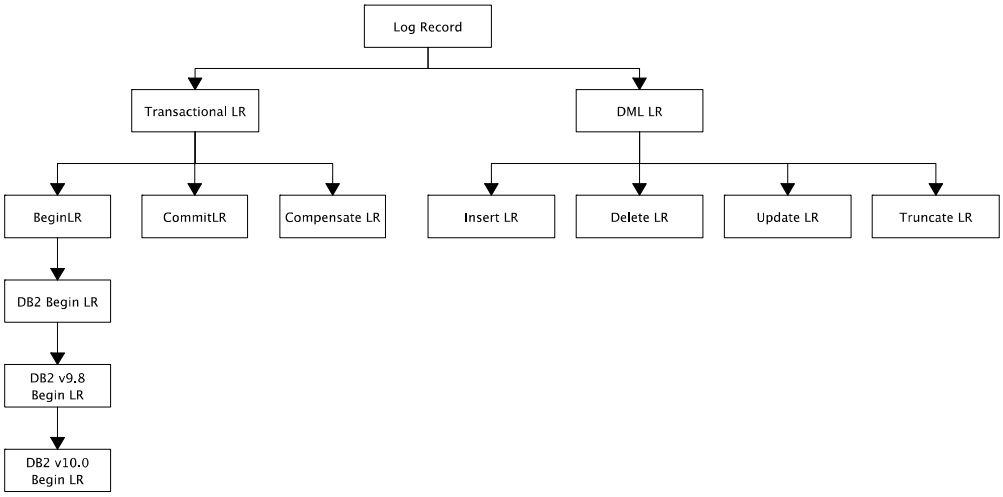

图 3-4 利用面向对象技术封装原始日志的异同的类图

Fig.3-4 Log Record Class Diagram

Log Record 类定义了所有日志记录的通用信息访问接口，例如日志记录序号、事

务序号、日志类型、长度等。Transactional LR 类定义了所有与事务相关的通用信息访问接口，主要是事务开始、结束的时间戳信息等。DML LR 定义了与数据变更相关的信息访问接口，例如获取数据变更所涉及的表对象序号等；Insert LR， Delete LR，Update LR， 定义了获取数据变更所影响的数据行的获取接口。Truncate LR 代表截断表数据的操作。虽然所有数据库的日志记录格式不同，但是对于复制相关的日志记录都可以归结到图 3-4 所示的日志记录类中。即便是同一数据库系统的不同版本所产生的日志差异也可以体现在图 3-4 所示的类结构中，例如对于 BeginLR 由针对 DB2 v9.8 和 v10.0 两个版本的具体实现。因此，这个设计可以真正实现面向对象的开闭原则，即允许扩展性修改，不允许修改现有类的实现。

由于所有的日志解析转换流程都已经封装在了具体的 LogRecord 的具体类中，因此原始日志记录解析转换任务的处理步骤非常简单，下面是其实现步骤的伪代码：

```
void LogRecordTranslatorTask(RawLogRecord[] rawLogRecords)
{
   foreach lr in rawLogRecords
   {
        // DB2LogRecordFactory 负责根据原始日志的类型，创建对应的中间日志记录对象
        // 如果日志记录不需要用于负责，则直接返回 NULL
        LogRecord convertedLR = DB2LogRecordFactory.create(rawLogRecords);
        If (convertedLR == null)
        {
             // 将原始日志记录标记为已过滤
             rawLogRecords.setFiltered();
             continue; // 过滤不需要的日志记录
        } else
        {
             // 将转换好的日志记录附加在原始日志记录对象中
             rawLogRecords.setResult(convertedLR);
             senderTask.notify();
        }
   }
}
```

此外，由于全部的转换任务之间不需要共享任何数据结构，因此可以允许多个原始日志解析转换任务同时并发转换读到的原始日志记录，这样的设计可以提高转换的速度。

日志抓取阶段的最后步骤是日志发送，即将转换好的中间日志记录发送给数据库复制系统。每一个原始日志记录都有 3 个状态，即：

- 未处理状态：表示这个原始日志记录尚未被原始日志解析转换任务处理；
- 已处理，需过滤：表示这个原始日志记录已经被处理，并且认定需要被过滤；
- 已处理，需发送：表示这个原始日志记录已经被处理，并且认定需要发送到复制系统，需要发送的人中间日志记录存放在这个原始日志记录对象中。

3.5 日志抓取进程与数据库复制系统的通讯命令协议设计

日志抓取进程与数据库复制系统之间需要一定的通讯协议和约定以便完成日志记录的正确传送，设计了如下通讯原语和行为协议：

- 日志抓取进程启动后将主动与数据库复制系统建立 TCPIP 协议的通讯链接，并首先发送如下命令给复制系统：

LOGREADER CONNECT. [PDS].[PDB] VERSION[nnn] [PERSIST | MEMORY]

在这个命令中，LOGREADER CONNECT.是用于向数据库复制系统表明当前连接的客户是一个数据库日志抓取程序。PDS.PDB 是用于向数据库复制系统表明日志抓取程序正在抓取哪一个数据库的恢复日志。VERSION[nnn]用于向数据库复制系统表明日志抓取程序的版本编号。PERSIST 和 MEMORY 是用于指示数据库复制系统是否需要将中间日志保存至入站磁盘队列或者入站内存队列中。需要指出的是，任何原因的连接中断恢复后，此命令一定会被作为第一条通讯命令发送给数据库复制系统进程。

- 数据库复制系统向日志抓取程序返回初始日志抓取点信息。

当数据库复制系统确认接入的日志抓取程序可以提供已经注册的数据库的恢复日志的抓取服务时，其将会查询这个数据库所对应的初始日志抓取点信息。我们也称这个初始日志抓取为第 2 数据库日志截断点。如果抓取程序是第一次连接到数据库复制系统则直接返回 0 作为抓取点，这将指示日志抓取程序直接从当前数据库最后的一条恢复日志开始抓取，否则，抓取程序将尝试从指定的日志抓取点开始抓取日志。日志抓取点的计算方法将在后续继续深入介绍。

- 传送中间日志记录，即将一条中间日志记录传送到复制系统。
- 传送当前日志抓取点。

在日志传送的过程中，日志抓取程序将定时向数据库复制系统询问当前的初始日志抓取点以便设定源数据库系统的第 2 日志截断点。

数据库的日志存储的空间是有限的。根据应用场景的不同，数据库管理员会配置足够大的日志存储空间用于数据库的恢复日志的保存。在没有数据库复制系统的情况下，每当数据库管理员执行了备份操作，则备份前所产生的恢复日志都可以被删除以便空出更多的日志存储空间，而截断点位于当前所有活动事务的最老的一个事务的开始事务(Begin Transaction)日志记录所在位置，设定在我们称这个点为数据库日志截断点。引入数据库复制系统后，为了保证尚未复制的日志文件不被删除，数据库系统引入了第 2 截断点的设计，这个点的设定完全由数据库复制系统决定。数据库会保证第 2 日志截断点之后的日志记录文件不会被删除。第 2 日志截断点由日志抓取程序定期向数据库复制系统询问，而日志截断点的位置为最后一个复制到目标数据库的事务的开始事务日志所在位置。可以看出，在复制的过程中，为了保证数据库的日志存储设备不被占满，必须尽可能快的移动第 2 日志截断点。否则源数据库系统的日志空间可能会由于无法删除日志文件而占满。

3.6 入站磁盘队列和入站内存队列

数据库的日志存储空间是有限的，数据库管理员需要定期将已经归档的日志文件移出已释放磁盘空间，但是为了满足数据复制的需要，数据库只允许删除已经复制到目标数据库的数据变更所对应日志文件。此时会遇到一个问题，即如果目标数据库暂时停机，或者由于某些原因导致数据变更写入目标数据库的速度跟不上源数据库产生日志变更的速度，日志文件会不断在源数据库系统堆积，最终导致日志空间磁盘空间占满而影响正常的产生作业。

为了最大可能的减少由于复制性能延时导致的源数据库系统的日志堆积，在数据库复制服务器上设置了一个入站磁盘队列。当日志堆积确实较为严重时，我们可以将接收到的日志记录全部存入入站磁盘队列，由于入站队列的写入已顺序写入为主，且磁盘的顺序写入性能较高（例如，一个普通硬盘的连续写入速度通常在 100MB/s 以上），当日志记录落实进入磁盘队列后，可以认为该日志已经复制（或最终会复制）到目标数据库系统，源数据库系统不必等待日志记录复制、落实到目标数据库后才对日志

文件进行移除操作。根据复制系统的规模的不同，入站磁盘队列的尺寸可以是几十个 GB 甚至到 TB 级别。引入了入站磁盘队列后，所有的日志记录都会经过磁盘写入和磁盘读出这 2 个额外的处理步骤，对单条日志的复制延时而言是有所增长的，但是入站磁盘队列的引入将日志堆积在数据库复制系统中的设计可以保护源数据库复制系统的正常运作。

但是在更多的情况下数据复制的性能是能够跟上源数据库复制系统产生日志记录的速度的，因此，入站队列也可以是基于内存的队列，从而避免了磁盘读写操作。在缺省情况下，所有中间日志都只会保存在入站内存队列中直致被复制到目标数据库中。当日志抓取程序发现所抓取的数据库系统的空闲日志存储空间低于某一个阀值时，将会主动断开与复制系统的连接，并重新连接，在重新建立连接时将通过 LOGREADER CONNECT PERSIST 命令指示数据库复制进程使用入站磁盘队列保存中间日志。在这里存在一个假设，即将中间日志写入磁盘的速度比将其所代表的变更写入磁盘的速度要更快。这样就可以允许数据库复制服务器更快的移动第 2 日志截断点从而允许数据库管理员对日志进行有效截断以释放日志磁盘空间。当源数据库的日志存储空间恢复到阀值以内后，日志抓取进程将主动断开并重建与复制系统的连接，并指示 LOGREADER CONNECT MEMORY 命令指示数据库复制系统使用内存队列。

图 3-5 展示了入站队列的两种工作模式。

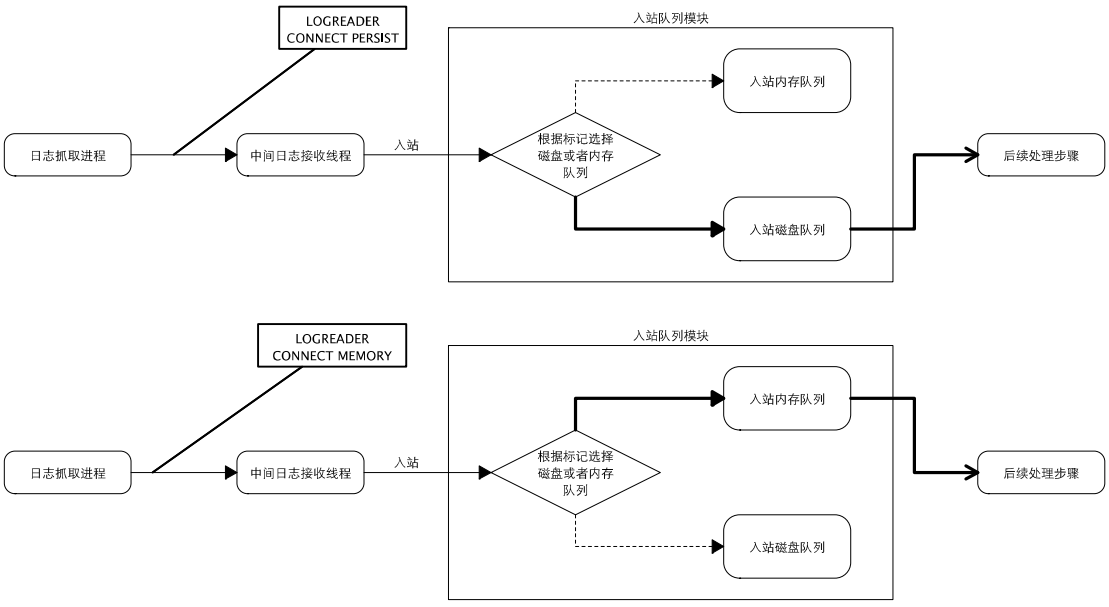

图 3-5 入站队列的 PERSIST 模式和 MEMORY 模式的切换

Fig.3-5 Inbound Queue Persist Mode and Memory Mode

3.7 选择需要复制的表对象

本步骤主要目的是过滤出用户期望复制的表对象的变更日志。在一个生产系统中，数据库中的表个数量成千上万，用户一般并不期望复制全部的数据表，因此需要过滤掉掉那些并不需要要被复制的表的中间日志记录。

数据库复制系统维护了一个需要被复制的表对象的复制状态信息。这个状态信息由用户根据实际需要进行配置。由于用户可能在不同的时刻提出需要复制某张表，再另外一个时刻又提出不需要复制同一张表，为了跟踪这些状态的变更，引入了版本化的数据库对象复制状态的设计，通过这个设计，可以保证在处理同一条日志时使用相同的复制判断状态，具体内容见 4.1 节。

3.8 事务排序

事务排序是重要的复制准备步骤之一，其目标是将中间日志按照事务为单位合并，并识别出事务在源数据库提交的先后次序。事务排序的原因是由于，不同事务的变更日志是交错的方式写入源数据库日志文件中的，而我们必须以事务为单位进行变更复制并且保证这些事务在目标数据库的提交的顺序与源数据库保持一致[16][17]，因此必须找到有哪些事务需要复制，每一个事务包括了哪些日志变更记录，以及每一个事务提交时间的前后次序。

事务排序基本方法如下：数据结构包括一个哈希映射表，主键是日志对应的事务序号，值是一个指向目标事务对象的指针。一个开放事务列表，用于保存全部尚未发现提交命令的事务对象的列表，该列表中的事务对象出现的次序和事务在源数据库的开始时间次序一致。一个已关闭事务列表，用于保存已经发现事务提交命令的事务对象列表，该列表中的事务对象的出现次序和事务在源数据库的提交时间次序一致。当遇到事务开始日志时，创建一个以该事务的事务序号为主键的事务对象，并存入映射表和开放事务列表。当遇到一个事务结束日志时，则将该日志记录中的事务序号所对应的事务对象从开放事务列表移动到已关闭事务列表中。

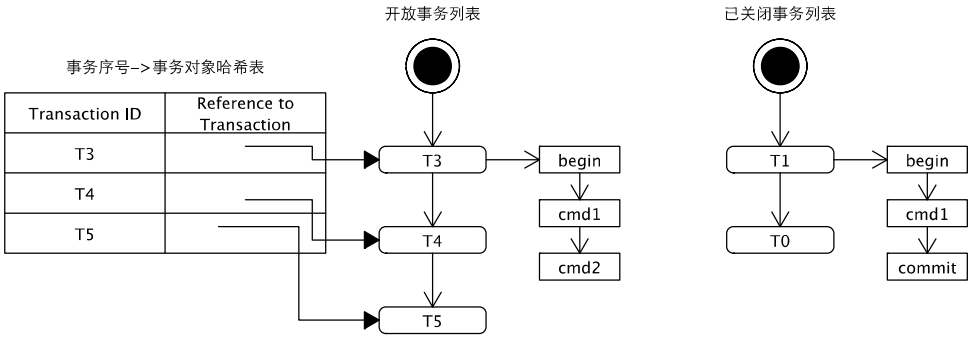

图 3-6 事务排序原理

Fig.3-6 Transaction Sorting

图 3-6 所示给出了一个数据库复制系统在进行事务排序时的运行时状态，当前有 3 个开放事务：T3， T4， T5，有 2 个已关闭事务：T0，T1。当下一个日志命令到来时，假设是 T3 事务的提交命令，此时会首先从［事务序号→事务对象哈希表］中，找到事务 T3 所对应的事务对象。事务对象 T3 中目前已经有了 3 条命令的列表，即 begin，cmd1 和 cmd2，当前提交命令会直接加入到这个列表的尾部。此时，由于事务 T3 已经遇到了对应的提交命令，意味着这个事务所对应的日志记录已经全部接收到，此时 T3 变为已关闭事务，其对象节点从开放事务列表中移动到已关闭事务列表的尾部，即 T0 之后。T3 事务所对应的在［事务序号→事务对象哈希表］中的节点也被移除。经过这个步骤后的运行时状态见图 3-7 所示。

为了向后续复制任务提供更多的帮助信息，在事务排序的过程中还会在排序的过程中对每一个事务的特征信息进行统计。例如一个事务所涉及到的表对象的数据字典的信息，由于每一个表的结构在复制过程中可能会被改变，因此必须寻找到日志记录当时所使用的表对象的数据字典信息，这就设计到了版本化管理数据库对象结构信息的设计，详见 4.1 节。此外，还需要统计每一个事务的增加、删除、更新操作的总数量，以及这些操作在每张表的数量分配等信息。关于事务特征信息的描述放在了第 4.2 节。

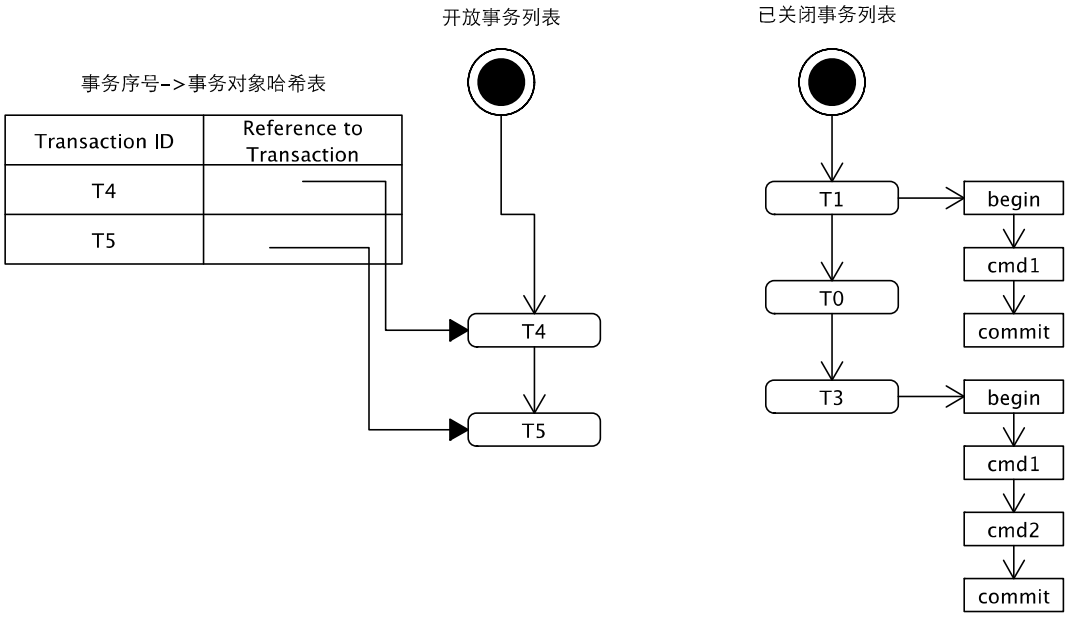

图 3-7 事务排序，插入 T3 事务的提交命令后的运行时状态

Fig.3-7 Transaction Sorting， after inserted T3's commit log record

3.9 事务复制、提交阶段

当一个事务被插入已关闭事务列表后，就可以开始将其落实到目标数据库中。总体而言，事务复制分为两个大步骤，即事务复制步骤和事务提交步骤。事务复制步骤是以尽可能快的速度将一个事务所包括的变更信息以适当的形式和接口（例如 SQL 语句）发送到目标数据库中，事务提交步骤是当一个事务包括的全部变更都写入目标数据库后，执行事务提交命令，以落实事务。

为了提高事务复制的性能，在事务复制阶段可以做各种研究和优化工作，为了尽可能利用目标数据库系统的并发出力能，不同的事务会按照一定的策略分配给不同的事务提交任务同时执行写入目标数据库的操作，在本论文第 4 章中描述了一些关于如何在事务复制阶段提高事务写入数据库性能的优化工作。这也是本论文研究的主要目的之一。

事务提交阶段必须保证所执行的提交命令的次序和这些事务在源数据库的提交顺序保持一致。只有这样才能保证目标数据库和源数据库保持事务的一致性。

3.10 日志抓取点和避免事务的重复提交

由于每当日志抓取程序重新连接复制系统时，将从数据库复制系统获取日志抓取点以便确定从哪一点开始重新抓取日志记录。

日志抓取点的确定分为两种情况：

- 当入站磁盘日志队列中存在日志记录时，日志抓取点对应入站磁盘队列中最后一条日志记录的日志序号。
- 当入站磁盘队列中不存在任何日志记录中，日志抓取点对应一个事务开始日志命令的日志记录序号，在复制系统复制的过程中，即对应事务排序步骤中开放事务列表中第一个事务所对应的开始事务日志的日志序号。日志抓取点信息将会保存于复制系统的内置数据库中。理想情况下，每当有事务提交后，都需要保存其所对应的第一个开放事务所对应的开始事务日志的日志序号。但是这就相当于在所有事务的提交步骤后插入了另外一个磁盘操作，势必会对复制性能产生影响。因此在本次设计中，采用了定时更新日志抓取点的设计。

由于采用了定时更新日志抓取点的办法，因此当日志抓取程序重新从日志抓取点发送日志信息时，可能会包括一部分已经成功写入目标数据库的事务集合。为了避免重复提交事务情况的发生，设计如下：

在目标数据库中建立一个表格，用于保存最后一次提交的事务的提交命令的日志序号。在事务提交任务执行提交命令之前，更新这个表格中的提交命令的日志序号的值。

当数据库复制系统重启后，会读取这个日志序号，并始终在内存中维护一个最后成功提交到目标数据的事务的提交命令的日志序号，简称最后提交事务日志序号。由于事务排序的存在，我们保证提交到目标数据库的事务是按照事务提交命令的日志序号的大小为顺序，因此，当事务排序发现的一个已关闭事务的提交命令的日志序号小于或等于最后提交事务日志序号时，则可以直接跳过这个事务的复制过程，从而避免了事务在目标数据库的重复提交。

3.11 本章小结

本章主要描述了基于数据库事务日志复制的基本流程。基本可以概括为事务日志抓取阶段、事务排序阶段和复制提交阶段。在不同阶段中，采用了基于磁盘的队列保存事务日志的内容。在某些对性能要求极高的场景下，也可以采用基于内存的队列来保存事务日志内容。针对每个流程步骤，本章还描述了其基本实现原理，比如如何进

行事务排序，如何避免事务重复提交。此外，对数据库事务日志的大致逻辑结构进行了简单介绍。

4 关键技术的设计和实现

本章的主要内容是详细介绍提高复制性能的相关设计和算法，介绍了一版本化管理复制对象状态和结构的办法，并由此给出了数据库结构变更的复制办法。通过并发复制不同事务来提高事务复制，描述了一种安全执行并发事务的设计，展示了一种并发提交事务内变更的办法。此外，研究了通过减少提交命令的频率，来提高事务复制性能的方法。

4.1 版本化管理数据库对象复制状态以及结构信息

数据库对象的复制状态表示一个数据库对象是否需要被复制。结构信息即数据库对象的数据字典的信息，包括数据库对象的模式名、表名，各个字段的名称、数据类型、长度、精度、可否为空的定义信息，以及这个表的主键构成信息。在事务复制的过程中的很多步骤，都需要获得数据库对象的复制状态信息以及结构信息。例如在数据库复制系统需要根据复制状态信息决定是否需要将一个数据库对象的变更发送到目标数据库中。也需要表结构信息获得表字段的名称和主键以便产生正确版本的 SQL 语句。一个数据变更日志记录必须对应一个唯一确定的数据对象，以及它的复制状态和结构信息。数据库允许用户更新数据库对象的结构，数据库复制系统也允许用户改变数据库对象的复制状态。理想情况下，所有这些信息都能通过数据库的数据字典查询得到，但是由于数据库变更日志写入恢复日志中的时刻和数据库复制系统处理这条日志的时刻存在一个间隙（可能很长，也可能很短），在这个间隙中，用户有可能更新了这个数据库对象的结构或者复制状态，如果此时还是通过查询数据库字典的办法获取则无法获取争取的结构或者复制状态的信息。

图 4-1 展示了某表 A 的结构变化对日志结构的影响。在时间点 T2，用户在表 A 中增加了一个新的字段 C3，对比 T1 和 T2 这两个时间点可以看出，日志结构是不同的：在日志 3 中，包括了 C3 字段的插入的值，而日志 1 中则没有。如果数据库复制系统开始处理日志 1 的时刻晚于 T3，则此时源数据库数据字典中的表 A 的结构已经是 T3 时刻的结构，那么数据库复制系统会发现日志 1 中并没有 C3 字段的值，但这实际上是由于日志与表结构信息已经不匹配了。

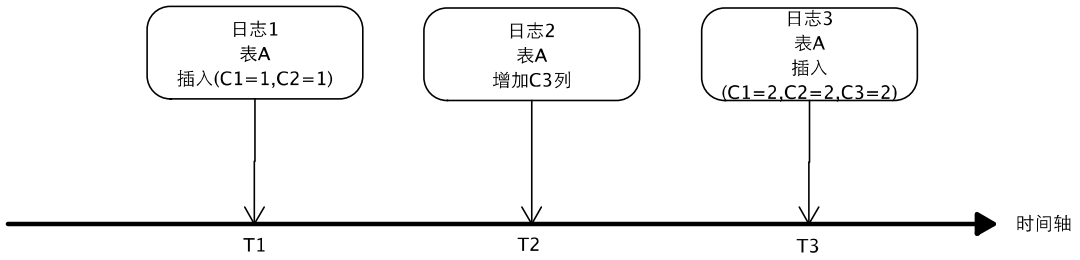

图 4-1 表结构变更对日志记录结构的影响

Fig.4-1 Table definition changes impact log record structure

为了解决这个问题，采用了版本化的方式管理数据库对象的复制状态以及结构信息。即保存所有的数据库对象的复制状态和结构的变更历史，每一个复制状态的变更、每一个结构变更，都被赋予一个版本编号。由于所有的变更本质上就是一个事务变更，因此我们采用事务变更的序号（即事务的提交命令的日志序号）作为版本编号的来源。图 4-2 展示了版本化管理表 A 结构的一个例子。在 T2 时刻，由于日志 2 表示表 A 增加了一个新的字段 C3，因此将建立一个新版本的表 A 结构，版本编号为日志 2 的日志序号（假设为 T2）。当数据库复制系统在处理日志 3 时，将获取一个版本编号小于 T3，并且距离 T3 最接近的表 A 结构信息。

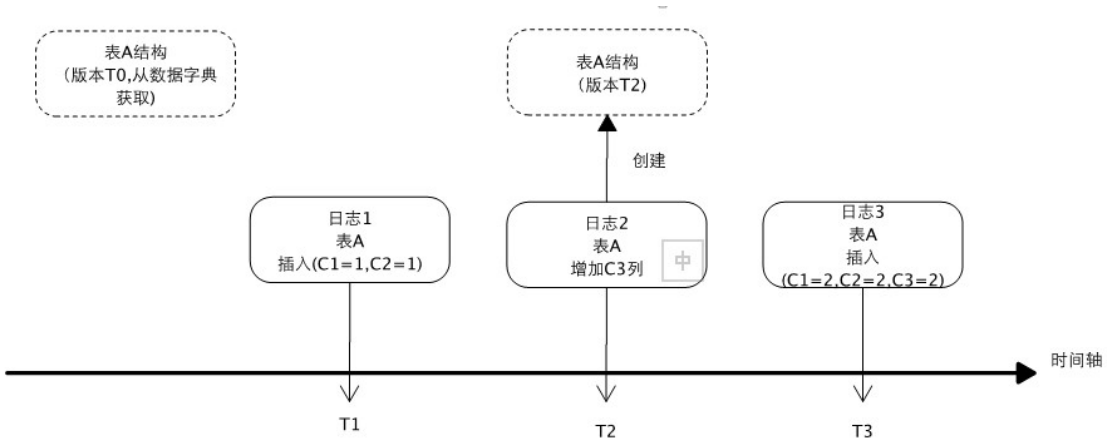

图 4-2 表结构变更对日志记录结构的影响

Fig.4-2 Table definition changes impact log record structure

为了保存这些版本化的表对象复制状态和表结构信息，将这些信息全部保存在数据库复制系统的管理数据库中的复制对象变更历史表中。该表包括 3 个字段，即表对象 ID，该字段即一个表对象在源数据库的序号，当一个表在源数据被创建后，这个序号将一直存在，在一般情况下表对象 ID 时不能被重用的；版本编号字段，即某一个版本的表结构信息生效时的日志序号；表结构，即一个表的复制状态和结构信息的综合体。

当一个数据变更日志进入数据库复制系统后，根据日志中的表对象 ID 和该数据变更日志的日志序号，可以从复制对象变更历史表中找到对应的表结构信息。但是如果每一条数据变更日志都会触发一次数据库查询操作，那么会对复制性能有较大的影响，因此在内存中维护了一组最近经常使用的表对象的变更历史信息。这个内存结构包括 1 个入口和 3 个节点，即表结构对象版本缓存入口，这是一个哈希表，保存了从一个表对象 ID 到其对应的缓存结构对象的指针。该指针实际上首先指向了一个包含该表的最新版本的缓存节点，这个缓存节点包括另外一个指针，指向了另外一个历史版本缓存节点，这个节点保存了该表对象的全部的历史版本变更缓存。图 4-3 展示了内存中和数据库中的表对象版本信息的数据结构。

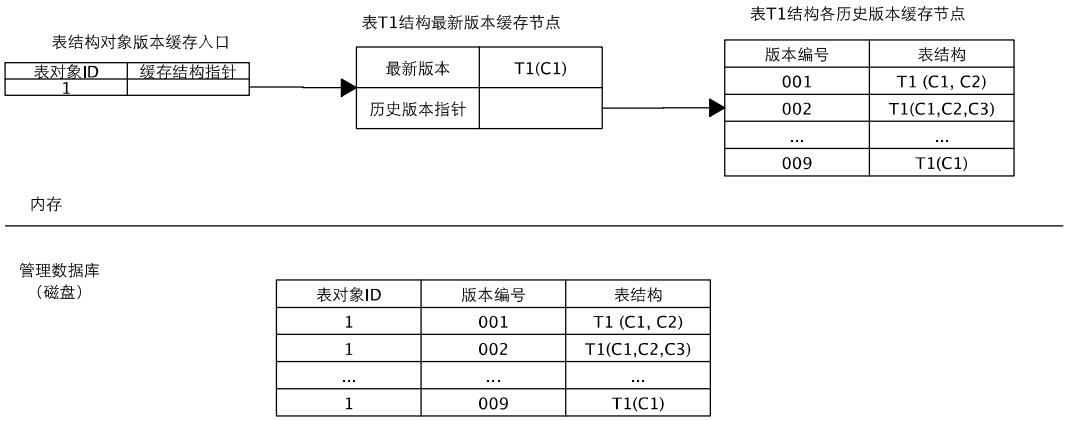

图 4-3 管理复制对象的版本信息

Fig.4-3 Managing multiple versions of table T1's table schemas

基于这上述设计，查询某一个表对象信息的版本的操作变为（伪代码）：

```
// 查询一个表对象的复制状态和结构信息
// table ID: 表对象序号
```

```
// lrSeq: 日志序号
TblSchemaObject queryGlobalTblSchema(tableID， lrSeq) {
// 首先查询版本对象缓存节点，确认版本信息确实在内存中
cachedNode = schemaCache.get(tableID);
    if (cachednode != null) {
        // 如果最新版本的表对象信息适用，则直接返回该对象信息
        if (lrSeq > cachednode.lastestVersion) {
            return cachednode.tblSchemaObject;
        } else {
            // 否则，顺序搜索所有该表对象的历史变更版本，直至找到
            return findInAllHistory(cachednode.allHistory， lrSeq);
        }
    } else {
        // 如果不在内存中，则从复制系统的系统数据库中读取
        return queryFromReplicationSchemaDB(tableID);
    }
}
```

为了提高内存利用效率，采用了 LRU(最少使用即淘汰)的算法确保内存中仅仅保存最常用的 N 个表的历史版本信息。每当一个表对象被正常查询返回，其使用计数器加 1。当一个表对象的历史变更信息从复制系统的系统数据库中读取被放入内存缓存结构中后，将会确保缓存中最多直保存最常用的 N 该表的历史版本信息，其他的表的版本信息将被全部从内存中删除。

此外，为了防止某一个表所包括的历史版本信息过多而造成的内存资和磁盘源浪费，每当数据库的第 2 截断点移动时，都会将版本编号小于第 2 阶段点所在日志序号表结构对象从内存和数据库中删除。这样的设计是安全的，这是由于所有在第 2 截断点前提交的事务已经全部写入并落实在目标数据库中，即所有版本编号小于第 2 截断点的表对象信息版本都不会被继续使用（除了小于并最接近第 2 截断点的那个版本）。

所有表结构的变更和其他数据类变更一样属于一个数据库事务。因此一个新版本的表结构信息必须在所在数据库事务提交时刻才能保存到数据库中。但是，有些数据库系统支持所谓在事务内的表结构变更操作（DDL in transaction），即在同一个事务里

，用户可以对一个表进行结构变更，并继续对该表进行数据变更。对于这种即包括表结构变更又包括被变更的表的数据变更的事务，必须保证所有的表结构的变更版本在事务提交后才能被提交到复制对象版本数据库中，如果该事务最终被回滚则无需提交这些变更版本到复制对象版本数据库中。

基于上述设计要求，我们将表复制状态和结构信息对象分割为两个空间，即事务内对象版本空间和全局对象版本空间，在事务内对象版本空间内保存了某一个尚未提交的事务的表对象的版本信息对象，而全局对象版本空间内则保存了全部已经提交的表对象的版本信息。基于这个设计，为了查询某张表的信息对象的正确版本，设计了如下查询过程（伪代码）：

```
// 查询一个表对象的复制状态和结构信息
// table ID: 表对象序号
// lrSeq: 日志序号
// trSeq: 事务序号
// 返回 TblSchemaObject：表的复制状态和结构信息对象
TblSchemaObject getSchemaObject(tableID， lrSeq， trSeq) {
    // 首先查询所在事务内是否有可用的复制状态和结构信息对象的版本
    TblSchemaObject retObj = queryInTransactionTblSchema(tableID， lrSeq， trSeq);
    if (retObj == null) {
        // 如果所在事务内没有适用版本，则从全局对象版本空间继续查询
        retObj = queryGlobalTblSchema(tableID， lrSeq， trSeq);
    }
    return retObj;
}
```

4.2 连续事务的并发式提交的设计和实现

经过事务的排序后，所有的变更都被放入了对应的事务对象，而这些事务对象也按照其在源数据库系统提交的次序连接在了一起。下面就是将这些事务写入目标数据库中的同时保证事务的提交前后次序。当今的数据库是能够并发处理多个事务的，如果能够利用这个特点将变更事务并发的提交给目标数据库，我们可以充分利用目标数据库系统的并发处理能力从而提高复制的吞吐量，即所谓的变更事务的并发式提交。事务并发复制主要分为 3 个环节：

（1） 事务派发环节

事务派发由事务派发任务完成。将事务从等待队列中取出，指派一个空闲的复制任务，然后将其插入提交队列。由于事务在等待队列中是已排序状态，因此事务在提交队列中依然是已排序状态。当一个事务从等待队列放入提交队列后，其状态从等待复制状态转换为正在复制状态。

（2） 事务复制环节

事务复制环节由复制任务完成。可以有多个复制任务，每个复制任务仅负责对被分配的事务进行数据库变更复制，互相之间完全独立。复制任务不会执行该事务的提交命令，这是由于其并不知道该事务之前的事务是否已经提交。当一个事务的全部变更（除提交命令之外）全部落实到目标数据库后，其状态从正在复制状态转换为等待提交状态。与此同时，复制任务的状态也从工作状态转换为空闲状态，等待下一次的事务指派。

（3） 事务提交环节

事务提交环节由提交任务完成。提交任务从提交队列的头部取出一个状态为等待提交的事务，并执行提交动作。由于提交队列中的事务都已经是提交状态，因此提交动作也是按照事务提交的顺序执行。

4.2.1 保证并发提交的事务不互相死锁

顺序地将事务按照事务提交顺序复制到目标数据库，是肯定无法实现高性能的数据复制的。这是由于主数据库中的不同事务都是高度并发执行的，但是在恢复日志中却丢失了这些并发执行的信息。因此，如果并发提交两个或者多个连续事务是有可能发生死锁的[19]。例如，见图 4-4 所示，如果 2 个事务(T1，T2)都更新了同一张表格中的相同行，T2 必须等待 T1 执行提交后才能提交，但是 T1 在等待锁定表 T 中的行 X 且这个锁已经被 T2 持有，则 T1 和 T2 形成死锁状态。

面对事务的死锁，有两种设计思路去解决，即基于乐观的设计和基于悲观的设计。基于乐观的事务并发执行假设所有连续提交的事务间不太会发生死锁或者很少发生死锁，复制系统可以在大多数时间将连续的不同事务直接放在不同的数据库连接上进行执行。基于悲观的设计则假设所有连续提交的事务间很有可能发生死锁，必须提前识别事务死锁的条件并尽量避免。在现实的数据库复制情景中，这两种情况都有可

能发生，因此在新的复制系统中，这两种事务并发复制模式都被支持，通过一个参数让用户自主选择。

在基于乐观的事务并发复制的设计中，我们假设当前准备复制的事务不会与正在复制中的事务发生死锁，因此该事务被直接提交给目标数据库进行执行复制。但是复制系统将会以一定的时间间隔在目标数据库中查询死锁条件，一旦发生死锁，则会主动回滚所有正在提交中的事务，然后按次序执行这些被回滚的事务。这个设计会利用目标数据库提供的一些状态信息用于死锁条件的查询。

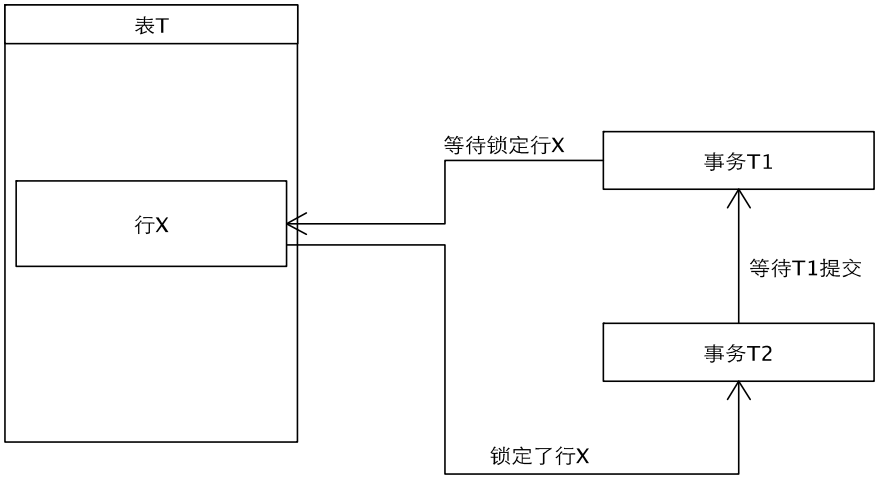

图 4-4 并发提交死锁的例子

Fig.4-4 A dead-lock situation

在基于悲观的事务并发复制的设计中，我们假设当前准备复制的事务有可能会与正在复制中的事务发生死锁。为了避免可能的死锁的情况，必须在派发该事务前，详细了解该事务的构成特征。为了保证两个连续事务 T1 和 T2 能够并发复制到目标数据库，其必须满足如下条件之一：

条件 1：确保 T1 所影响的数据表集合和 T2 所影响的没有交集；

条件 2：如果 T1 和 T2 都影响了相同的数据表，则需要确保 T1 和 T2 影响了这些数据表中的不同的数据行集合。

4.2.2 乐观的事务并发复制

并发的事务复制通过多个事务复制线程来实现。每一个事务复制线程都创建一个独立的到目标数据库的连接。每一个事务复制线程都有一个数字编号，该编号从 1 开始。待复制事务提交线程按照轮询的方式将待复制的事务分配给一个事务复制线程，假设有 4 个事务复制线程，则按照 1-2-3-4-1-2-3-4 待顺序分配。每一个事务复制线程最多仅复制一个事务，当一个事务复制线程尚未完成被分配的事务复制时是不能接受其他待复制事务的。

为了防止并发执行中的事务互相死锁，有一个单独的死锁检测线程用于检测并解决死锁。该线程被周期性唤醒以执行检测操作。检测的方法是通过查询目标数据库系统提供的用户回话信息。所有的主流关系行数据库管理系统都提供了对用户与数据库的会话信息的查询，会话信息都会被保存在一个系统表中，可以通过普通的 SQL 语句进行查询。每一个用户会话信息包括连接编号，是否正在执行事务，以及是否在等待其他连接的信息。其中是否在等待其他连接的信息可以用于帮助我们判断用于复制的数据库连接会话间是否发生了死锁。由于采用了轮询使用每一个数据库会话的方式提交事务，因此在未发生死锁的情况下，等待链应该是不存在环的，例如 4-->3-->2-->1 或者完全没有等待。一旦在等待链中发现了环，例如图 4-5 所示，连接 2 等待连接 3，并且连接 3 等待连接 2，则说明发生了死锁。一旦发生了死锁，必须对其进行解锁。在本次设计中，将会对所有活动的事务进行回滚操作，然后将这些事务按照次序通过同一个事务复制线程复制。

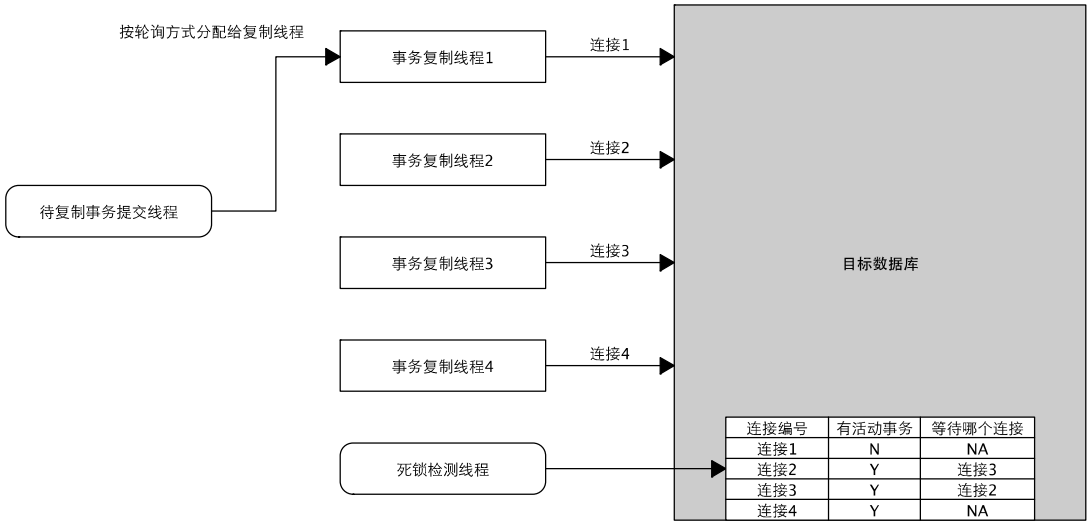

图 4-5 乐观的事务并发复制的实现

Fig.4-5 Parallel Transaction Apply， Pessimistic Mode

所有的事务复制线程只会复制除了提交命令之外的数据变更，这是由于复制线程并不知道其所复制的事务是否已经可以提交（为了保证事务提交顺序）。因此必须采用一定的设计保证事务提交顺序。其设计和实现见 4.2.4 节。

4.2.3 悲观的事务并发复制

悲观的事务并发复制假设事务之间很可能会发生死锁，因此通过计算确保提交给事务复制线程的事务间不会发生死锁。从实现角度而言，通过维护正在复制中和等待提交的事务的特征信息来确认等待提交的事务不会与正在复制中的事务产生死锁。

为了了解每一个事务所影响到的表的名称集合以及每张表中有哪些数据行受到了影响，在每一个事务所对应的上下文(Transaction Context)中保存了这个事务的特征信息(Transaction Profile)。事务特征信息的保存采用了哈希表的数据结构，分为两个层次：

第一层，这个层次可以用于支持某张表是否出现在该事务的查询。其哈希主键是数据表名称，值为数据行哈希表。

第二层，这个层次可以用于支持查询某行变更所影响到的数据行是否出现在该事务中。即数据行哈希表，其哈希主键是由被影响到的数据行的主键值计算出来的哈希值，值为一个顺序链表，该顺序链表中的元素为该事务中出现的所有影响到该数据行的日志记录。需要特别指出的是，对于更新日志操作，其会被分解为删除和插入两个日志操作，因此，一条更新日志相当于影响了 2 个数据行。

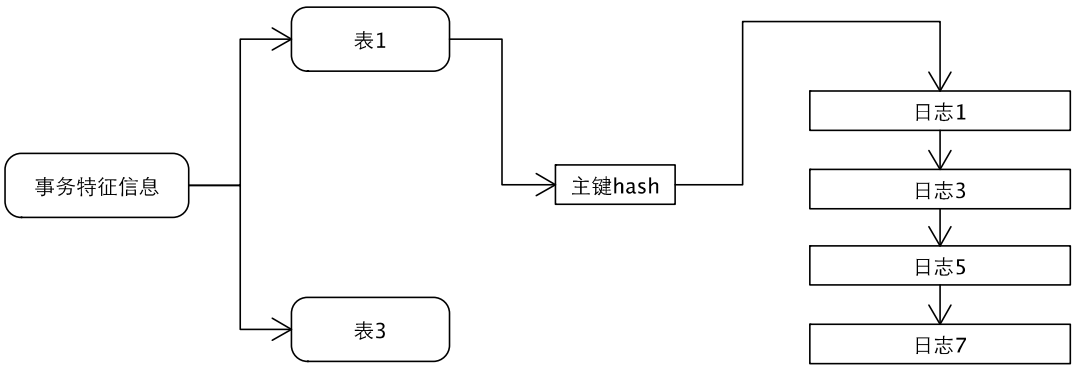

图 4-6 事务特征信息的数据结构

Fig. 4-6 Transaction Profile Data Structure

图 4-6 展示了一个事务特征信息数据结构实例。包括了 2 张表。表 1 中某一个行受到了 4 条变更日志的影响。

为了了解当前正在复制中的事务所影响的表和数据行的集合，需要维护一个建立在这些事务的特征信息之上的事务特征信息的集合，称之为活动事务特征信息(Active Transaction Profile)。如图 4-7 所示，活动事务特征信息是一个哈希表。其主键为表名，其值为所有当前正在复制的事务所影响到的该表的数据行哈希值的集合。

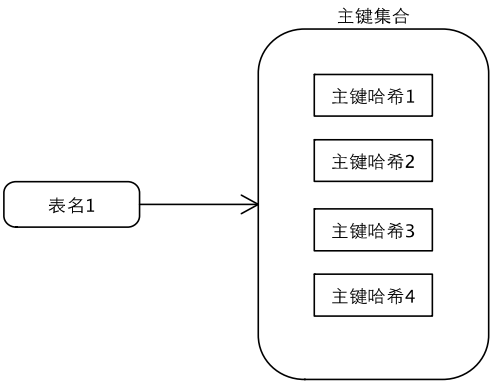

图 4-7 活动事务特征信息的数据结构

Fig.4-7 Active Transaction Profile Data Structure

当一个状态为等待复制的事务被派发给一个复制任务之前，其事务特征信息将会被用于和当前活动事务特征信息进行比对和合并：

步骤 1 确定等待复制的事务所影响的表是否出现在活动事务特征信息中。如果出现，则表示该事务可能会与现在正在复制中的事务形成死锁，进入步骤 2 进行近一步确认；否则表示该事务不会与现在正在复制中的事务形成死锁，进入步骤 3；

步骤 2 确定等待复制的事务所影响的数据行没有出现在活动事务特征信息中。如果没有出现，则进入步骤 3。否则，派发任务进入等待状态。

步骤 3 将等待复制的事务的事务特征信息加入到活动事务特征信息中。

当一个事务的状态由复制中转换为等待提交状态时，该事务的特征信息将从当前活动事务特征信息中移除。同时，通知唤醒派发任务退出等待状态，重试派发事务动作。可以看出，（1）和（2）所示步骤共享了一个可能性的数据对象-活动事务特征信息。因此，（1）和（2）必须互斥访问活动事务特征信息。

4.2.4 保证事务提交顺序

不论采用了乐观还是悲观模式的并发事务复制，由于采用了多个复制线程分别复制不同的事务的数据变更并且这些线程互相之间并不互相通信，因此必须采用某种机制保证这些复制线程执行提交命令的相对时间次序。为了实现这个要求，在复制系统中维护了一个事务提交顺序单项链表，这个链表成为待提交事务链表。所有的待提交事务都是按照事务提交的顺序分配给不同的复制线程的，当一个事务被提交给一个复制线程时，该事务会被插入到待提交事务链表的头部。有一个独立的事务提交线程会检查该链表头部事务的状态，如果状态为可提交状态，就会直接执行该事务的提交命令。否则，将会进入等待状态。当一个复制线程完成了一个事务中除去提交命令之外的数据变更的复制后，该线程会将事务的复制状态修改为可提交状态，于此同时，该复制线程会比较自己所复制的事务序号是否与待提交事务链表头部的事务序号，如果一致，该线程还会对事务提交线程执行唤醒操作。其运行时的流程见图 4-8 所示。

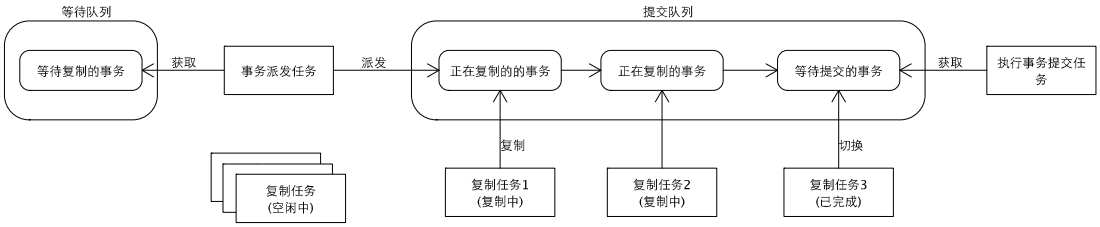

图 4-8 并发事务提交过程

Fig. 4-8 Parallel Transaction Apply

4.2.5 理想条件下乐观和悲观并发复制的性能特点

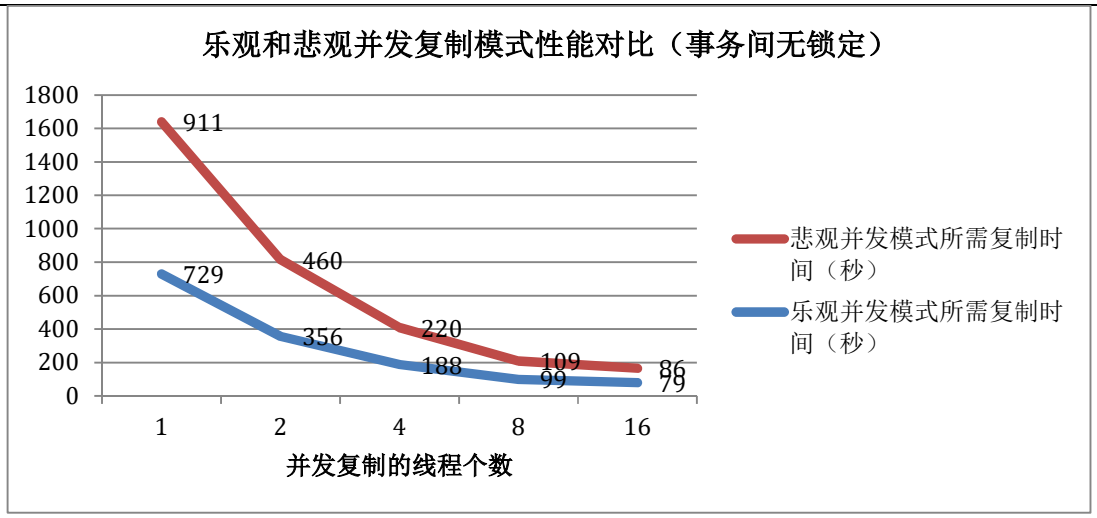

图 4-9 对比乐观和悲观并发复制模式的复制性能，使用理想化事务变更测试数据

Fig. 4-9 Performance comparisons between pessimistic and optimistic parallel mode using ideal transaction profile

如图 4-9 所示，如果使用比较理想的事务类型，即所有变更事务间都不会更新相同的表的数据行，不管是乐观还是悲观的并发复制模式都能通过增加并发复制线程的个数来加速事务复制的性能。悲观并发模式相对慢于乐观并发模式，这是由于悲观并发模式需要进行额外的 CPU 运算用于计算每一个事务的特征信息，比较幸运的是计算事务特征信息的计算都可以已事务为单位进行并发计算。可以看出，在理想情况下，乐观并发模式能够提供最优的并发复制性能。但是，在现实的生产过程中，事务之间是有可能分别更新相同的数据行的。一旦发生这种情况，不论是乐观还是悲观模式，都会有一定的性能损失：

对于乐观并发模式，需要对相互死锁的事务进行回滚，然后再顺序提交这些事务，此外，由于死锁的检测是在一定时间间隔后才会进行，这等于又增加了新的复制延时。

对于悲观并发模式，虽然能够提前识别出可能发生的死锁的情况，但是依然无法解决这些死锁，即可能互相死锁的事务依然是按顺序复制到目标数据库，失去了并发执行所带来的性能提升。在最坏情况下，即每一条事务都与上一条事务可能发生死锁，悲观并发模式倒退为事实上的单线程复制模式。为了解决上述问题需要在悲观并发模式的基础上进一步增加新的技术改造。

4.2.6 关联事务的并发复制

两个事务成为关联事务，是指这两个事务影响了同一个或者多个表的同一个或者多个数据行。在 4.2.5 节中可以得知，悲观并发模式需要进一步的技术改造，以便提高对关联事务的复制性能。

需要分析为什么两个关联事务需要按照事务提交的顺序提交。因为如果直接将这两个关联事务进行并发复制（即在两个数据库连接上并发、分别执行这两个事务的数据变更），可能会导致的情况是，需要最后提交的事务提前获取了第一个事务所影响的数据行的行级锁（row-level locks），此时第一个事务只能进入等待状态，由于必须严格遵循事务提交顺序，该事务必须等待第一个事务提交后方能提交，因此最简单的解决办法是首先把第一个事务的变更全部复制到目标数据库，然后再把第二该事务的变更复制到目标数据库。

根据上述的分析可以看出，正确并发复制两个关联事务的关键是以正确的顺序复制两个关联事务中共同涉及到的数据行的更新操作，正确的顺序即按事务提交的顺序。图 4-10 用一个例子说明了可能的情况。事务 T1 和 T2 都同时影响到了表 T1 的行 1，且 T1 提交早于 T2，如果要正确的并发复制这两个事务，对表 T1 的行 1 的更新，作为两个事务共同影响的行，必须按照事务提交的顺序，即事务 T1 中的更新先执行。

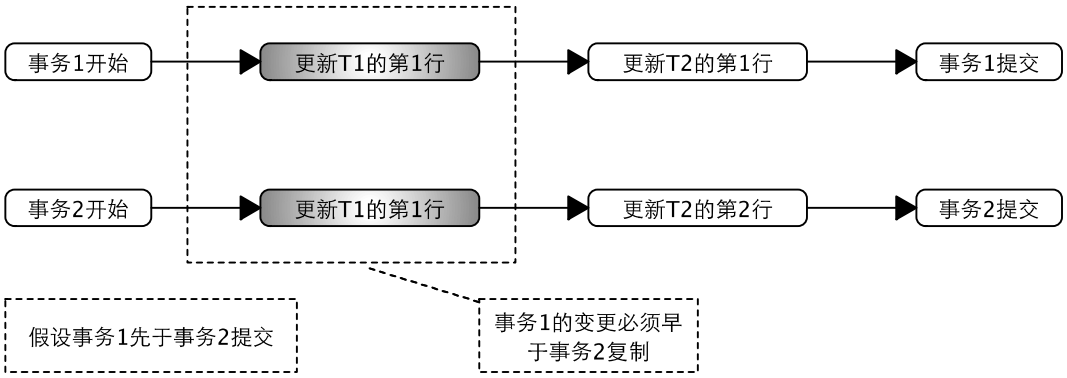

图 4-10 关联事务 T1，T2 的并发复制

Fig. 4-10 parallelized replicating two related transactions T1 and T2

此外有一种特别情况需要进行处理，即如果两个事务的共享的更新行所对应的数据更新日志操作恰好是这两个事务的第 1 条日志记录，即图 4-10 所示情况，由于当事务 2 在执行表 T1 的行 1 更行时会被数据库锁定直至事务 1 被提交或者回滚，即事务 2 依然在事务 1 提交后才开始被写入目标数据库，因此实际执行顺序如图 4-11 所示，这种方式完全无法获得并发事务提交所带来的好处。

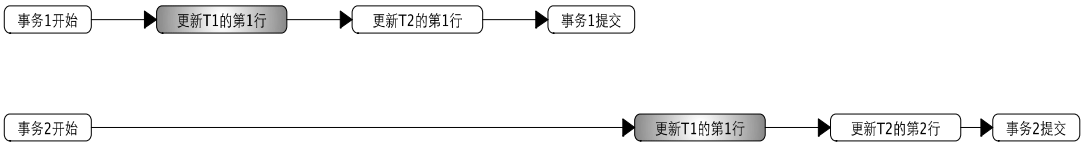

图 4-11 关联事务 1 和事务 2 的实际执行顺序

Fig. 4-11 The actual execution sequence of transaction 1 and 2

为了保证事务之间获得最大可能的并行度，必须将所有事务内部的数据变更进行重新排序，即先执行非共享更新的数据行的变更，最后执行共享更新的数据行的变更。这样做对目标数据库的数据使用者而言是感知不到的，这是由事务的原子性保证的。按照这个思路，图 4-10 所示的关联事务的重新排序后的执行时序如图 4-12 所示：

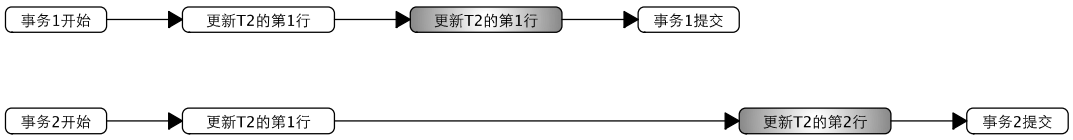

图 4-12 关联事务 1 和事务 2 的优化后执行顺序

Fig. 4-12 The optimized execution sequence of transaction 1 and 2

为了实现上述的关联事务的复制方式，必须解决两个子问题，即：

- 问题 1:如何找到 N 个给定事务中的关联事务？并识别出这些关联事务所共同涉及到的数据行的集合，即关联变更集合。
- 问题 2:如何以最快的速度将关联事务所共同涉及的数据行复制到目标数据？

问题 1 可以简化描述为，给定 M 个非空集合 S_1，S_2，S_3…S_m，每个集合包括 N_1，N_2，N_3…N_m 个元素（元素不小于 1），找出一个集合 R，确保 R 中的每一个元素都出现在至少两个给定的集合中。解决问题 1 的一个简单的算法是，顺序遍历给定集合中的每一个元素，遇到一个元素，就将该元素所对应的计数器加 1。这个算法的时间复杂度为 O(N)，其运行时间与所有集合中包括包的元素的总个数成线性关系。在此基础上

做更一步的优化是对每一个集合 fork 出一个单独的 task，在 task 中遍历给定集合中的每一个元素并更新对应的计数器。在这种设计下，计数器本身是一个共享资源，通常的办法是采用同步锁定的方式，但是在 JAVA 中，有一种支持原子累加操作的类，即 AtomicInteger，其实现基于 xchgcmp 操作，可以在不引入锁的前提下安全进行原子累加操作，因此采用 AtomicInteger 作为计数器非常合适。

问题 2 的解决需要做更多性能上的考量。一个较为简单的实现是由提交线程负责执行每一个事务中与其他事务关联的变更操作，由于提交线程是按照事务提交的顺序处理事务的，这自然保证了关联的变更操作按照事务提交的顺序写入目标数据库的技术要求。但是，考虑到关联变更集合可能会很大，因此必须想办法将关联变更以并发的方式写入目标数据库。为了实现这个目标，对关联变更做了更多的额外处理，即：

- 对同一个表的所有关联变更，计算其净变更；
- 对不同表的净变更，并发写入目标数据库。

净变更是针对同一个表的全部关联变更而言的。一个比较简单的例子是，对某一个表的某一个数据行，在先后提交的两个事务中，分别先后对该数据行执行了更新和删除操作，这 2 个操作的实际效果和直接删除该数据行是一致的。下面是一个简单的例子：

事务 T1，T2 为关联事务。

以下为事务 T1 的内容：

BEGIN TRANSACTION 1

UPDATE T1 SET C=’NewValue’ WHERE PK=1

COMMIT TRANSACTION 1

以下为事务 T2 的内容：

BEGIN TRANSACTION 2

DELTE T1 WHERE PK=1

COMMIT TRANSACTION 2

上面两个关联事务的净变更为：

BEGIN TRANSACTION 3

DELETE T1 WHERE PK=1

COMMIT 3

可以看出净变更的计算可以减少向目标数据库实际发出的命令总个数。对于关联

变更较多的事务集合，对净变更的计算可以大幅减少实际写入目标数据库的命令个数从而对性能有一定的好处。关于关联事务的净变更计算的规则，详见 4.6 节。当净变更准备就绪后，可以将每一个表的净变更并发写入目标数据库。

综上可以看出，当 N 个有关联关系的事务进行并发提交时，包括了如下几个和目标数据库的交互步骤，且这些步骤的交互必须在同一个数据库事务中完成：

- 并发复制这 N 个事务中的非关联部分；
- 并发复制这 N 个事务中的关联部分。

在本次设计中，采用了两阶段提交，即 XA（两阶段提交）的技术来实现上述步骤的单事务提交。所有的主流数据库系统都支持这种技术，该技术的主要功能是允许一个事务包括一组子事务，当该事务提交后，数据库系统可以确保其下所有子事务都被提交，否则，这些子事务会被全部回滚。

如图 4-13 所示，获取 N 个事务之间所共同影响的数据行可以用 Fork-Join 的方式进行计算从而实现并行处理以加快速度。

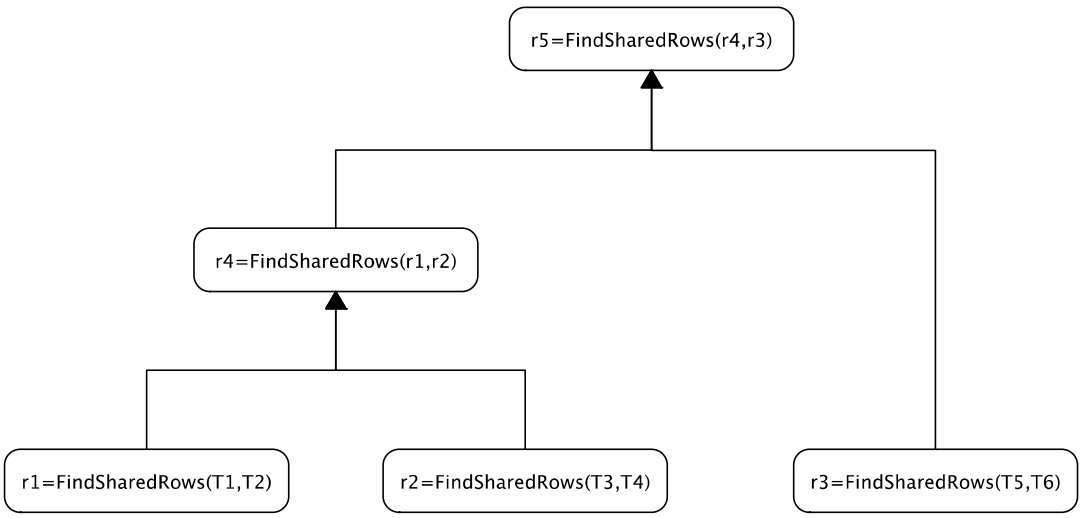

图 4-13 利用 ForkJoin 模型将一个大问题分解出可以并发执行的子问题

Fig. 4-13 Split a big problem into smaller problems using ForkJoin model

4.3 合并提交事务

4.3.1 提交命令对事务复制的性能影响

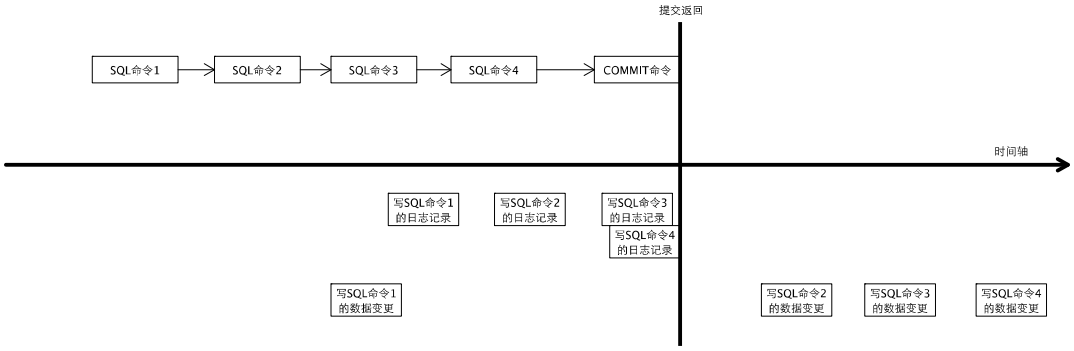

图 4-14 事务提交时刷新日志记录

Fig. 4-14 Flush transaction log upon commit command

由图 4-14 所示，数据库在执行提交命令时是有一定延时开销的。当数据库执行提交命令时，数据库系统必须保证将当前事务以及其他事务所产生的尚未被写入磁盘的事务日志同步地刷新到物理磁盘。而这些事务所产生的数据文件的变更，则是始终异步地写入到磁盘中的。由于数据库在执行数据变更命令时主要的性能延迟来自于 IO 系统，因此，执行提交命令的日志同步写入的动作对数据复制的性能有显著影响。

图 4-15 展示了用不同大小的事务顺序插入 1，000，000 行数据所需的时间，目标数据库是 ORACLE 11g R2，我们可以明显看出，随着每个事务包括更多个数的插入命令（意味着总的事务个数减少），插入全部数据行所需的总时间明显减少了。

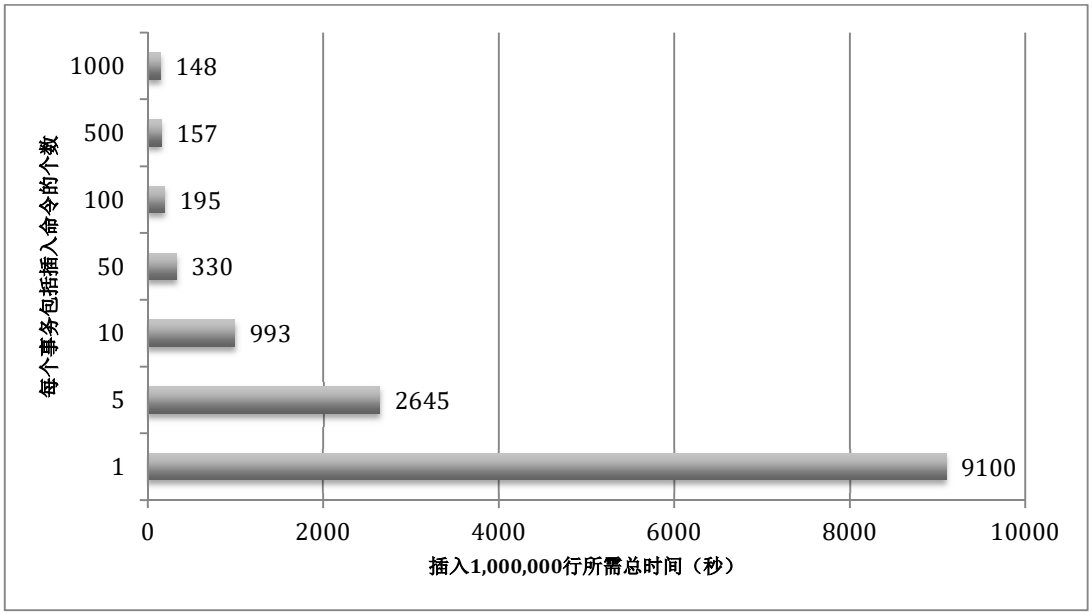

图 4-15 不同的大小的事务对复制性能的影响

Fig. 4-15 Transaction sizes' impact to replication performance

4.3.2 存储设备的性能和数据库刷新日志记录性能的关系

IOPS 是一种衡量存储设备性能的指标，即一个存储设备每秒钟最多能完成的无数据传输的读写操作的次数。影响 IOPS 的因素主要包括，所采用的磁盘的机械和电子性能、RAID 级别等。常见的 10,000 转的单机械磁盘的 IOPS 在 150 以内，15,000 转的单机械磁盘的 IOPS 在 200 以内，而基于固态硬盘（SSD）的 IOPS 可以达到 10，000 以上。IOPS 衡量了硬盘单位时间内所能接受的读写操作的上限值，由存储设备本身固有的设计和结构决定，例如对于磁盘，IOPS 主要受到磁盘寻道时间（将磁头定位到所需要数据的磁盘扇区所需时间）的影响，如果再加上传送数据的时间，则实际的 IOPS 会更低。本论文主要假设基于机械硬盘的存储设备。

数据库执行提交命令时，日志缓冲区里的日志记录就会被同步的刷新到日志存储设备。每当日志记录刷新到磁盘时，就发生了一个写操作。写日志记录的操作的发生频率不会超过其所在存储设备 IOPS 性能。为了降低磁盘寻道时间所带来的延迟，数据库会维护一个基于内存的日志缓冲区。在没有提交命令的前提下，所有的日志都会被首先写入日志缓冲区，然后异步的刷新到磁盘。但是只要出现了提交命令，日志缓冲区中的日志记录会被全部同步的刷新到磁盘。因此，需要尽可能的保证每次日志缓冲区被刷新时，缓冲区内部有足够多的日志记录可以被写入存储设备，也就意味着，需

要尽可能减少向数据库高频率地提交小事务。

4.3.3 数据库提交性能评价

数据库提交性能是指在单位时间内，向目标数据库执行提交命令的次数（命令执行频率(CR， Commit Rate)）。一个合理的 CR 必须介于提交性能下限(CRLV，Commit Rate Low Value)和提交性能上限（CRHV， Commit Rate High Value）之间，并尽可能接近 CRLV。

CRHV 是指，单位时间内提交仅有一条简单插入命令的事务的个数，该数字接接近于日志存储设备的 IOPS 值。对于一条仅仅包括一条简单插入命令的事务而言，其对应的数据文件更新的数据量很小，而且数据文件的更新始终都是异步执行的，因此，这个事务主要的时间都花在在执行提交命令时同步刷新日志文件所需要的时间。而日志文件更新的数据量也很小，因此提交命令的延时中主要是磁盘设备的寻道时间，该数值接近于日志存储设备的 IOPS 值。

CRLV 是指单位时间内，完成同步刷新一个已经充满的日志缓冲动作的次数。该数值接近于理想极限情况，即每一个事务所产生的日志都正好占满整个日志缓冲区，在执行提交命令时，整个缓冲区的日志变更全部同步刷新到存储设备。在这种情况下，存储设备的寻道时间将不会占到整个日志同步刷新的最主要部分。

CRHV 和 CRLV 全部由程序自动测试得到。

4.3.4 自动选择合并后事务的尺寸的算法

合并连续几个小事务组成一个相对大的事务，是减少向目标数据库执行提交命令次数的一种办法。通过合并几个连续的变更事务，我们可以减少单位时间内向目标数据库执行提交命令的总次数，从而减少了提交命令本身所固有的延迟。

如图 4-16 所示。在最初始，事务派发任务以合并度为 1 对等待复制的事务进行合并，即每个合并后的事务对应 1 个原数据库事务。经过一段时间的提交，提交任务计算这段时间内的事务提交频率。通过比较事务提交频率 CR 与 CRLV 和 CRHV 的值，给事务派发任务发出建议提高合并度或者降低合并度的通知，目标是，希望 CR 接近 CRLV 远离 CRHV：

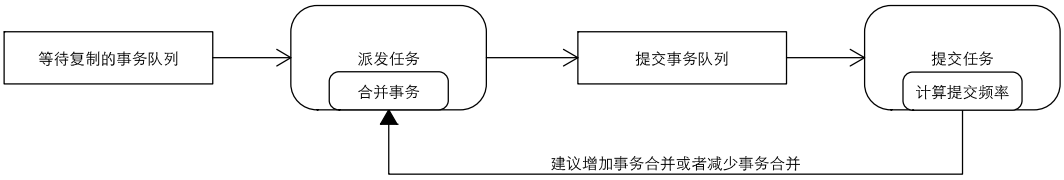

图 4-16 提交任务向派发任务建议增加或者减少事务合并度

Fig. 4-16 Transaction commit task gives feedback to transaction dispatcher to group more or less transactions

(1) 如果 $CR > CRHV - (CRHV - CRLV) * 0.90$，则建议提高合并度；

(2) 否则，建议降低合并度

当事务派发任务接收到提高事务合并度建议后，事务合并度被扩展为原来的 2 倍；否则，缩减为原来的 2 倍。事务派发任务并不会为了完成目标事务合并度而做任何等待，如果发现等待复制的事务队列已经为空，那么无论当前合并后事务是否已经达到目标事务合并度，都会将这个合并后事务派发给一个空闲的复制任务。

4.3.5 单项性能测试

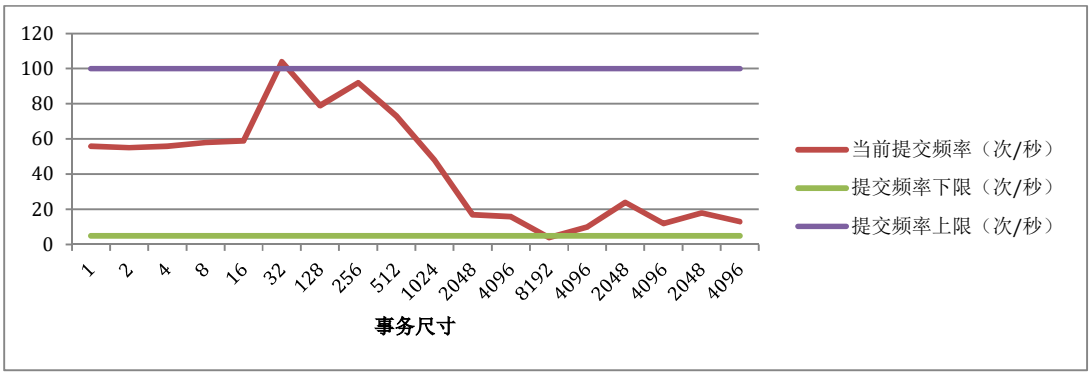

图 4-17 通过调节事务尺寸自动调节提交频率

Fig. 4-17 Adjust grouped transaction size for tuning commit rate

为了测试事务合拼算法的有效性，设计了一个如下实验。复制 1，000，000 个事务，每个事务包括 1 个 SQL DML 操作，在复制过程中监控事务派发任务的合并后事务的大小和当前提交频率。

图 4-17 可以看出，当前提交频率在复制的过程中，在接近提交频率下限附近波动，尽量远离了提交频率上限。测试的环境采用了一台具有 4 个志强处理器和 16G 内存的 PC 服务器，包括一块 7200 转的 STAT3 接口的硬盘，运行 CentOS 6.4 操作系统和 ORACLE 11gR2。ORACLE 的重做日志缓冲区被设置为 32M。根据测算，提交频率上限为 100，提交频率下限为 5，而在复制运行的过程中，当前提交频率确实始终在上下限区间内变化。

4.4 采用预编译 SQL 提交增删改操作

所有主流的数据库都提供了一种称为预编译 SQL[20]语句(Prepared SQL Statement)的接口。执行预编译时，用户只需要提供一个语法正确的 SQL 语句的模板，比如：

INSERT INTO table (c1， c2) VALUES (?， ?)

当用户真正开始执行这个语句时，只需要直接模板中问号处的值即可。

在正常情况下，当用户提交一个完整的 SQL 语句给数据库系统后，这个 SQL 语句会经过一系列复杂的计算分析步骤，主要包括：

- 解析 SQL 语句
- 检查权限
- 产生多种执行计划
- 选择一种较优的执行计划
- 调用执行引擎，执行这个执行计划
- 返回结果集

但是对于数据复制这个特殊的场景，需要复制执行的数据变更 SQL 语句只有插入、更新、删除 3 种，且对于同一张表而言，这 3 个语句的基本格式全部相同，唯一不同的就是数据行的值不同。如果每次执行某一个 SQL 数据库系统都要经过相同的解析、分析、计划、执行这些步骤，则是一种对数据库 CPU 资源的浪费。

预编译语句可以缩短 SQL 语句执行的时间，这是由于经过预编译后，数据库系统将会保存这个 SQL 语句的执行计划，当下次用户再次执行这个语句时，只需要提供 SQL 语句所需要的输入参数值，以避免重新对 SQL 语句进行语法分析和执行计划的计算动作，而直接开始按照预先编译好的执行计划落实这个 SQL 语句的语义功能。

创建预编译句柄是一个很耗时的操作。所有的预编译 SQL 在使用前都需要通过与

数据库通信以完成预编译过程，并得到一个对应的预编译句柄。但是在复制的过程中，没有必要每次复制某个表的某个变更日志时都去做一次预编译过程，可以缓存这个句柄以便重用。

预编译句柄和数据库连接是关联的。即由数据库连接 1 创建出的预编译句柄不能在数据库连接 2 的上下文使用，即便这两个数据库连接是连接到同一个数据库。由于采用了并发复制的机制，需要为每一个连接到目标数据库的数据库连接缓存预编译句柄。

在本次设计中，为了实现上述目标，设计和实现了一个预编译句柄的缓存。每一个目标数据库连接都维护自己的预编译句柄缓存。为了节约内存资源，采用了 LRU 的算法对缓存中的预编译句柄进行淘汰。

表 4-1 是一个比较预编译 SQL 语句和普通 SQL 语句在执行增、删、改操作的性能对比实验结果。测试的方法是分别用普通 SQL 或者预编译 SQL 增加、删除、更新 1，000，000 行数据所需要的时间，可以看出，预编译后的 SQL 语句执行速度比普通 SQL 语句快 70%以上。

表 4-1 不同类型的 SQL 操作在预编译模式下的复制时间

Table 4-1 Replication latency time between normal SQL parsing mode and prepared SQL execution mode

测试案例	复制时间（单位：秒）
INSERT，普通 SQL	467
INSERT，预编译 SQL	148
DELETE，普通 SQL	539
DELETE，预编译 SQL	174
UPDATE，普通 SQL	569
UPDATE，预编译 SQL	143

4.5 采用 BULK LOAD 接口执行连续的数据插入

有的时候，用户需要对同一张表执行大量地、连续地插入操作。如果用普通 SQL 语句来执行，即便采用了动态 SQL 的技术，插入速度也仅仅是满足基本要求。如果希望更快，则必须采用 BULK LOAD 接口。

BULK LOAD 接口之所以迅速，需要首先看看数据库系统是如何执行插入语句的

。首先，数据库需要检查被插入数据的合法性，例如边界，长度等，然后，计算出所需要的存储空间的长度，申请对应长度的行空间，并按照一定的格式把这些列的值依次填入行数据存储空间中。接着，产生一条对应的数据库插入操作日志记录并写入恢复日志文件。最后，将数据行提交到数据文件写入缓存。对于复制操作而言，上述操作很多是不需要的，例如数据合法性检查，由于复制到从数据库的操作都是从主数据库的日志文件中获取，这些数据都是经过主数据库验证的，无需再次验证。

BULK LOAD 操作则是直接将列值格式化成数据库系统能够直接写入数据文件的行格式，数据库系统也不会对数据的合法性做详细检查，仅会做一些基本校验，直接将数据行写入到将数据文件中。

通过测试，BULK LOAD 的性能比普通 SQL 插入操作有非常显著的提升。但是 BULK LOAD 操作具有一定的启动开销，即如果插入的数据行数比较小，在某一个阀值之下后，BULK LOAD 的平均性能比 SQL 插入操作要差。因此，我们需要寻找这个阀值，并尽可能根据事情情况选择 BULK LOAD 操作还是 SQL 插入操作。

表 4-2 是一些针对 ORACLE 数据库 BULK LOAD 性能的测试数据。

表 4-2 不同插入方式下连续插入 1,000,000 行数据所需的时间

Table 4-2 Replication latency time for inserting 1,000,000 data rows using different insert execution mode

测试案例	所需时间（单位：秒）
用 SQL 语句连续插入 1，000，000 行数据	500
用预编译 SQL 连续插入 1，000，000 行数据	148
用 BULKLOAD 连续插入 1，000，000 行数据	4

4.6 关联事务的净数据变更(Net Data Change)的计算

由 4.2.6 节中，我们将一组关联事务的复制分解为两个部分，即非关联变更部分的复制和关联变更部分的复制[46]。关联变更即针对相同表，相同数据行在不同事务中的变更行为。为了提高关联变更的复制性能，我们引入了净变更的概念。简单的说，净变更就是将针对同一个表、同一个数据行的变更进行预处理，然后得出其实际最终的变更。计算关联变更的规则由下表列出 [47][48]：

- 如果同一行的上一次操作是插入操作，当前操作是插入操作，则是出错情况；

- 如果同一行的上一次操作是更新操作，当前操作是插入操作，则是出错情况；
- 如果同一行的上一次操作是删除操作，当前操作是插入操作，则将删除操作从净删除行集合移除，将插入操作加入净插入行集合；
- 如果同一行的上一次操作是插入操作，当前操作是更新操作，则将插入操作从净插入行集合移除，将更新操作加入净更新行集合；
- 如果同一行的上一次操作是更新操作，当前操作是更新操作，则是合并这两个更新操作的更新后列值，并替换净更新行集合中的更新操作；
- 如果同一行的上一次操作是删除操作，当前操作是更新操作，则是出错情况；
- 如果同一行的上一次操作是插入操作，当前操作是删除操作，则将插入操作从净插入行集合移除；
- 如果同一行的上一次操作是更新操作，当前操作是删除操作，则将更新操作从净更新行集合中移除，并将删除操作放入净删除行集合；
- 如果同一行的上一次操作是删除操作，当前操作是删除操作，则是出错情况；

净变更复制主要适用于两种事务类型，第一种是高速、重复更新固定数据行的事务，另外一种是包括大量连续删除、更新类型的事务。对第一种类型的事务，其性能提升是显然的，因为净变更可以将一段时间内的大量更新操作转换为一个相对更小的更新集合，降低了数据库执行大量 SQL 变更的总时间。

对第二种类型的事务，净变更可以将一组连续删除变为一组插入（只需要插入需要删除的数据行的主键值，插入到一个临时表 T 中）和 JOIN DELETE 操作（将临时表 T 与目标表做链接后执行删除，DELETE 目标表，T WHERE 目标表.PK = T.PK）。由于插入操作可以通过 BULK LOAD 的方式进行，根据 4.5 节可知，速度是非常快的。而 JOIN DELETE 的操作，数据库可以通过主键索引进行循环匹配，速度比单独发送 N 条 DELETE 语句要更为迅速。UPDATE 语句的执行情况与 DELETE 类似。

下面是针对第二种事务类型的单项测试数据。测试中，我们首先采用净变更法去删除和更新 100 万行数据，再用普通 SQL 语句的方式去删除和更新 100 万行相同的数据（既 100 万条 DELETE 语句和 100 万条 UPDATE 语句），分别获取这两种情况下的 SQL 语句的执行时间，并列表（表 4-3）进行数据比较。这些测试数据都是测试 10 次后的平均值，可以看出采用净变更法的数据提交方式性能较好，而采用单条 SQL 语句

的数据提交方式性能较差。

表 4-3 采用净变更执行删除或者更新 1,000,000 行数据所需要的时间

Table 4-3 Replication latency time after using net-change apply method

测试案例	所需时间（单位：秒）
用 DELETE SQL 语句删除 1，000，000 行数据	2178
用净变更法删除 1，000，000 行数据	64
用 UPDATE SQL 语句删除 1，000，000 行数据	2412
用净变更法更新 1，000，000 行数据	81

4.7 本章小结

本章主要针对新的数据库复制系统中的关键技术进行了详细的描述。这些技术包括：版本化管理复制对象结构信息的技术，通过这个技术保证复制系统在处理日志信息时不再直接依赖源数据库数据字典的信息；连续事务的并发式提交实现了乐观和悲观的事务并发提交技术，以便适用于不同的应用场合；针对悲观的事务的并发提交，设计了一种关联事务的并发复制算法，即并发提交关联和非关联变更部分，由于采用了净变更的压缩技术，可以将关联部分中实际需要写入目标数据库的数据变更的个数降低到最少。此外，还进一步研究了不同的数据库编程接口对数据库复制的性能影响，可以看出正确使用数据库现有的编程接口，可以提高数据变更的写入性能。

5 数据复制性能对比测试设计

本章主要描述数据库复制性能测试的原理，并设计一系列性能测试实验，对比新的复制数据流和现有产品之间的性能差别。

5.1 数据复制性能测试基本原理

5.1.1 数据库复制性能指标

数据库复制的性能指标主要有如下几种：

（1） 恢复日志处理能力(LOGT， Log Size Throughput)

该指标用于表示一个复制系统每个小时能够处理的数据库回复日志的总尺寸。

单位是 GB/HR (GB 每小时)

（2） 事务处理能力（TPS，Transaction Per Second）

该指标用于表示一个复制系统每秒钟能够处理的事务个数。

单位是 T/S (Transaction 每秒)

（3） 命令处理能力（CPS，Command Per Second）

该指标用于表示一个复制系统每秒钟能够处理的命令个数。

单位是 C/S (Command 每秒)

5.1.2 数据库复制性能测试原理

（1） TICKET 的处理过程

- 用户在主数据库执行一个特定的存储过程向 ticket 表格中插入一行新数据，这行数据的 SRCDBTS 字段被填入主数据库系统当前系统时间；
- 这条插入操作的日志记录会被发送到复制系统。当日志记录进入复制系统时，其 CAPTURETS，LOGSIZE， CMDCOUNT， TXCOUNT 会被填入复制系统的当前系统时

间，已经复制的恢复日志总数，已经复制的命令个数，已经复制的事务个数。当日志记录离开复制系统时，APPLYTS 会被填入复制系统的当前系统时间。

- 最终，TICKET 会被写入从数据库的 ticket 表格中，并且 DSTDBTS 会被写入从数据库的当前系统时间。

TICKET 表如表 5-1 所示。

表 5-1 TICKET 表结构

Table 5-1 Table Structure of TICKET

字段名称	意义
NAME	TICKET 的名称
SRCDBTS	TICKET 插入主数据库时的系统时间，单位是微秒
CAPTURETS	TICKET 进入复制系统时的系统时间，单位是微秒
APPLYTS	TICKET 离开复制系统时的系统时间，单位是微秒
DSTDBTS	TICKET 进入从目标数据库时的系统时间，单位是微秒
LOGSIZE	TICKET 进入复制系统时已经复制的总恢复日志尺寸，单位是字节
CMDCOUNT	TICKET 进入复制系统时已经复制的命令总个数
TXCOUNT	TICKET 进入复制系统时已经复制的事务总个数

（2） 利用 TICKET 计算复制系统的复制性能

通常，我们需要至少两个 TICKET 才能计算复制系统的复制性能。

- 在源数据库插入 Ticket 1
- 在源数据库产生若干数据变更
- 在源数据库插入 Ticket 2
- 取出目标数据库中的 TICKET1 和 TICKET2 的数据行

根据这两个 TICKET 的字段值，我们可以得到如下计算公式：

- 复制总时间(单位:小时)

$$TotalRepTime = \frac{Ticket2.ApplyTS - Ticket1.CaptureTS}{1000 * 3600} \tag{5-1}$$

- 复制的恢复日志总量（单位：GB）

$$TotalLogSize = \frac{Ticket2.LogSize - Ticket1.LogSize}{1024^3} \tag{5-2}$$

- 复制的总事务个数（单位：个）

$$TotalTXCount = Ticket2.TXCount - Ticket1.TXCount \tag{5-3}$$

- 复制的总命令个数（单位：个）

$$TotalCmdCount = Ticket2.TXCount - Ticket1.TXCount \tag{5-4}$$

- 恢复日志处理能力（单位 GB/HR）

$$LogT = \frac{TotalLogSize}{TotalRepTime} \tag{5-5}$$

- 命令处理能力（单位：个/S）

$$CPS = \frac{TotalCmdCount}{TotalRepTime * 3600} \tag{5-6}$$

- 事务处理能力（单位：个/S）

$$TPS = \frac{TotalTXCount}{TotalRepTime * 3600} \tag{5-7}$$

5.2 性能测试实验设计

5.2.1 实验目的

实验的主要目的，是测试现有复制系统和改进后的复制系统的性能差别。主要是

获取如下两个方面的测试数据：

（1） 复制吞吐量

复制吞吐量是用于衡量一个复制系统处理恢复日志、事务、命令的速度。由于复制系统的复制性能在不同的事务、命令、表结构、字段长度的条件下会有不相同的表现。因此，在本次测试中，我们选用了几种比较常见的数据库事务产生模型用于测试数据的产生工具。此外，我们还会测试几种常见的表模型用于更加了解新复制数据流的性能。

（2） 可伸缩性能

可伸缩性性能测试主要用于测试一个复制系统在不同的 CPU 资源情况下的复制吞吐量。如果一个复制系统具有良好的可伸缩性，则其性能会跟随可用 CPU 个数的增多而提高，这样的系统可以提供更好的复制出力能以应对未来数据不断增长的趋势。

5.2.2 复制性能测试以及不同的事务产生模型

(1) 复制性能测试的步骤

- 启动主数据库，从数据库，复制服务器，并进行必须的配置
- 创建测试需要的表格，并装入初始化数据
- 暂停主数据库到复制服务器的日志传送
- 发送 TICKET 1
- 执行变更事务产生器
- 发送 TICKET 2
- 重新启动主数据库到复制服务器的日志传送
- 等待，所有的 TICKET 都已经抵达从数据库
- 计算复制性能指标
- 重复上述测试过程，取各个性能指标的平均值

(2) 变更事务产生模型

- TPCC Benchmark
- CITI Bank Benchmark

- Knight Security Benchmark
- 模拟测试 1:单表、全关联事务性能测试
- 模拟测试 2:多表、全关联事务性能测试

5.2.3 可伸缩性性能测试

(1)测试步骤如下

- 启动主数据库，从数据库，复制服务器，并进行必须的配置使得复制服务器最多使用 N 个 CPU 内核
- 创建测试需要的表格，并装入初始化数据
- 暂停主数据库到复制服务器的日志传送
- 发送 TICKET 1
- 执行变更事务产生器
- 发送 TICKET 2
- 重新启动主数据库到复制服务器的日志传送
- 等待，所有的 TICKET 都已经抵达从数据库
- 计算复制性能指标
- 重复上述测试步骤，并设置 N=1，2，3，4，并收集性能测试结果

(2)变更事务产生模型

- TPCC Benchmark

5.3 性能测试结果

测试过程中，一共使用 3 台配置完全相同的 Linux 服务器，其具体配置如下所示：

硬件配置

CPU: Intel® Xeon® CPU X7350 @ 2.93GHz (超线程打开，共 4 核，8 线程)

内存：64G

硬盘：146GB x 4 / 15000 转 / SAS 接口

网卡：10GBPS LAN

软件环境

操作系统：Redhat Linux Enterprise V5.9

主数据库: Sybase Adaptive Server Enterprise 15.7.1 ESD1

从数据库：Sybase Adaptive Server Enterprise 15.7.1 ESD1

5.3.1 复制性能测试结果

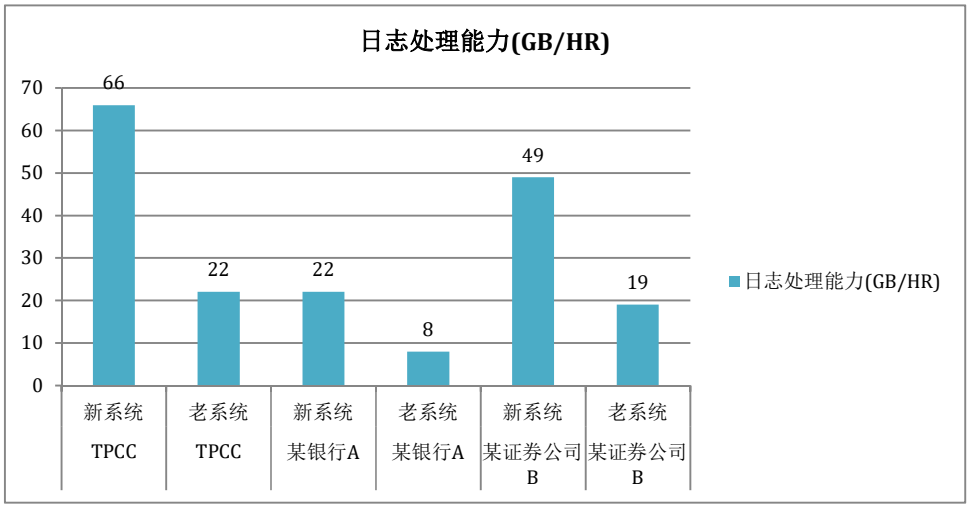

图 5-1 新、老复制系统日志处理能力对比

Fig. 5-1 Log processing capability comparison

图 5-1 显示了新的数据复制系统在不同的试用场景下的日志处理能力的实测平均值，也显示了老的数据复制系统的日志处理能力的实测历史平均值。可以看出，采用新的数据复制系统后，日志处理能力得到了提高。

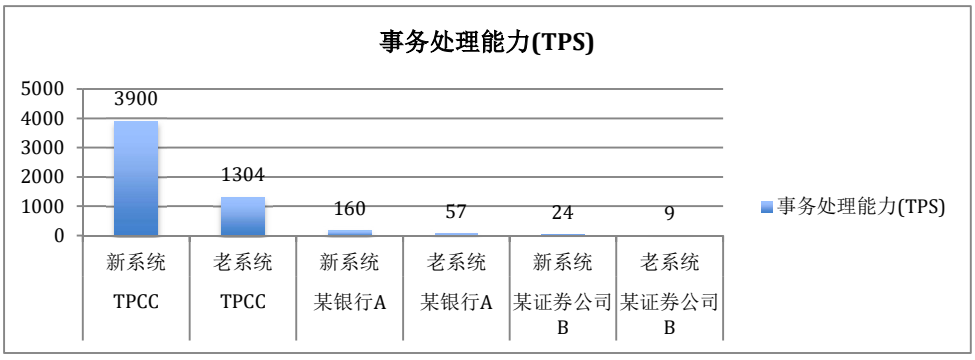

图 5-2 新、老复制系统事务处理能力对比

Fig. 5-2 Transaction processing capability comparison

图 5-2 从事务处理性能的角度对比了新、老数据复制系统的处理能力。可以看出，事务处理能力得到了一定的提高。

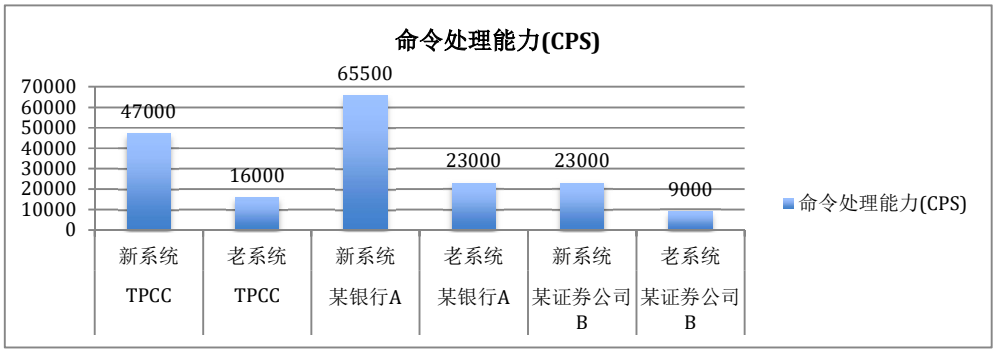

图 5-3 新、老复制系统命令处理能力对比

Fig. 5-3 Command processing capability comparison

图 5-3 从命令处理性能的角度对比了新、老数据复制系统的处理能力。可以看出，新系统可以提高每秒钟能够处理的 SQL 语句的个数。

5.3.2 可伸缩性性能测试结果

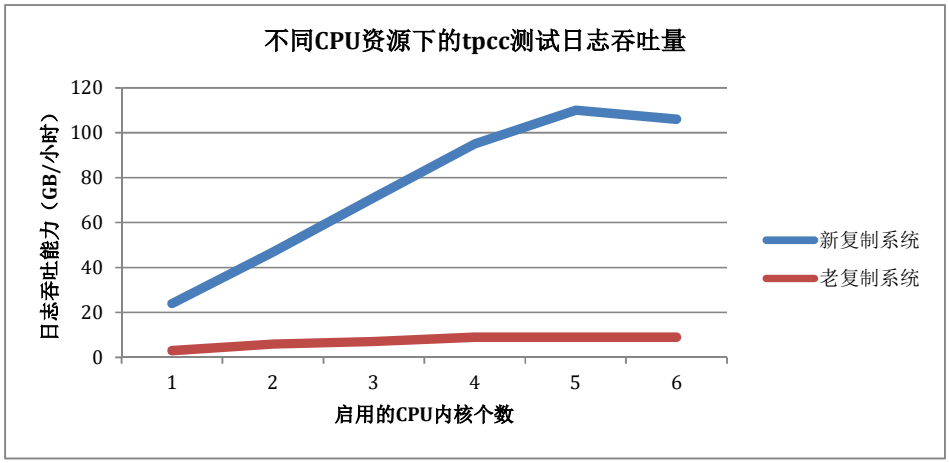

图 5-4 新、老复制系统的可伸缩性的对比测试

Fig. 5-4 Scalability capability comparison

图 5-4 是显示了在启用 1 个至 6 个 CPU 内核的测试条件下的，采用 TPCC 数据产生器的数据复制日志处理能力的比较测试数据。可以看出，新的数据复制系统由于采用了较细力度的任务分割设计，随着启用 CPU 内核个数的增多，日志处理能力有随之得到改善。

5.4 实验结果分析

通过上面的测试结果可以看出，新的数据流的恢复日志处理能力较现有产品有了成倍的提升。并且，新的数据复制数据流的可伸缩性较明显的优势。

TPCC 是标准的数据库性能基准测试标准，用于反应和测试数据库的在线事务处理能力。通过复制 TPCC 所产生的测试变更数据，可以反应出数据复制系统在在线事务处理环境内的复制性能和行为。在 TPCC 的测试中，性能提高到原先的 8.6 倍。

某外资银行 A 的测试中，其系统为全球超过 2 亿个客户提供各种金融银行服务。在测试中，所涉及到的系统日间事务产生频率不高，但是在夜间的批量处理中包括大量插入和更新操作，涉及到的数据行超过 1，000，000，000 行，复制性能提高到原先的 9.1 倍，对夜间的批量处理所产生的事务的复制延迟由原先的 8 小时缩短到 3 小时。

某证券服务公司 B 的测试中，所涉及到的系统每秒产生超过 10，000 个事务，以插入操作为主。在本项测试中，其数据复制的性能提高到原先的 11.4 倍。平均复制延时低于 5 秒。

6 总结

本章主要对本论文的贡献进行了总结，并对本论文的不足进行了总结，为下一步工作的工作方向做出了建议。

6.1 论文的主要贡献

本论文的主要贡献是对传统的基于事务的数据库复习系统进行了技术升级，即通过广泛采用并发执行的机制对所需要复制的连续事务进行并发复制并保证事务提交的前后次序[49][50]。具体表现在：

1) 设计和实现了一种基于并发执行的子任务流水线架构，用于实现高性能的数据库复制系统。
2) 设计实现了通过分析数据变更事务之间的相对依赖关系，并发复制不同事务中的数据变更，提高数据复制性能的方法。
3) 设计实现了一种合并事务提交的算法，通过合并小事务的方式，提高了数据复制性能。
4) 研究了数据库的特别编程接口的性能特点，针对特定事务类型进行特别优化，提高了数据复制的性能。
5) 设计实现了基于数据净变更的复制方法。
6) 设计实现了一种基于版本化管理数据库对象复制状态的方法。
7) 对实现后的系统进行了针对性的性能测试，验证了本文复制方法的有效性。

本文实现的数据复制性能优化方法，已经在多个银行、证券公司内部得到成功试用性验证，相对传统的数据复制系统的复制性能得到了明显的提升。

6.2 存在的不足

在测试和用户现场测试的过程中，也发现了一些问题，主要有:

1) 数据复制优化算法较多，用户不知道如何选择[51][52]

所有的数据库复制性能提升技术都有其应用的局限性，是为了某一种特定类型的事务有显著效果，而对其他事务类型效果不明显甚至产生副作用。甚至在不同时间段，事务类型也是不同的。为了得到最好的数据复制性能，用户只能通过不断测试不同的复制方法的组合确定最佳配置，然而，这种方法的问题在于需要用户清楚理解每一种复制性能提高方法的适用范围并对生产系统所产生的事务类型有一定了解。这种要求对用于而言是相当高的。

2) 内存占用与 JAVA 垃圾回收对复制性能的影响

在实验室和客户现场试用中发现，复制系统的内存占用较高，有时甚至高达 30GB 以上，并产生大量的垃圾回收操作（Major GC），从而导致明显的复制停顿。经过初步分析发现，系统中所有数据变更都采用对象表述，表名，子段名，字段值全部都是 JAVA 对象，随着复制吞吐量提高，JAVA 对象的个数也随之提高，对内存和垃圾回收系统带来一定的压力。

6.3 下一步工作的建议

根据测试和使用中发现的具体问题，将会开始如下方面的研究：

1） 研究识别事务类型的方法

为了减少用户的使用和调优难度，必须研究一种自动识别当前事务特性的方法。下一步的研究将重点研究这种识别方法。

2） 研究如何减少 java 对象的创建个数以及频率

根据 profiling 的结果，在数据复制的过程中，大于 40%的对象是用于表达每一个数据变更中的字段值，初步的研究意向是将这些字段值对象进行合并，即同一个变更操作中的所有字段的值全部按照一定的格式放入到同一个字节缓冲区中， 从而减少这部分对象的实际创建个数，减轻对 JAVA 内存回收操作的压力。

3） 在这个数据复制框架的基础上，开发无数据丢失的数据复制协议

在数据库双机热备中，如果要实现真正的高可用性(HA, High Availability)[53][54]，必须保证数据复制能够实现零数据丢失。这需要数据库系统和复制系统进行更紧密的结合。这是一个具有一定深度的研究方向。

参考文献

[1] 闫波, 数据复制技术在灾备系统中的研究与应用[J], 计算机技术与发展, 2006, 17(9): 250-253.

[2] 王春晓, 分布式数据库数据复制技术的研究[J], 中山大学学报(自然科学版), 2009, (1): 366-368.

[3] 郑祥云, 数据库同步中差异数据捕获方案设计与实现[J], 电脑知识与技术, 2009, (7): 1544-1548.

[4] 熊华平, 大型异构数据库数据迁移系统的研究与应用[J], 计算机应用与软件, 2012, (7): 178-181.

[5] 孙健, 中国铁路信息共享技术研究和实现[J]，计算机工程, 2004, (20): 171-173.

[6] 王运霞, 铁路客票系统数据一致性的实现技术[J], 中国铁道科学, 2000, (4): 3-7.

[7] 高振清, 分布式数据库数据复制技术研究[J], 延安职业技术学院学报, 2013, (5): 105-106.

[8] 杜凯, 数据库复制技术研究进展[J], 计算机工程与科学, 2008, (7): 118-121.

[9] 周芃, 利用高级复制功能和触发器实现数据复制[J], 中国医疗设备, 2008, (11): 37-39.

[10] Oracle, Inc, Understanding Oracle Streams Replication[EB/OL], http://docs.oracle.com/cd/B28359_01/server.111/b28322/gen_rep.htm#STREP011, 2015-01-01.

[11] 邹先霞, 基于数据库日志的变化数据捕获研究[J], 小型微型计算机系统, 2013, (3): 531-536.

[12] 王玉标, 异构环境下数据库增量同步更新机制[J], 计算机工程与设计, 2011, (3): 948-951.

[13] 孟雷, 基于 P2P 的异构数据库数据同步研究[J], 山东大学学报(理学版), 2008, (11): 72-76.

[14] Sybase, Inc, Replication Strategies:Data Migration，Distribution and Synchronization[EB/OL], http://tdwi.org/whitepapers/2013/05/sap_rtdm_replication-strategies-data-migration-distribution-and-synchronization.aspx, 2013-05-01.

[15] 毕利, Sybase Replication Server 数据复制技术及实例[J], 铁路计算机应用, 2007, (8): 139-144.

[16] 李海荣, 基于 Web Service 的异质数据库对等更新的研究与实现[J], 计算机应用与软件, 2006, (5): 43-45.

[17] 倪泳智, 一种基于虚拟日志的数据复制解决方案[J], 计算机科学, 2007, (3): 78-85.

[18] 杨剑, Oracle Data Guard 容灾技术的研究与实现[J], 现代计算机(专业版), 2012, (19): 40-43.

[19] 周春莲, 基于时间戳的数据同步技术实现研究[D], 南昌:南昌大学, 2009.

[20] 袁满, 一种融入 MD5 的影子表算法研究[J], 计算机应用研究, 2013, (1): 149-151.

[21] 史晶波, 在 DB2 中提取增量数据的一种方法[J], 计算机与数字工程, 2004, (6): 15-16.

[22] 程守远, 基于 Replication Server 的数据复制系统及模型设计[J], 铁路计算机应用, 2009, (11): 58-65.

[23] 朱跃龙, Sybase Replication Server12.0 的复制服务器技术研究及应用[J], 计算机与现代化, 2002, (12): 59-61.

[24] 宋兴彬, 基于 Sybase 复制技术的分布式数据库系统的建立[J], 山东科学, 2000, (1): 42-45.

[25] 王新, Oracle 容灾在银行业务系统中的研究应用[D], 四川省 :电子科技大学, 2011.

[26] 武冬春, 基于 GoldenGate 技术实现关键业务容灾的解决方案[J], 信息通信, 2013, (7): 237-238.

[27] 许继楠, 解读 IBM InfoSphere 大数据分析平台[N], 中国计算机报, 2012-01-16(016).

[28] Doug, Lea, A Java Fork/Join Framework[EB/OL], http://gee.cs.oswego.edu/dl/papers/fj.pdf, 2008.

[29] 林怀忠, 数据复制与一致性[J], 计算机工程与应用, 2001, (20): 107-108.

[30] 金煜利, 数据复制和数据一致性[J], 计算机科学, 1996, (3): 54-57.

[31] 王明钟, 在线数据复制系统的设计与实现[D], 陕西省 :西北工业大学, 2005.

[32] 程群梅, 动态 SQL 方法 4 在数据同步中的应用[J], 兵工自动化, 2004, (5): 42-42.

[33] 李新莉, 大批量数据出入数据库方法研究[J], 计算机光盘软件与应用, 2012, (4): 37-43.

[34] José, Enrique, Armendáriz-Iñigo, Trying to Cater for Replication Consistency and

Integrity of Highly Available Data[EB/OL], http://www.researchgate.net/publication/220271638_Trying_to_Cater_for_Replication_Consistency_and_Integrity_of_Highly_Available_Data, 2006.

[35] Chi, Zhang, rading replication consistency for performance and availability: an adaptive approach[EB/OL], http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1203520&filter%3DAND%28p_IS_Number%3A27093%29, 2003.

[36] Chun-Chen, Hsu, Optimal replication transition strategy in distributed hierarchical systems[EB/OL], http://www.iis.sinica.edu.tw/papers/cmwang/7418-F.pdf, 2008.

[37] Roberto, Baldoni, Impact of WAN Channel Behavior on End-to-end Latency of Replication Protocols[EB/OL], http://www.researchgate.net/publication/221246595_Impact_ofWAN_Channel_Behavior_on_End-to-end_Latency_of_Replication_Protocols, 2008.

[38] Wikipedia, Data recovery[EB/OL], https://en.wikipedia.org/wiki/Data_recovery.

[39] 郝之晓, 带窗口的净增量数据库复制优化算法[J], 计算机工程与科学, 2010, (2): 103-106.

[40] Google, Inc, Database Replication[P], USA: 8924347 B1,2012.

[41] Cellco, Partnership, Transitive Database Replication[P], USA: 8977591 B1,2011..

[42] Sybase, Inc, System Providing Methodology for Replication Subscription Resolution[P], USA: 7346633,2008.

[43] Sybase, Inc, Statement Logging in database[P], USA: 8676749,2014.

[44] Sybase, Inc, System, method and computer program product for determine SQL replication process[P], USA: 8751441,2014.

[45] Sybase, Inc, High Volume, High Speed Adaptive Data Replication[P], USA: 8996458,2015.

[46] Sybase, Inc, Statement Categorization and Normalization[P], USA: 12/183,794,2008.

[47] Heping, Shang, Non-disruptive data movement and mode rebalancing in extreme OLTP environments[P], USA: 12/970,219,2010.

[48] Heping, Shang, Distributed cache database architecture[P], USA: 13/091,303,2010..

[49] Heping, Shang, Hybrid data replication[P], USA: 13/362,892,2011.

[50] Heping, Shang, Directing a data replication environment through policy declaration[P], USA: 13/326,928,2011.

[51] Heping, Shang, Replication Description Model for data distribution[P], USA:

14/142,037,2013.

[52] Ye Guo Gang, Applying transaction log in parallel[P], USA: 14/290,421,2014.

[53] Ming-Chen, Lo, Ensuring the same completion status for transactions after recovery in a synchronous replication environment[P], USA: 14/290,421,2014.

[54] Xia, Ge, Dai, Zero Data Loss Transfer Protocol[P], USA: 2058.941US1,2014.

致谢

我的论文所属项目是在我工作期间的导师 Heping Shang 的精心指导和大力支持下完成的，他严谨的工作作风和丰富的数据库复制领域的实践经验给了我深深的启迪，论文中得点滴技术都凝聚了他的心血，在此我向他表示衷心的谢意。

此外，我也要感谢我的学业导师胡飞。他在百忙之中指导我完成这篇论文，他渊博的知识和开阔的视野给了我很多启发。他以严谨的治学态度和敬业精神感染了我，对我今后的工作和学习生活产生了深远的影响。

“明师之恩，诚为过于天地，重于父母”，对 Heping Shang 和胡飞的感激之情我无法用语言表达，在此向你们至以最崇高的敬意和最真诚的谢意！

最后，我也要感谢我在上海交通大学攻读在职硕士期间的各位老师和辅导员，是你们扶持我走向了人生新的高度，在此也向你们至以最崇高的敬意和最真诚的谢意！

攻读学位期间发表的学术论文目录

[1] 罗健, 一种并发式数据库复制技术的设计与实现[EB/OL], http://backup.se.sjtu.edu.cn/elearning/announcement/index.asp?range=all&courseid=1073&page=27, 2014-05-27.