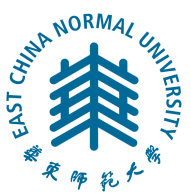
2019 届博士学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：52131500013



East China Normal University

博 士 学 位 论 文

DOCTORAL DISSERTATION

论文题目：
支持可扩展事务处理的数据库日志技术

院　　系：	数据科学与工程学院
专 业 名 称：	软件工程
研 究 方 向：	数据库系统
指 导 教 师：	钱卫宁 教授
学位申请人：	周欢

2019 年 5 月

Dissertation for doctor degree in 2019

University Code: 10269

Student ID: 52131500013

EAST CHINA NORMAL UNIVERSITY

Database Logging for Scalable Transaction Processing

Department:	School of Data Science and Engineering
Major:	Software Engineering
Research direction:	Database System
Supervisor:	Prof. Weining Qian
Candidate:	Huan Zhou

2019.05

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《支持可扩展事务处理的数据库日志技术》，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：周欢　　日期：2019年 5月 15日

华东师范大学学位论文著作权使用声明

《支持可扩展事务处理的数据库日志技术》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的研究成果归华东师范大学所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和相关机构如国家图书馆、中信所和“知网”送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

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

(√) 2. 不保密，适用上述授权。

导师签名：　　本人签名：周欢

2019 年 5 月 15 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）。

周欢 博士学位论文答辩委员会成员名单

姓名	职称	单位	备注
黄林鹏	教授	上海交通大学	主席
关佶红	教授	同济大学	
薛向阳	教授	复旦大学	
童维勤	教授	上海大学	
翁楚良	教授	华东师范大学	

摘　要

从上世纪七十年代以来，关系型数据库管理系统被广泛地运用于金融、交通、通讯等领域来高效地组织和管理数据。为了保证当出现软件和硬件故障时数据不会丢失，数据库系统实现了基于数据库日志的事务处理技术。数据库日志是一个存储所有事务执行结果的顺序文件，它由多条拥有全序关系的日志记录组成。为了保证数据库系统的可靠性和可用性，大多数传统数据库系统采用 ARIES 事务日志技术串行地将日志记录写入一个集中式日志缓冲区，然后再将日志缓冲区中的内容一起追加到存储在磁盘的日志中，最后使用日志复制技术将数据库日志通过以太网传输到远端的数据库系统副本。

在多核 CPU 和大容量内存的双重推动下，数据库系统实现了可扩展、高性能的事务处理技术来满足互联网企业的应用需求。然而，传统数据库日志技术的集中式设计、串行执行方式、顺序性约束以及磁盘网络 IO 操作限制了系统事务处理的性能。为此，本文实现了新型的事务日志技术和日志复制技术来满足事务处理系统可扩展、高性能、高通量的需求。本文的主要贡献总结如下：

1. **针对传统事务日志技术的集中式日志缓冲区竞争和固定组提交问题，本文提出了一种可扩展、自适应的集中式事务日志技术 Laser。**该技术实现了一种基于原子指令的日志号分配方法和一种并行的日志填充方法，从而提高了系统的可扩展性。结合动态变化负载，该技术首先利用生产者/消费者模型分析了负载变化对事务性能的影响，然后提出了一种负载自适应的组提交协议。此外，本文基于开源的分布式数据库系统 CEDAR 实现了可扩展、自适应的事务日志技术，并验证了该技术的可扩展性和高效性。

2. **针对传统事务日志技术的有限磁盘带宽问题，本文提出了一种面向可扩展存储的并行事务日志技术。**该技术使用多个日志缓冲区和多块磁盘来代替传统的集中式设计。事务根据负载感知的日志分区策略将日志记录均匀地分布到多块磁盘。这种日志分区策略避免了负载倾斜和跨分区事务对系统性能的影响。为了保证系统的正确性和可恢复性，该技术实现了一种偏序的日志号（GSN）和一种持久化的组提交协议。此外，本文结合并行事务日志技术、乐观多版本并发控制协议以及并行恢复方法，实现了一个内存事务处理原型系统（Plover），并在该系统上验证了事务日志技术的并行性和可扩展性。

3. **针对传统事务日志技术的顺序性约束问题，本文提出了一种支持系统可恢复**

性的偏序事务日志技术（Poplar）。该技术首先明确了事务日志技术在保证系统正确性和可恢复性的基础上需要具备的必要约束条件，然后给出了事务日志可恢复性的定义并验证了该定义的正确性。最后，基于可恢复性定义，该技术实现了一种可扩展的偏序日志号（SSN）和一种快速的事务提交协议。此外，本文基于开源的内存数据库原型系统 DBx1000 实现了支持系统可恢复性的偏序事务日志技术，并验证了该技术的并行性和高效性。

4. **针对传统日志复制技术的有限网络带宽问题，本文提出了一种面向主备复制系统的自适应日志复制技术**。该技术根据实时的应用负载选择传输日志记录或者传输数据增量给远端的系统副本。在处理高负载时，增量传输方法能够有效地减少网络传输量，从而避免网络成为系统的性能瓶颈。此外，本文结合并行事务日志技术和自适应日志复制技术，实现了一个高性能的主备复制内存数据库原型系统。为了保证系统主备副本之间的数据一致性，该技术实现了一种基于段的日志记录、数据增量合并算法和一种基于段的并行回放方法。最后，本文在复制系统上验证了自适应日志复制技术的有效性。

综上所述，事务日志技术和日志复制技术是保证数据库系统可靠性和可用性的重要手段。然而，在多核 CPU 和大容量内存硬件平台下，随着可扩展内存事务处理技术的不断发展，传统数据库日志技术成为了限制系统性能的主要瓶颈。本文发现传统数据库日志技术主要存在以下四个性能瓶颈：（1）集中式日志缓冲区竞争和固定组提交；（2）有限磁盘带宽；（3）事务日志的顺序性约束；（4）有限的网络带宽。为了解决这些瓶颈，本文分别展开了四项研究并提出了可扩展、自适应的集中式事务日志技术，面向可扩展存储的并行事务日志技术，支持系统可恢复性的偏序事务日志技术和面向主备复制系统的自适应日志复制技术。最后实验结果表明，这些新型的数据库日志技术实现了可扩展、高通量、低时延的事务处理性能。

关键词: 数据库管理系统，事务处理，事务日志，日志复制，并行日志

ABSTRACT

Since the 1970s, relational database management systems have been widely used in many fields, such as finance, transportation and communications, to efficiently organize and manage core data. To ensure that data is not lost in the event of software and hardware failures, database systems typically rely on the use of a database log. The database log is a sequential file that stores information about transactions and the state of the system in certain instances. Each entry in the database log is called a log record which is assigned with a unique and monotonically increasing log sequence number (LSN). To ensure the reliability and availability, traditional database systems use the ARIES transaction logging to write log records into a central log buffer and flush them into a disk, and then use the log replication to transfer log records to the remote backups.

However, with the emergence of multi-core and large memory, centralized design, serial execution, sequential constraint and IO operation of traditional database logging have become the main performance bottlenecks of scalable on-line transaction processing systems. To this end, this paper implements new transaction logging and log replication to achieve high-performance and scalability of database systems. The main contributions are summarized as follows:

1. **For the centralized log buffer contention and fixed group commit of traditional transaction logging , this paper presents a scalable and load-adaptive transaction logging (Laser).** It uses an LSN calculation based on atomic instructions and a parallel log insertion to improve the scalability of database systems. In order to get the lowest commit latency of transactions in changing workloads, it proposes a load-adaptive group commit protocol to dynamically determine an optimized group time for different workloads. In addition, this paper implements Laser in the open-source distributed database system CEDAR and evaluates its performance.

2. **For the limited disk bandwidth of traditional transaction logging, this paper proposes a parallel transaction logging on scalable storage devices.** It uses multiple log buffers and disks instead of centralized design. To enable parallel logging, it proposes a global sequence number (GSN) that provides a partial order of log records and a persistent group commit protocol. To further alleviate the performance overheads caused by log partitioning, it proposes a workload-aware log partitioning to minimize the number of cross-partition transactions, while maintaining load balance. In addition, this paper implements an in-memory transaction engine (Plover) with parallel logging, optimistic concurrency control protocol and parallel recovery. Experiments demonstrate its parallelism and scalability.

3. **For the sequential constant of traditional transaction logging, this paper proposes a recoverable and partial transaction logging (Poplar).** It defines recoverability for transaction logging and demonstrates its correctness for crash recovery. Based on recoverability, it enables log records to persist on multiple disks in parallel, proposes a scalable log sequence number (SSN) to track RAW and WAW dependencies between transactions, and implements a speedy transaction commit protocol. In addition, this paper implements Poplar in the open-source in-memory database system DBx1000 and compares it with other transaction loggings.

4. **For the limited network bandwidth of traditional log replication, this paper proposes an adaptive log replication for hot standby systems.** It uses an adaptive shipping method to reduce the network traffic and avoid the network becoming a bottleneck under a heavy workload. In addition, this paper implements a high-performance in-memory replication system with adaptive log replication and parallel transaction logging. It uses a segment-based algorithm and a segment-based replay to ensure the consistency between master and backups. Experiments verify its effectiveness.

In summary, transaction logging and log replication are essential constituents to guarantee the reliability and availability of database systems. However, traditional database logging becomes a major performance bottleneck in the multi-core and large-scale memory platforms. There are four key problems: (1) centralized log buffer contention and fixed group commit protocol; (2) limited disk bandwidth; (3) sequential constraint; (4) limited network bandwidth. In order to solve these bottlenecks, this paper launches four types of researches and proposes four databases logging, e.g., a scalable and adaptive transaction logging, a parallel transaction logging on scalable storage devices, a recoverable and partial transaction logging and an adaptive log replication for hot standby systems. Finally, the experimental results show that new database loggings enable scalable, high-throughput and low-latency transaction processing.

Keywords: *Database Management System,Transaction Processing, Transaction Logging, Log Replication, Parallel Logging*

目录

插图

表格

第一章　绪论

1.1　研究背景与意义

传统企业，如银行、航空公司、零售商等，主要通过将重要的数据，如账目记录、销售记录，存放在关系型数据库管理系统（RDBMS）中来提供正确、可靠的服务。在许多情况下，它们还需要提供 7×24 小时的不间断服务。为此，DBMS 需要具备两个核心功能：（1）保证大量并发的用户能够准确、快速地访问数据；（2）保证数据不会丢失。这两个功能主要依赖于关系型数据库系统的事务处理技术 [1]。事务是关系型数据库系统的一个逻辑处理单元，它由一系列对数据的读/写操作组成，并且具备四个重要属性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability） [2]。银行的每一次转账交易可以称为一个事务，它主要包含两个数据库操作：从 A 账户的余额中扣除一部分金额，然后将扣除的金额存入 B 账户。事务的原子性要求扣除和存入两个操作要么都发生，要么都不发生。两个操作都发生的事务叫做已提交事务，其它的称为未提交事务。事务的一致性要求从 A 账户中扣除的金额不能超过他的总余额。事务的隔离性要求多个并发执行的转账事务之间不会有任何影响。事务的持久性要求已提交转账事务对数据库的修改必须永久地保存下来。严格的事务语义保证了关系型数据库系统的完整性、一致性、并发性和可恢复性。

传统数据库系统是基于单核 CPU，磁盘和内存两层存储层级进行设计的。数据库系统使用大容量、非易失、廉价的磁盘作为主存储介质来永久性地存储数据，而将低容量、易失、高速读写的内存作为缓存来处理事务逻辑。由于硬件和软件存在天然的不稳定性，因此数据库管理系统可能遇到各种软件和硬件故障。数据库系统遭遇的故障主要分为三种：事务故障、系统故障和硬件媒介故障 [3]。当事务因冲突或者因用户强制终止服务而中断时会发生事务故障；当操作系统崩溃或者内存损坏时会发生系统故障；当自然灾害（火灾、地震等）导致磁盘损坏时会发生

硬件媒介故障。为了保证在发生故障时，系统能够恢复到一个正确的状态且继续对外提供服务，数据库系统实现了基于数据库日志的恢复方法 [4]。

数据库日志 [5] 是一个存储所有事务执行结果的顺序文件，它由多条日志记录组成并保存在非易失的磁盘中。每条日志记录存储了事务的一次更新内容和一个满足全局单调递增关系的日志序列号（LSN）。LSN 指向日志记录在日志文件中的位置并表示事务执行的先后顺序。为了保证当发生硬件媒介故障时数据不会丢失，数据库管理系统冗余地将日志存储在多个系统副本中。这种系统模式被称为主备复制系统 [3]，它由一个主副本和多个备副本组成，其中主副本负责用户的事务请求。在事务执行过程中，传统数据库管理系统采用 ARIES 事务日志技术 [6] 将日志记录按 LSN 顺序串行地写入主副本的磁盘，然后使用日志复制技术将日志记录串行地复制到异地的备副本。当主副本发生事务故障和系统故障时，数据库系统按 LSN 顺序恢复主副本中的日志记录，然后由主副本继续对外提供服务。当主副本发生硬件媒介故障时，数据库系统恢复备副本中的日志记录，然后由备副本替代主副本对外提供服务。因此，基于数据库日志的事务日志技术和日志复制技术保障了关系型数据库管理系统的可靠性和可用性。

近年来，随着宽带、无线 4G 和移动终端的普及和快速发展，互联网企业不断地向金融、零售、交通、餐饮等传统行业渗透。在“互联网 +”时代，用户可以随时随地、简单地通过移动 APP 进行在线购物、订票、订餐和支付。便捷的生活方式不仅导致了互联网用户数量的急剧增长，还使得企业面临了大量并发的事务处理请求。中国互联网络信息中心（CNNIC）发布的《第 43 次中国互联网络发展情况统计报告》[7] 中指出，截至 2018 年 12 月，中国网络用户规模为 8.29 亿，较 2008 年底增长了近 37%。据新华网报道 [8]，在 2018 天猫双 11 狂欢节当日，阿里巴巴平台的成交额高达 2135 亿元人民币，仅 2 分 05 秒的时候成交额就突破了 100 亿人民币，每秒的用户请求高达 17.18 亿。面对如此庞大的数据规模和高通量的事务处理请求，以磁盘为主存储介质并且依赖单核处理器设计的传统关系型数据库系统可谓是心有余而力不足。无论采用分库分表的方式还是系统进行垂直扩展的方式，

传统数据库系统的事务处理能力都无法满足互联网业务的需求。

然而，硬件技术的快速发展使得实现可扩展、高性能的事务处理能力已不再是痴心妄想。一方面，随着内存的容量不断增加同时价格不断降低，数据库系统由以磁盘为主存储介质转向以内存为主存储介质 [9]。数据库系统直接将所有数据保存在内存中，而磁盘只作为一种线下的备份设备。与传统的数据库系统相比，面向内存的数据库系统消除了低速磁盘读写与高速 CPU 计算之间的性能差和磁盘写入放大等问题，从而避免了事务执行过程中 85% 的额外开销 [10]。另一方面，随着处理器核数的增多，数据库系统由原来面向单核的串行事务执行方式转为面向多核的并行执行方式。这种转变使得数据库系统的事务处理得到了近乎线性地提升。

在应用需求和硬件技术发展的双重推动下，实现可扩展、高性能的数据库管理系统，尤其是在线事务处理系统（OLTP）成为了学术界和工业界共同关注的问题。大量高性能的商业数据库系统和事务处理原型系统如雨后春笋般涌现出来，如 OceanBase 0.2 [11]、MySQL 8.0 [12]、PostgreSQL 11.2 [13]、TiDB [14]、Microsoft Hekaton [15]、CEDAR [16]、Solar [17]、H-Store/VoltDB [18, 19]、Deuteronomy [20]、ERMIA [21]、DBx1000 [22]、IBM SolidDB [23]、Oracle TimesTen [24, 25] 等。这些新型数据库系统要么基于存储和计算分离的架构 [26] 实现了可扩展的数据存储；要么基于内存存储实现了高性能的事务处理技术；要么基于多核 CPU 实现了可扩展的事务处理技术，如确定性事务执行技术 [27]、轻量级的乐观并发控制协议 [28–30]、高性能的内存索引技术 [31, 32] 等，从而大幅度地提升了事务的处理性能。但是，为了保证数据的可靠性和可用性，大多数新型数据库系统仍然采用传统的基于磁盘的先写事务日志技术和基于日志传输的复制技术。由于它们使用串行的执行方式来保证数据库日志的严格顺序性，并且依赖磁盘 IO、网络 IO 操作来保证事务的持久性，因此随着数据库系统其它模块的可扩展和性能提升，传统的数据库日志技术成为了限制数据库系统事务处理性能的最主要瓶颈。因此，研究、设计、实现一种新型的事务日志技术和日志复制技术来支持数据库系统高性能和可扩展的事务处理能力是非常必要的。

1.2 国内外研究现状

为了保证数据的可靠性和可用性，关系型数据库管理系统实现了基于先写日志协议（Write-Ahead Logging，WAL）的集中式事务日志技术和基于日志传输的复制技术。接下来，本节首先详细地分析了两种数据库日志技术在支持可扩展事务处理的数据库系统中存在的性能瓶颈，然后总结了当前国内外的研究现状。

1.2.1 事务日志技术

为了使得在发生系统故障之后数据不会丢失并且系统能够恢复到一个正确的状态，数据库系统采用了一种 ARIES 事务日志技术 [6] 来保证事务的原子性和持久性。ARIES 是一种基于先写日志协议的集中式事务日志技术，它最早由 IBM 提出，之后实现在 System R [33] 数据库系统中。ARIES 采用逻辑物理日志记录来存储事务的修改信息。每条日志记录存储了一个全局唯一的日志序列号（LSN），它遵循全局单调递增的关系并且保证了事务的执行顺序。ARIES 首先按 LSN 顺序将日志记录临时缓存在一个位于内存的集中式日志缓冲区，然后再顺序地把日志记录写入存储在磁盘中的日志文件。在事务执行过程中，事务首先获取数据项上的读写锁并执行事务逻辑，然后进入事务日志处理流程。结合先写日志协议，ARIES 的日志流程主要包括以下四个步骤：（1）LSN 分配：事务首先获得集中式日志缓冲区上的排他锁，然后为每条日志记录分配一个 LSN（日志序列号表示日志记录在缓冲区中的存储位置）；（2）日志填充：事务首先将日志记录写入集中式日志缓冲区，填充完成之后再释放集中式日志缓冲区上的排他锁；（3）日志持久化：事务将缓冲区中的日志记录追加到存储在磁盘中（如机械硬盘 HDD）的日志文件；（4）事务提交：等到事务的所有日志记录都已写入磁盘之后，事务才能提交。在执行提交时，事务先释放更新数据项上的写锁，然后响应客户端。

性能瓶颈。随着多核 CPU 和大容量内存的出现，基于集中式设计并且涉及磁盘 IO 的 ARIES 事务日志技术成为了限制可扩展事务处理系统性能的主要瓶颈。通过对 ARIES 的执行流程分析，本文发现它主要存在以下八个性能问题。图 1.1给出

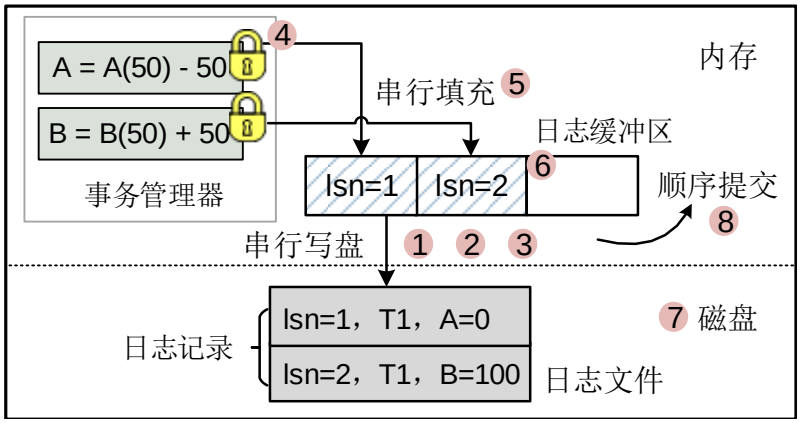


图 1.1: 传统 ARIES 事务日志存在的性能瓶颈

了八个性能瓶颈在事务日志执行流程中的具体位置。

1. **日志持久化时延**。先写日志协议规定只有当事务的所有日志记录都已持久化之后事务才能提交。对于部署在机械硬盘 HDD 上的数据库系统来说，一次日志持久化操作需要几十甚至几百毫秒。即使部署在固态硬盘 SSD 上，一次持久化操作也需要几百微秒甚至几毫秒。相比于内存纳秒级的读写性能来说，日志持久化成为了事务执行过程中耗时最长的部分。大量、频繁的 IO 操作会增加日志的持久化时延，从而大幅度降低了数据库系统的事务吞吐量。

2. **固定组提交**。为了降低事务日志持久化的平均时延，ARIES 实现了基于固定触发条件的组提交协议（group commit）[34]，即每隔固定时间或者每当缓冲区中积累的日志记录长度超过一定值时，数据库系统执行一次批量写盘。等日志记录都已持久化之后，对应的事务才能提交。然而，固定的组提交协议没有考虑真实的应用负载。以固定时间的组提交为例，当处理低负载的应用时，太长的组提交时间会导致大量事务处于等待持久化的状态；当处理高负载的应用时，太短的组提交时间又将导致事务因过载的磁盘 IO 而被阻塞。因此，固定的组提交协议不适用于动态变化的应用负载。

3. **上下文切换**。ARIES 采用了一种非流水线式的执行模式，即由一种线程（这里称为工作线程）负责所有的事务日志操作，包括获取 LSN、日志填充、日

志持久化以及事务提交。在日志持久化阶段，工作线程需要在用户态和内核态之间进行切换。一次上下文切换需要执行上千条指令，这将导致额外的操作系统调度开销。随着 CPU 核数的增多，大量的上下文切换将导致大量的 CPU 资源浪费，从而影响系统性能。

4. **日志引起的事务冲突**。结合先写日志（WAL）协议，事务需要在所有日志记录都已持久化之后才能释放更新数据项上的写锁。长时间的持有数据项写锁将加剧事务并发层面的冲突，从而降低整体的事务处理性能。

5. **串行化填充**。ARIES 采用了一种串行的执行方式来填充日志记录，即事务需要等待前一个事务的日志填充完成之后才能开始填充。这种方式使得事务日志成为数据库系统的一个全局串行点，从而严重地限制了系统的可扩展性。

6. **集中式缓冲区竞争**。在日志填充之前，事务需要依赖集中式日志缓冲区上的写锁来分配 LSN 和日志记录在缓冲区上的位置。当大量事务试图同时填充日志记录时，集中式日志缓冲区上的锁竞争将变得异常的激烈，从而导致大量事务失败然后重试。随着 CPU 核数的增多，集中式缓冲区上的竞争将成为限制系统可扩展性的首要瓶颈。

7. **有限 IO 带宽**。在 ARIES 事务日志流程中，日志记录需要被串行地写入一块磁盘。磁盘的顺序写性能决定了数据库系统的峰值吞吐量。随着 CPU 核数越来越多，数据库系统每秒能产生超过 1.1GB 的 ARIES 日志量 [15, 20]。然而，基于 SATA 接口的机械硬盘以及固态硬盘每秒大约只能处理几百兆的顺序写入请求。因此单磁盘有限的 IO 带宽限制了系统的最高吞吐量。

8. **顺序性约束**。ARIES 要求事务的日志序列号（LSN）、日志填充顺序、日志持久化顺序以及事务提交顺序遵循一个相同的全序关系，即 LSN 保证全局单调递增，日志填充、日志持久化以及事务提交操作全都按照 LSN 顺序串行地执行。如此强的顺序性约束限制了事务日志的可扩展性。

表 1.1: 事务日志技术的国内外研究现状

事务日志技术	事务日志瓶颈					技术的不足之处
	串行化填充	集中式缓冲区竞争	固定组提交	有限 IO 带宽	顺序性约束	
ARIES [6]						存在所有瓶颈
Aether [35]	✓					基于锁获取 LSN
H-Store [18]	✓	✓		✓		恢复性能极差
OceanBase [11]		✓				单线程处理日志
Hekaton [15] Deuteronomy [20] ERMIA [21] ELEDA [36]	✓	✓				不适用于动态变化的应用负载
Laser *	✓	✓	✓			
Silo [37]	✓	✓		✓		依赖特定的 OCC
NV-Logging [38]	✓	✓		✓		依赖 NVM，频繁 IO
NVM-D [39]	✓	✓		✓		依赖 NVM，频繁 IO
Plover*	✓	✓	✓	✓		
Poplar*	✓	✓	✓	✓	✓	

✓表示事务日志技术已解决性能瓶颈；* 表示本文提出的事务日志技术

目前，国内外研究工作提出了许多新的事务日志技术来解决传统 ARIES 技术存在的性能瓶颈。表 1.1总结了当前事务日志技术的国内外研究现状。由于当前所有事务日志技术都已解决了 ARIES 存在的日志持久化时延、上下文切换以及日志引起的事务冲突瓶颈，因此表中没有给出它们的比较结果。根据不同的日志持久化方式，本文将当前的事务日志技术分为集中式事务日志技术和并行事务日志技术两类，然后分别总结了它们存在的优缺点。

在集中式事务日志技术中，Aether 采用并行日志填充方式解决了传统 ARIES 因串行日志填充导致的可扩展性问题。但由于它仍使用基于锁的方式来获取日志序列号（LSN），因此集中式日志缓冲区上的锁竞争仍然存在。H-Store [18] 实现了一种基于 Command Logging 的事务日志技术。该技术使用一种事务逻辑日志来代替传统 ARIES 的逻辑物理日志。由于 Command Logging 每次产生的事务日志量远远小于 ARIES 日志量，因此它不仅降低了日志持久化的时延，还避免了单磁盘 IO 带宽成为系统的瓶颈。然而，当发生系统故障时，H-Store 必须串行地恢复日志记录，因此它的恢复性能特别差。为了实现一个通用、高性能的事务日志技术，OceanBase 使用一个单独的线程（日志线程）来处理所有的事务日志操作。虽然它能够避免集中式日志缓冲区的竞争，但是大量的内存拷贝操作以及上下文切换使

得单线程的处理能力成为了事务日志技术的新瓶颈。为了提高事务日志的多核可扩展性并且不产生额外的性能问题，Hekaton [15]、Deuteronomy [20]、ERMIA [21] 和 ELEDA [36] 采用原子指令（fetch-add）来分配日志序列号。基于 latch-free 的 LSN 分配方法极大地减轻了集中式日志缓冲区的竞争，从而实现了较好的事务处理性能。虽然现有的集中式事务日志技术提高了系统事务处理的可扩展性，但是它们仍然没有解决传统 ARIES 存在的固定组提交问题，因此它们不适用于动态变化的应用负载。为此，本文提出了一种可扩展、自适应的集中式事务日志技术 Laser。

在并行事务日志技术中，Silo 使用多个日志缓冲区和多块磁盘来提高事务日志的并行度，从而避免了单块磁盘的 IO 带宽限制系统的整体性能。虽然 Silo 获得了可观的事务日志性能，但是由于它依赖于一种基于 epoch 的乐观并发控制协议，因此它并不是一种通用的日志技术并且不易移植到其他数据库系统。NV-Logging [38] 和 NVM-D [39] 是基于新型非易失内存设备（Non-Volatile Memory ）提出的并行事务日志技术。得益于 NVM 的可持久性和低延迟随机读写能力，该技术消除了日志持久化时延和有限 IO 带宽瓶颈。但是，在事务日志执行过程中，它们允许工作线程直接持久化事务日志记录。大量的 IO 写盘操作严重影响了系统的整体性能。此外，这种非流水线式的事务日志技术不适用于面向磁盘的数据库系统。为了实现一个更加通用且易移植的解决方法，本文提出了一种既适用于磁盘又适用于 NVM 的并行事务日志技术。到目前为止，所有的事务日志技术都没有解决 ARIES 存在的顺序性约束问题。为此，本文在保障数据库系统可恢复性的基础上分析了事务日志技术需要的必要约束，并提出了一种可恢复的事务日志技术 Poplar。

1.2.2 日志复制技术

为了使得在发生硬件媒介故障之后数据不会丢失且系统能够迅速地提供服务，一个最简单的方法是在多地部署相同的数据库系统副本。这种模式被称为主备复制系统，也叫热备系统 [2]。它被广泛地运用于商业数据库系统来保证 7×24 小时的可用服务，如 SQL Server [40]、DB2 [41]、MySQL [42, 43] 和 PostgreSQL [44, 45]。

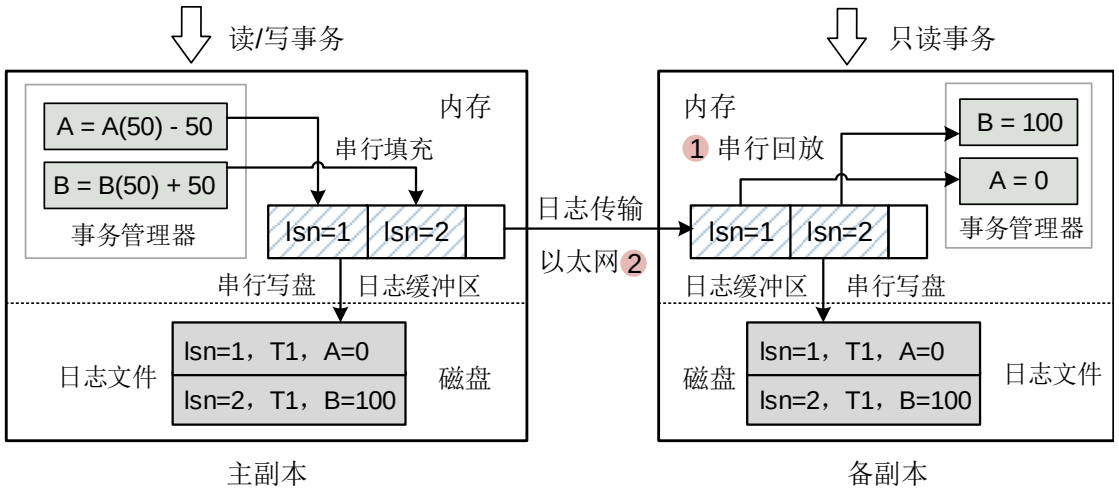


图 1.2: 传统主备日志复制存在的性能瓶颈

主备复制系统通常由一个主副本和多个备副本组成，主备副本之间采用以太网进行连接。主副本主要负责处理用户的读写事务请求而备副本提供只读服务。为了保证主备副本之间的数据一致性，热备系统通常采用基于日志传输（Log Shipping）的复制技术 [4, 46]，即主副本将事务执行结果以日志记录的形式存储下来，然后按事务执行顺序传输给备副本。备副本接收到复制信息之后，按相同的事务顺序进行回放。当主副本发生硬件媒介故障时，备副本可以成为新的主副本对外提供服务。一般而言，备副本对外提供了通用的快照读服务（GSI） [47]。GSI 是对快照隔离级别 [48] 的一种扩展，它允许只读事务读取备副本上相对比较旧的数据库快照。

结合事务日志技术，主备日志复制技术的执行流程主要分为以下阶段：（1）主副本事务执行：主副本首先执行事务逻辑，然后产生日志记录并将日志记录串行地写入集中式日志缓冲区和磁盘；（2）日志传输：主副本将日志记录按日志序列号（LSN）串行地传输给备副本；（3）备副本回放：备副本接收到日志记录之后，首先将日志记录写入集中式日志缓冲区和磁盘，然后响应主副本。之后备副本再按 LSN 顺序串行地回放日志记录；（4）主副本事务提交：主副本有两种提交模式，当系统采用同步日志传输方法时，事务需要等到日志记录写入主备副本的磁盘之后才能提交。当系统采用异步日志传输方法时，事务只需要等日志记录持久化到主副本的磁盘之后就能提交。基于同步日志传输方法的主备复制系统保证了数据的可

靠性。但是，由于事务提交涉及网络 IO 传输和备副本的磁盘 IO 操作，因此系统的整体性能比较差。相反，基于异步日志传输方法的系统具有较好的事务处理性能，但是它却面临了数据丢失的风险。在处理核心业务时（转账业务），传统企业通常部署了基于同步日志传输方法的主备复制系统。

性能瓶颈。在多核 CPU 硬件平台下，随着主副本的事务处理能力越来越高，传统基于日志传输方法的复制技术存在以下两个性能问题。图 1.2显示了两个瓶颈在主备日志复制流程中的具体位置。

1. **串行回放。**为了保证主备副本之间的数据一致性，备副本通常采用单线程串行地回放日志记录 [49]。串行回放方式使得备副本的事务处理能力远远低于主副本。随着主副本事务处理能力的提升，主备副本之间的数据版本差逐渐增大 [50]，甚至出现备副本永远追不上主副本的情况。这将导致系统不得不面对承担丢失大量数据的风险或者降低主副本事务处理性能的艰难抉择。

2. **有限的网络带宽。**主备日志复制技术需要基于网络来传输事务日志记录，从而保证数据库系统的可用性。网络的传输能力决定了主备复制系统的最大吞吐量。随着主副本事务处理能力的提升，数据库系统每秒可以产生大于 1.5GB 的 ARIES 日志量 [21, 51]。然而，传统以太网，如千兆网、万兆网，每秒最多能传输 1.12GB 的数据。因此有限的网络带宽限制了系统的事务处理性能。

表 1.2: 主备日志复制技术的国内外研究现状

日志复制技术	日志复制瓶颈		技术的不足之处
	串行回放	有限网络带宽	
Log shipping			存在所有瓶颈
KuaFu [49]	✓		依赖追踪代价高
H-Store [18]		✓	回放性能太差
Calvin [27]		✓	依赖确定性事务执行技术
OceanBase [11]	✓		网络带宽瓶颈
Scalable replication [52]	✓	✓	依赖确定性事务执行技术
Query Fresh [53]	✓	✓	依赖 RDMA 和 NVM
Plover*	✓	✓	

✓表示日志复制技术已解决性能瓶颈；* 表示本文提出的日志复制技术

最近的研究工作提出了一些新的主备日志复制技术。表 1.2给出了近几年国内

外的研究情况。KuaFu [49] 实现了一种基于事务依赖追踪的并行回放方法来提高备副本的事务处理性能。但是由于它采用单线程来构建事务依赖图，因此依赖追踪成为了系统新的性能瓶颈。基于 Command Logging 实现的 H-Store [18] 减少了主副本每秒产生的日志量，从而避免了有限的网络 IO 带宽成为瓶颈。但是，H-Store 的备副本仍然需要串行地回放日志记录，因此主备副本之间的数据版本差仍然存在。Calvin [27] 采用了一种事务逻辑日志（只记录事务的读写集合）来减少网络传输量，从而解决了有限网络 IO 带宽问题。但是，这种逻辑日志只适用基于确定性事务执行技术的数据库系统。此外，备副本仍需要串行地回放日志记录来保证主备副本之间的数据一致性。可扩展的日志复制技术 [52] 和 Query Fresh [53] 解决了传统日志复制技术的所有瓶颈。但是，这些技术要么依赖确定性事务执行要么依赖于新型硬件设备（远程内存直接访问 RDMA 和非易失内存 NVM），因此它们不能被广泛地运用到商业数据库系统。为此，本文实现了一种基于自适应日志传输方法的通用日志复制技术。

1.3 研究内容与挑战

尽管事务日志技术和日志复制技术均得到了较大的发展，但是这些技术仍然存在一些不足之处，例如，现有可扩展的集中式事务日志技术在动态变化负载中的性能表现不稳定；面向新型非易失内存设备的并行事务日志技术不适用于部署在磁盘上的数据库系统；所有事务日志技术依然遵循严格的顺序性约束；依赖逻辑日志和新型硬件 RDMA 的日志复制技术不适用于现有的商用数据库系统。为了解决现有事务日志和日志复制技术存在的问题，实现一种通用的数据库日志技术来满足在线事务处理系统的高性能和可扩展的需求，本文基于一个主备复制数据库系统展开了以下四项研究：

1. **结合动态负载的可扩展事务日志问题。**随着多核 CPU 技术的发展，数据库系统采用数据/负载分区、事务预编译、内存高效计算等方法有效地降低了事务并发执行的代价。然而，作为数据库系统持久性和可恢复性的保障，传统事

务日志技术仍然采用集中式设计、串行执行方式和基于磁盘存储来保证日志记录的可靠性和顺序性。事务日志技术的磁盘 IO 操作、缓冲区竞争以及串行执行方式严重地限制了数据库系统的事务处理性能。

为了解决传统事务日志的性能瓶颈，提高事务处理的可扩展性，最行之有效的方法是允许多个事务并行地执行事务日志操作以及批量地将日志记录写入磁盘。一方面，采用 latch-free 的日志序列号分配方法和并行日志填充方式会不可避免地产生日志空洞，即一段连续的日志缓冲区空间中会出现部分填充的日志记录。如果将存在日志空洞的日志记录写入磁盘，那么数据库系统将面临丢失数据的风险。因此，如何高效、正确地检测日志空洞是实现可扩展事务日志技术最大的挑战。另一方面，传统采用固定组提交的批量持久化方法没有考虑动态变化的负载。当处理低负载应用时，如果组提交条件过长，已完成日志填充的事务不得不等待组提交条件触发之后才能持久化和提交；当处理高负载应用时，如果组提交条件过短，数据库系统将产生大量的磁盘 IO 操作从而导致事务被阻塞。如何选择一个合适的模型来模拟、分析负载与事务日志的关系，并决策出一个负载自适应的组提交条件，是实现稳定、高性能的事务日志技术最大的难点。因此，如何结合动态变化负载实现一个可扩展、高性能的集中式事务日志技术是本项研究最主要的内容和挑战。

2. **高速日志数据流的并行处理问题**。日志记录是一种高速的数据流，尤其在可扩展事务日志技术中，数据库系统每秒几乎能产生超出 1.1GB 的日志量 [21]。为了保证数据的可靠性，事务的日志记录需要持久化到磁盘。面对如此高速的日志量，部署在一块磁盘上的数据库系统只能鞭长莫及。为了提高数据库系统的最大吞吐量，最简单却也最有效的方法是将事务日志记录并行地写入多块存储设备。然而，为了保证系统的可恢复性和正确性，日志记录需要保证多个事务在同一数据项上的先后执行顺序，以及同一事务内读写操作的先后顺序。在传统集中式事务日志技术中，事务的执行顺序可以直接用日志序列号 LSN 来追踪。但在并行持久化方法中，修改同一数据项的事务日志记

录或者来自同一事务的日志记录可能分布在不同的磁盘上。因此，如何高效地追踪事务之间的顺序依赖是实现并行事务日志技术最大的挑战。此外，先写日志（WAL）协议规定，只有当事务的所有日志记录都已持久化之后，事务才能提交。但在并行事务日志技术中，事务还需要等存在依赖关系的事务提交之后才能提交。例如，两个存在读写依赖的事务，事务 T_i 先修改了数据项 a，之后 T_j 读到了 T_i 在 a 上的修改。假设 T_j 的日志记录已持久化而 T_i 未持久化。如果此时数据库系统发生故障，在恢复过程中 T_j 会被恢复而 T_i 不会，那么恢复之后的数据库系统处于一个不正确的状态——T_j 读到了一个不存在的值。因此，如何追踪事务的持久化情况并保证事务正确提交是并行事务日志技术另一个重要的研究内容和挑战。

3. **事务日志顺序性约束问题。**为了保证系统在发生故障之后能够恢复到一个正确的状态，事务日志技术不但要具有可靠性，还需要保证事务之间的正确执行顺序。为此，传统集中式事务日志技术采用全序的日志序列号 LSN 来追踪事务之间的执行顺序，并规定日志填充、日志持久化以及事务提交都必须遵循和 LSN 一样的全序性。顺序性约束使得事务日志的所有操作，如日志序列号分配、日志填充、日志持久化和事务提交必须串行地执行，这种执行方式严重地限制了系统的可扩展性。事实上，事务日志技术并不需要遵循如此强的顺序性约束就足以保证数据库系统的可恢复性和正确性。尤其是对于没有任何依赖关系的事务来说，它们在事务日志执行流程中的任何乱序操作都不会对数据库系统产生任何的负面影响。放松事务日志技术的约束条件可以进一步提高数据库系统的并行性。然而，当前所有的研究工作都没有解决事务日志的顺序性约束以及给出事务日志并行执行的理论依据。因此，明确事务日志技术在保证系统正确性和可恢复性的基础上所需的必要约束条件是一项值得研究且富有挑战的问题。

4. **面向高通量事务日志的主备复制问题。**为了保证在发生硬件媒介故障时数据的可用性，数据库系统通常采用基于主备系统的日志复制技术。系统主副本

将产生的日志记录串行地传输给系统备副本。随着主副本并行事务日志技术的发展，数据库系统每秒产生的网络传输量远远超出了以太网的传输能力，从而导致有限的以太网带宽成为了主备系统的性能瓶颈。为此，结合并行事务日志技术，实现能处理高通量日志数据流的主备复制技术是本项研究的主要内容。为了保证主备副本之间的数据一致性，主副本需要传输一个完整、一致性的数据库快照给备副本。如何结合并行事务日志技术上形成一个全序数据库快照是本项研究最主要的技术难点。

1.4 主要贡献

为了使得数据库系统具有可扩展的事务处理能力来应对互联网企业的应用需求，本文主要研究了保证系统可靠性和可用性的数据库日志技术。本文发现在多核 CPU 平台下，传统集中式事务日志技术和基于日志传输方法的主备复制技术还存在以下性能瓶颈：（1）集中式日志缓冲区竞争；（2）固定组提交问题；（3）有限的磁盘 IO 带宽；（4）事务日志的顺序性约束问题；（5）主备日志复制中的有限网络 IO 带宽。为了解决这些瓶颈问题，本文实现了可扩展、自适应、高性能的事务日志技术和日志复制技术。主要的贡献主要包括以下四点：

1. 针对集中式日志缓冲区竞争和固定组提交问题，本文提出了一种可扩展、自适应的集中式事务日志技术 Laser。该技术实现了基于原子指令的日志号分配方法以及并行的日志填充方法，该方法提高了事务处理的可扩展性。然而，并行日志填充方法会导致日志空洞，即日志缓冲区中存在部分填充的日志记录。如果将存在日志空洞的日志记录写入磁盘，那么数据库系统会面临丢失数据的风险。为了解决这个问题，该技术实现了基于日志组的空洞检测方法。空洞检测方法首先动态地将日志缓冲区中的日志记录划分成多个大小不一的日志组，然后将日志组串行地写入磁盘。该方法规定：当且仅当每个日志组里的所有日志记录都完成填充之后，日志组里的记录才能被持久化。为了保证数据库系统能够在动态变化负载中一直保持稳定的事务处理性能，该技

术首先使用生产者/消费者模型分析了应用负载和事务处理性能之间的关系，然后提出了一种负载自适应的组提交协议。此外，本文基于开源数据库系统CEDAR 实现了可扩展、自适应的事务日志技术并对其进行了性能评估。实验结果表明，Laser 具有良好的多核可扩展性，并且能够在动态变化负载中获得最大的事务吞吐量和最小事务提交时延。

2. 针对有限的磁盘 IO 带宽问题，本文提出了一种面向可扩展存储的并行事务日志技术。该技术采用多个日志缓冲区和多块磁盘来并行地存储日志记录。日志缓冲区和磁盘之间存在一对一的关系。每个日志缓冲区还对应一个专属的日志线程，日志线程负责将缓冲区中的日志记录写入对应的磁盘。由于日志记录分布在多个日志缓冲区中，传统的日志序列号（LSN）无法追踪所有日志记录之间的事务依赖，因此该技术实现了一种新的事务日志号（GSN）。GSN 采用去中心化的方式进行分配，它保证了事务之间的偏序关系，即事务之间的先写后读依赖（RAW）、写后写依赖（WAW）和先读后写依赖（WAR）。为了保证事务提交的正确性，该技术实现了一种持久化的组提交协议。该协议规定：当且仅当事务的所有日志记录以及它所依赖事务的日志记录都已持久化之后，该事务才能被提交。为了便于 GSN 计算，并行事务日志技术实现了一种基于数据的日志分区策略，即修改同一数据项的事务日志记录存储在相同的日志缓冲区和磁盘中。但是，这种分区策略会使得系统性能受到不均匀负载的影响。为了解决这个问题，该技术提出了一种负载感知的日志分区策略。它能够根据实时的应用负载产生一个合适的分区策略，从而使得日志记录能够均匀地分布到多块磁盘。此外，本文结合并行事务日志技术、乐观并发控制协议以及并行恢复方法，实现了一个内存事务处理原型系统（Plover）并对其进行了性能评估。实验结果表明，并行事务日志技术的性能能够随着磁盘数量的增加呈线性扩展。

3. 针对事务日志的顺序性约束问题，本文提出了一种支持系统可恢复性的偏序事务日志技术（Poplar）。该技术首先明确了事务日志在保证系统可恢复性和

正确性的基础上需要具备的必要约束条件，然后给出了一个事务日志可恢复性的定义并验证了它的正确性。可恢复性规定：事务日志技术只需要保证事务的提交顺序遵循事务之间的 RAW 依赖，并且事务的日志号遵循事务之间的 WAW 依赖。事务日志的可恢复性彻底地打破了传统集中式事务日志技术的顺序性约束，并为实现并行事务日志技术提供了理论依据。基于恢复性定义，该技术允许事务并行地将日志记录写入多个日志缓冲区和多块磁盘，同时实现了一种可扩展的偏序事务日志号（SSN）和快速的事务提交协议。SSN 采用去中心化的方式进行计算，它追踪了事务之间的 WAW 和 RAW 依赖。提交协议允许存在 RAW 依赖的事务按 SSN 顺序进行提交，而其它事务可以并行提交。此外，本文基于一个开源的内存数据库原型系统 DBx1000 [22] 实现了支持系统可恢复性的偏序事务日志技术 Poplar 并对其进行了性能评估。实验结果表明 Poplar，当部署两块固态硬盘时，Poplar 的事务吞吐量是其他事务日志技术的 2 倍——280 倍，事务提交时延比其他日志技术减少了 5 倍。

4. 针对主备日志复制技术中的有限网络 IO 带宽问题，本文提出了一种面向主备复制系统的自适应日志复制技术。该技术实现了一种传统日志传输和增量传输相结合的自适应日志传输方法。主备复制系统中的主副本能够根据实时的负载情况选择传输日志记录或者数据增量到远端的备副本。当处理高负载时，主副本使用增量传输方法来代替传统的日志传输方法，从而有效地减少了网络传输量并且避免了网络成为系统的性能瓶颈。此外，本文结合自适应日志传输方法和并行事务日志技术，实现了一个高性能的主备复制原型系统 Plover。为了保证主备副本之间的数据一致性，主副本首先采用基于段的合并算法将散列的增量数据和日志记录合并成一个有序的数据库快照，然后基于自适应日志传输方法将数据库快照传输给备副本，备副本接收到数据快照之后，采用基于段的并行回放方法恢复数据库状态。最后，本文验证了自适应日志复制技术的高效性。实验结果表示，在高负载应用中，自适应日志复制技术产生的网络传输量是传统日志复制技术的 1/5，并且自适应日志复制

技术在混合负载中具有更稳定的性能。

1.5 章节安排

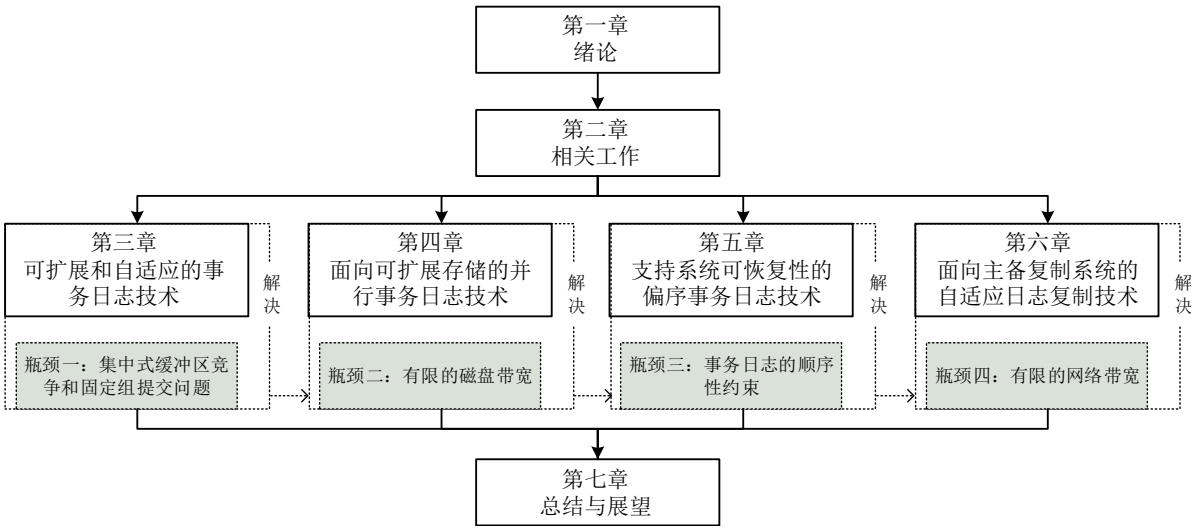


图 1.3: 本文的组织结构

本文结构组织如图 1.3所示，本文的后续章节内容安排如下：

1. 第二章介绍了数据库日志技术的相关工作，并总结出数据库日志技术在多核平台下仍存在的四个主要性能瓶颈：（1）集中式日志缓冲区竞争和固定组提交问题；（2）有限的磁盘带宽问题；（3）事务日志的顺序性约束；（4）有限的网络带宽问题。本章的第一节首先介绍了集中式事务日志技术和并行事务日志技术近十年来国内外的研究现状，然后总结了当前日志技术存在的不足之处；本章的第二节首先介绍了主备日志复制系统中相关的日志复制技术，然后总结了当前日志复制技术仍存在的性能问题。

2. 第三章实现了一种可扩展、自适应的事务日志技术来解决传统事务日志的集中式日志缓冲区竞争和固定组提交问题。本章的第一节总结了集中式事务日志在多核 CPU 环境下的首要性能瓶颈；第二节首先阐述了实现可扩展事务日志技术面临的挑战，然后详细地描述了可扩展事务日志技术的设计与实现；第三节首先分析了传统的固定组提交协议在动态变化负载中存在的问题，然

后基于生产者/消费者模型提出了一种负载自适应的组提交协议；第四节全面地验证了新事务日志技术的可扩展性和稳定性。

3. 第四章实现了一种面向可扩展存储的并行事务日志技术来解决事务日志的有限磁盘带宽问题。本章的第一节总结出：随着系统可扩展性的提升，有限的磁盘带宽成为了事务日志技术最主要的瓶颈；第二节介绍了一个实现并行事务日志技术、多版本并发控制协议以及并行日志恢复方法的内存数据库原型系统；第三节详细地描述了并行事务日志技术的整体框架、执行流程以及负载感知的日志分区策略；第四节介绍了结合并行日志技术实现的并发控制协议和日志恢复方法；第五节验证了该事务日志技术的并行性和可扩展性。

4. 第五章实现了一种支持系统可恢复性的偏序事务日志技术来解决事务日志的顺序性约束问题。本章的第一节总结出：目前的事务日志技术仍然存在顺序性约束问题；第二节首先分析了事务日志在保证系统正确性和可恢复性的基础上需要具备的必要条件，然后定义了三种事务日志级别并验证了事务日志可恢复性的正确性，最后总结了现有事务日志技术的日志级别以及它们的优缺点；第三节详细地描述了可恢复性事务日志技术的设计与实现；第四节验证了该事务日志技术的高通量、低时延和可扩展性。

5. 第六章实现了一种自适应的主备日志复制技术来解决主备复制系统中有限网络带宽的问题。本章的第一节总结出：随着事务日志性能的不断提升，有限网络带宽成为了主备复制系统最主要的性能瓶颈；第二节介绍了一个实现自适应日志复制技术和并行事务日志技术的主备复制内存数据库系统；第三节详细地描述了日志复制技术中自适应日志传输方法、基于段的合并方法以及回放方法的设计与实现；第四节验证了自适应日志复制技术的高效性。

6. 第七章总结了全文的研究内容并探讨了未来的研究方向。

第二章　相关工作

2.1　事务日志与恢复

为了实现事务的原子性和持久性，数据库管理系统（DBMS）需要保证在发生故障之后系统能恢复到一个正确的状态：包含故障发生之前所有已提交事务（已完成事务）的更新结果，以及不包含故障发生之前所有未提交事务（失败或部分完成事务）的更新结果 [54, 55]。数据库管理系统通常会遇到三种主要的故障：事务故障（Transaction failure）、系统故障（System failure）和硬件媒介故障（Media failure）[5]。事务故障一般发生在事务因冲突中断或者用户强制中断服务的时候。系统故障一般发生在操作系统故障或者内存损坏的时候。硬件媒介故障一般发生在因自然灾害（地震或者火灾）导致存储设备被损坏的时候。本文主要讨论了系统故障和硬件媒介故障的恢复策略。它们主要依赖数据库日志技术来应对发生的故障。

从上世纪七十年代开始，数据库管理系统，如 System R [33]，通常使用数据库日志来存储所有事务的执行结果。数据库日志是一个存储在磁盘中的顺序文件，它其实可以看作是一个冗余的数据库副本。数据库日志是由一条条日志记录组成的。事务每修改一个数据项时就会产生一条日志记录，并为其分配一个全局唯一的日志序列号（LSN）。日志序列号表示了日志记录在日志文件中的存储位置并保证了事务的全序关系。当发生系统故障时，数据库系统能够根据持久化的日志记录和日志序列号来重构已提交事务的执行结果并且撤销未提交事务的修改，从而保证数据库系统能够恢复到故障前的正确状态。

为了保证数据库系统恢复的正确性，事务在执行过程中需要遵循先写日志（Write Ahead Logging）协议 [54, 56]。先写日志协议规定：（1）事务修改某一数据项时产生的日志记录和数据更新值需要遵循日志记录先于更新值写入存储设备；（2）只有当事务产生的所有日志记录都已持久化之后，事务才能提交。第一条规则保证了持久化日志记录可以用于撤销未提交事务的执行结果，而第二条规则保证了日志

记录可以用于恢复所有已提交事务的状态。此外，为了加快系统恢复，数据库系统会周期地将内存中完整的数据库状态写入存储在磁盘上的检查点文件（checkpoints）[57, 58]。当系统发生故障后，它首先从检查点中恢复出某一时刻的数据库状态，然后选择必要的日志记录将数据库恢复到故障发生之前的最新状态。基于检查点的设计，数据库系统不用在故障之后恢复所有的日志记录。

2.1.1 集中式事务日志技术

ARIES [6] 是一种经典的基于先写日志协议的数据库系统恢复方法。它由 IBM 数据库团队设计与实现并广泛地运用于大多数关系型数据库管理系统，如：MySQL [12]、PostgreSql [13]、Orcale [59]、Microsoft SQL Server [60]、OceanBase [11]、TiDB [14] 和 X-DB [61]。ARIES 实现了一种基于逻辑物理日志记录的集中式事务日志技术。该日志记录主要存储了一个日志序列号（LSN）、事务号、数据项被修改以前的值（before image）和数据项被修改以后的值（after image）。在事务执行过程中，所有的事务首先将产生的日志记录按顺序写入一个集中式的日志缓冲区，然后再将日志缓冲区中的内容追加到日志文件。当某个事务所有的日志记录都已持久化之后，事务才能释放更新数据项上的写锁然后响应客户端。ARIES 的集中式设计和日志持久化操作使得事务日志和恢复模块成为了可扩展数据库系统一个主要的性能瓶颈。即使在一个单线程的数据库系统中，TPC-C 事务在集中式日志模块所花的时间占据了整个事务执行时间的 12% [10]。传统基于 ARIES 的集中式事务日志技术主要存在以下 8 个问题：（1）日志持久化时延；（2）上下文切换；（3）日志引起的事务冲突；（4）串行化填充；（5）集中式缓冲区竞争；（6）固定组提交；（7）有限磁盘 IO 带宽；（8）顺序性约束。为了解决这些问题，现有研究工作提出了许多高效的事务处理方法和日志技术。

组提交（Group Commit） [34, 62, 63] 通过减少磁盘 IO 操作次数提高了日志持久化性能。它将多条日志记录累积在日志缓冲区中，然后每隔一段时间或者每当日志记录累积到一定长度之后再将缓冲区中所有的日志记录一起写入存储设备（如：

机械硬盘 Hard Disk Drive），最后等日志记录都已持久化之后才允许事务提交。批量写盘可以避免机械磁盘不必要的寻道和寻址时间，从而减少事务的平均持久化时延。这种组提交协议被广泛地运用于基于磁盘存储的数据库管理系统。

异步提交（Asynchronous Commit） [64] 是对组提交的一种扩展。它不仅支持批量持久化，还允许事务在日志记录持久化之前提交。异步提交彻底解决了集中式事务日志技术中日志持久化时延问题，但同时也带来了数据丢失的风险。如果系统故障发生在已提交事务的日志记录还未持久化之前，那么数据库系统将无法恢复已提交事务的更新结果，从而导致数据丢失。尽管异步提交是一种不安全的策略，但是由于它能提高事务日志性能，因此大多数商业开源数据库系统 [59, 65] 仍将它配置成一个可选项供企业选择。

提前锁释放（Early Lock Release） [66–68] 允许事务在日志记录持久化之前释放数据项上的写锁。这种方法可以避免因日志持久化而导致的事务冲突问题，从而提高系统的整体性能。为了保证正确性，事务仍需要等所有日志记录都已持久化之后才能提交，并且事务的提交顺序需要遵循事务之间的先写后读（RAW）依赖。也就是说，在实现提前锁释放方法的数据库系统中，事务 T_i 可以读到事务 T_j 的修改值，但是 T_i 必须在 T_j 提交之后才能提交。

基于闪存（Flash Memory）和固态硬盘（Solid State Drive）的集中式事务日志技术 [69, 70] 使用闪存或者固态硬盘代替传统的机械硬盘来存储日志文件。由于闪存和固态硬盘具有更好的顺序写性能，因此这些事务日志技术可以降低日志持久化的时延，从而进一步提升系统的整体性能。但是，集中式的设计仍然限制了事务日志技术的可扩展性。

Aether [35, 71] 是由 Ryan 等人提出的一种可扩展的集中式事务日志技术。它是基于一个可扩展的数据库原型系统 Shore-MT [72] 进行设计与实现的。Ryan 等人总结出 ARIES 事务日志技术在多核平台中主要存在四个性能瓶颈：磁盘 IO 相关的时延问题，日志引发的锁竞争，上下文切换问题以及日志缓冲区竞争。为了解决这四个瓶颈，Aether 实现了组提交协议、提前锁释放方法、流水线执行方法以及

可扩展的日志缓冲区。流水线执行方法将事务日志流程分成日志填充、日志持久化和事务提交三个阶段。其中，日志填充和事务提交阶段由多个工作线程负责，而日志持久化由一个日志线程负责。这种执行方式解决了传统集中式事务日志技术因持久化操作而导致的上下文切换以及线程阻塞问题。可扩展的日志缓冲区通过减少事务对集中式日志缓冲区的访问次数以及占有时间解决了日志缓冲区竞争问题。它首先基于可扩展的先进先出队列 [73, 74] 将多个事务的填充请求合并成一次请求，然后为该请求分配一个缓冲区空间（日志序列号），最后允许事务并行地将日志记录填充到日志缓冲区中。虽然 Aether 大幅度地提高了传统集中式事务日志的性能和可扩展性，但是由于它仍采用了基于锁的方式来分配日志序列号，因此集中式日志缓冲区上的竞争仍然存在。

Hekaton [15]、Deuteronomy [20]、ERMIA [21] 和 Eleda [36] 在 Aether 的基础上提出了一种 Latch-free 的可扩展集中式事务日志技术。该技术采用原子指令 fetch-add 来分配日志记录在集中式日志缓冲区中的位置（日志序列号），这种方式解决了 Aether 存在的日志缓冲区锁竞争，从而进一步提高了事务日志的可扩展性。最新开源的 MySQL 8.0 [75] 实现了这种可扩展的事务日志技术。

H-Store [18] 及其企业版本 VoltDB [19] 实现了一种基于 Command Logging [76] 的事务日志技术。与 ARIES 不同，Command Logging 的日志记录主要包含事务的执行信息，即事务存储过程标识符以及事务的输入参数。这种粗粒度的日志记录方式减少了数据库系统产生的日志量，从而不仅降低了事务的平均持久化时延，还避免了有限磁盘 IO 带宽限制系统的吞吐量。虽然基于 Command Logging 的事务日志技术能够有效地提高数据库系统的事务处理性能，但是它的恢复性能却很差。当发生系统故障时，采用 Command Logging 的数据库系统需要按 LSN 顺序串行地读取日志记录，然后串行地恢复日志记录，而采用 ARIES 事务日志技术的数据库系统可以简单地根据 last-writer-wins 原则 [77] 并行地恢复日志记录。为了提高 H-Store 的恢复性能，Pacman [78] 提出了一种并行的 Command Logging 恢复方法。它首先通过对事务存储过程进行分析并生成事务依赖图，然后根据依赖图使得不存在冲

突的事务可以并行的恢复。虽然 PACMAN 能够有效地提高数据库系统的事务处理性能和恢复性能，但是由于 Command Logging 事务日志技术极度依赖于事务存储过程，因此它不适应于处理金融业务的数据库管理系统。

Calvin [27] 提出了一种基于逻辑日志记录的事务日志技术。该技术的日志记录存储了事务执行过程中的读写集合。事务日志记录大小主要由应用负载决定。在处理更新密集型应用负载时，Calvin 产生的日志记录量远小于基于 ARIES 数据库系统产生的日志记录量，因此它解决了有限磁盘 IO 导致的性能问题。但是，这种基于逻辑日志的事务日志技术只适用于实现确定性事务执行模型的数据库系统，因此它并不是一个通用的事务日志技术。

Write-Behind Logging（WBL） [79] 是一种面向新型非易失内存数据库系统 Peloton [80] 的新事务日志协议。WBL 协议规定：（1）事务在数据项上的修改可以先于事务日志记录被持久化；（2）只有当事务的日志记录持久化之后，事务才能提交。在面向非易失内存设备（Non-Volatile Memory，NVM）的数据库系统中，由于事务的数据更新项先于事务日志记录持久化，因此 WBL 的日志记录里只需要记录未提交事务的执行结果。当发生系统故障时，数据库系统只需要撤销未提交事务的更新结果就能恢复到一个正确的状态。WBL 借助新硬件的低延迟、持久性的特性消除了传统 ARIES 事务日志技术的持久化问题从而提高了系统的整体性能。但是，由于 NVM 现在还处于发展阶段，因此这种技术并未得到广泛地应用。

2.1.2 并行事务日志技术

Silo [37] 提出了一种基于 ARIES 的并行事务日志技术。它使用多个日志缓冲区和多块存储设备（固态硬盘）来存储事务日志记录。日志缓冲区和磁盘存在一对一的关系。Silo 采用了一个基于事务的日志分区策略，即同一个事务的日志记录写入一个日志缓冲区。事务与日志缓冲区存在多对一的关系。在事务日志执行过程中，多个工作线程并行地将事务日志记录写入对应的日志缓冲区，然后由多个日志线程将各自对应日志缓冲区中的内容写入磁盘。由于修改同一数据项的不同事

务可能将日志记录写入不同的日志缓冲区，因此事务间的日志记录需要反映事务的执行顺序，以便在恢复时数据库系统能按照事务执行顺序恢复到一个正确的状态。为了追踪事务之间的执行顺序同时避免集中式竞争，Silo 结合自身的乐观并发控制实现了一个基于 epoch 的事务提交协议。在并发控制协议验证成功之后，Silo 以分布式地方式为每个事务分配一个事务号。事务号主要由一个周期更新的 epoch 和追踪事务顺序的时间戳组成。Epoch 是全局单调递增的，它保证了事务之间的全序关系。时间戳记录事务之间的偏序关系，它保证了同一个 epoch 内事务之间的先写后读（RAW）依赖和写后写（WAW）依赖。当事务号分配完成之后，Silo 将事务日志记录和事务号一起写入日志缓冲区和磁盘。然后，事务以 epoch 为单位进行提交，即当且仅当属于同一个 epoch 内的所有事务日志记录都已持久化之后，对应的事务才能提交。当发生系统故障后，Silo 同样以 epoch 为单位进行恢复，即当且仅当同一个 epoch 内所有日志记录都恢复成功之后，epoch 内的事务修改才能对外可见。这种粗粒度的事务提交协议和分布式的事务号分配消除了数据库系统中的全局竞争问题，如：集中式事务号分配和集中式缓冲区竞争，从而使得系统性能得到了显著的提升。此外，FOEDUS [81] 和 Cicada [29] 也实现了类似 Silo 的并行事务日志技术。结合 Silo 的并行事务日志技术，SiloR [51] 提出了一种并行的系统恢复方法。当发生系统故障后，SiloR 能够并行地从多块磁盘中读取检查点和日志文件进行恢复，从而缩短了系统的恢复时间。虽然 Silo 和 SiloR 实现了高吞吐量的事务处理性能和恢复性能，但是事务提交时延很高。

基于非易失内存设备（NVM）的分布式事务日志技术（这里简称为 NVM-D）[39]。该技术在非易失内存中创建了多个日志文件来存储事务日志记录。事务可以按基于数据的分区方式或者按基于事务的分区方式将生成的日志记录写入对应的日志文件。由于事务的日志记录会被分布到多个日志文件中，所以 NVM-D 实现了一种新的日志号（GSN）来追踪事务之间的先后执行顺序。GSN 不仅保证了多个事务在同一数据项上的先写后读（RAW）依赖、写后写（WAW）依赖和先读后写（WAR）依赖，还保证了同一事务内的请求执行顺序。此外，它还提出了一种被动

的组提交（passive group commit）协议来保证事务的正确提交。该协议规定只有当某一事务的所有日志记录，以及它依赖事务的日志记录都被持久化之后，该事务才能提交。这种分布式事务日志技术消除了传统事务日志技术的集中式缓冲区竞争和串行持久化问题，从而提高了系统的可扩展性和并行性。然而，由于它允许工作线程直接持久化日志记录，因此频繁的 IO 操作（mfence）会降低系统性能。此外，这种非流水线的事务日志执行方式不适用于部署在普通存储设备上的数据库系统。

NV-Logging [38] 是一种充分利用 NVM 字节寻址和持久化特性的并行事务日志技术。该技术为事务的每条日志记录在 NVM 上分配一个私有的日志空间，并且使用遵循全序关系的日志序列号 LSN 来保证事务的执行顺序。在事务日志执行过程中，NV-Logging 首先使用原子指令为每条日志记录分配一个 LSN，然后并行地将日志记录写入 LSN 指向的 NVM 空间，最后 NV-Logging 按 LSN 顺序依次检测日志记录的填充情况并完成事务提交。基于 NVM 的事务日志技术不仅消除了传统 ARIES 日志的集中式日志缓冲区竞争，还缩短了日志持久化的时间。此外，PCMLogging [82] 和 SCMLogging [83] 也利用了非易失存储内存设备实现了并行的日志持久化。尽管所有基于 NVM 的事务日志技术都提高数据库系统的性能和可扩展性，但由于目前 NVM 的技术还不成熟，因此它们并未得到广泛地应用。

总结。到目前为止，传统事务日志技术在多核 CPU 平台下仍然存在以下问题：（1）固定组提交不适用于动态变化的负载；（2）有限磁盘 IO 带宽限制了事务日志的最大吞吐量；（3）顺序性约束限制了事务日志的并行性。为了解决这三个问题，本文分别提出了可扩展和自适应的事务日志技术、面向可扩展存储的并行事务日志技术和支持系统可恢复性的事务日志技术。

2.2 复制技术

复制是一种使用多个服务器或者多个资源来实现系统可用性的技术。每一个服务器/资源称为一个副本，每个副本中存储了相同的数据镜像。当其中某一个副本出现故障之后，其他副本可以继续提供服务。对于金融领域的企业来说，数据库

系统即使出现短暂的中断都将造成巨大的损失。因此，一般会在多个城市部署多个数据库系统副本，如：现在常说的两地三中心或者五地三中心。

2.2.1 主备日志复制模型

为了避免媒介故障造成系统服务中断，最简单的媒介故障恢复方法是在两个或者多个地理位置上部署相同的硬件和数据副本。这种方法被称为系统级复制 [2]。基于系统级复制的数据库系统主要包含一个主系统（主副本）和一个或者多个备系统（备副本）。主备系统之间通常采用以太网连接。这种系统也被称为主备复制系统或者远端热备系统。

在主备复制数据库系统中，主副本（primary）首先处理客户端的读写请求，然后以日志记录的形式将执行结果传输给备副本（backups）。当接收来自主副本的日志记录之后，备副本将日志记录写入存储设备并按事务执行顺序回放日志记录从而使得备副本拥有和主副本一样的数据库状态。为了提高资源利用率和系统的整体性能，备副本可以对外提供只读服务，但是它不能接收写请求。当主副本发生故障之后，备副本在回放完所有日志记录之后可以替代旧主副本成为新的主副本并提供读写服务。主副本的日志传输（log shipping）主要分为两种方式：同步传输方式和异步传输方式。根据不同的传输方式，主备复制系统分为三种模式：最大性能模式、最大可用性模式和最大保护模式：

1. 最大性能模式。属于最大性能模式的主备复制系统允许：只要事务的日志记录持久化到主副本之后，事务就能提交。

2. 最大可用性模式。在这种模式下，如果备副本存在，那么事务必须等待所有的日志记录持久化到主副本和备副本之后才能提交。如果备副本发生了故障，那么事务的日志记录只需要持久化在主副本，事务就可以提交。

3. 最大保护模式。在这种模式下，不论备副本是否发生故障，事务只有在日志记录持久化到主副本和备副本之后才能提交。也就是说，如果备副本发生故障，那么事务将不能提交。

在最大性能模式中，主备副本之间的日志传输采用了异步方式。这种方式将日志传输和备副本持久化操作从事务处理过程中剥离出来，从而使得主备复制系统的性能类似于单节点系统性能。但是，这种方式可能出现大量已提交事务的日志记录没有传输给备副本的情况。如果主副本发生了硬件媒介故障，那么主备复制系统将丢失大量的数据。相反，最大可用性模式和最大保护模式不同程度上保证了数据不丢失。但是，由于事务执行过程中增加了额外的网络传输时延和备副本日志持久化时延，因此系统的整体性能会降低。事务处理性能和系统安全性一直是主备复制系统需要权衡的一个问题。如何在保证数据不丢失的情况下尽可能的提高系统的性能也是当前研究的热点。此外，大多数数据库系统 [42, 59] 实现了上述三种模式供企业自由选择。

2.2.2 日志复制技术

许多商业数据库系统，如：MySQL [12]、DB2 [41]、Oracle [84]、PostgreSQL [45] 和 SQL Server [40]，实现了基于日志传输的主备日志复制技术。它们的主副本在处理完读写事务请求之后，将基于 ARIES 的日志记录按照日志序列号（LSN）传输给备副本。备副本接收到日志记录之后，首先将日志记录写入本地的存储设备，然后再采用单线程串行地回放日志记录，从而保证主备副本拥有相同的数据库状态。随着多核硬件和并行事务处理技术的出现，主副本的事务处理性能越来越高，传统基于日志传输方法的日志复制技术成为了主备复制系统最主要的性能瓶颈 [50]。它主要存在两个问题：（1）备副本单线程的回放方法；（2）有限网络 IO 带宽。为了解决这这些问题，现有研究工作提出了许多高性能的日志复制技术。

KuaFu [49] 基于传统的日志复制技术提出了一种并行回放方法。它通过日志记录来追踪事务之间的写后写依赖，然后允许不存在写写冲突的事务能够并行地回放。为了使得备副本能够对外提供一致性的读服务，事务在备副本中的提交顺序需要与主副本保持一致。因此，KuaFu 结合多版本并发控制协议实现了一种基于快照版本的并行回放方法。并行回放能够提高备副本的处理性能，从而缩短主备

副本之间的数据版本差。但是，由于 KuaFu 采用单线程来构建事务依赖图，因此事务依赖追踪方法成为了限制备副本性能的新瓶颈。

H-Store [18] 及其企业版本 VoltDB [19] 和 Calvin [27] 实现了基于事务逻辑日志的日志复制技术。随着多核硬件的出现，传统主备复制系统每秒产生的 ARIES 日志记录量超出了以太网的传输能力。为了减轻网络的负担，它们允许主副本将事务的 Command Logging 或者读写集合传输给备副本。虽然这种方法减少了网络传输量，避免了有限网络带宽成为瓶颈，但是备副本必须按日志序列号串行地回放事务日志记录，因此备副本的回放性能很差，

可扩展的日志复制技术 [52] 是一种基于确定性事务执行模型提出的快速日志复制技术。它不仅可以降低网络日志传输量，还支持备副本的并行回放。在可扩展日志复制技术中，主副本只需要将事务的输入集合按事务执行顺序串行地传输给备副本。备副本接收到日志记录之后，结合事务存储过程并行地回放日志记录。虽然这种方法集合了逻辑日志传输和并行回放的所有优势，但是它极度依赖于确定性事务执行模型和存储过程，因此不适用了大多数关系型数据库管理系统。

Query Fresh [53] 是基于高速网络（InfiniBand）[85] 和非易失内存设备（NVM）实现的快速日志复制技术。由于高速网络具有高带宽、低时延特性并且支持远程直接内存访问（Remote Direct Memory Access，RDMA） [86]，因此主副本可以直接将 ARIES 日志记录快速地写入备副本的日志缓冲区。Query Fresh 将备副本的日志缓冲区存储在 NVM 上，从而减少一次备副本日志持久化操作。此外，Query Fresh 基于多版本数据存储结构实现了并行地日志回放。虽然，Query Fresh 解决了传统基于 ARIES 日志复制的网络瓶颈和备副本单线程回放问题，但是由于它依赖于新型硬件因此并不适用了商业数据库。

总结。到目前为止，传统基于日志传输的日志复制技术在多核 CPU 平台下仍然存在一个问题：有限的以太网 IO 带宽限制了主备复制系统最大的吞吐量。为了解决这个问题，本文提出了一种面向主备复制系统的自适应日志复制技术。

第三章　可扩展和自适应的事务日志技术

3.1　引言

随着多核 CPU 和大容量内存的出现，可扩展的事务处理技术得到了前所未有的关注和发展 [87]。然而，数据库系统仍然采用传统 ARIES [6] 的集中式设计、串行执行方式和磁盘存储模式来保证数据的可靠性和可恢复性。现有的集中式事务日志技术大多采用多个生产者（并行执行的事务）和一个消费者（一个日志持久化线程）模型，这种模型在多核平台下主要存在以下两个性能瓶颈：（1）集中式日志缓冲区竞争。尽管在多核数据库系统中多个事务能够并行地执行，但是它们生成的日志记录仍按全序串行地写入集中式日志缓冲区。为了保证日志记录之间的全序关系，集中式事务日志技术使用全局锁来分配日志记录在缓冲区上的位置（日志序列号）。当成百上千的事务同时分配日志序列号时，集中式日志缓冲区上的竞争将导致大量的 CPU 资源浪费，从而影响系统的可扩展性；（2）固定组提交。传统集中式事务日志技术采用了一种固定的组提交协议来降低日志持久化时延 [34]，即每隔某一固定时间或者每当日志缓冲区中的日志记录量达到某一固定值时，数据库系统触发一次批量日志写盘操作然后提交事务。但是，这种固定的组提交协议并不适用于动态变化的负载。以固定时间的组提交为例，当处理低负载应用时，太长的组提交时间会导致大量已完成日志填充的事务等待持久化和提交；而当处理高负载应用时，太短的组提交时间又会导致事务因过载的磁盘 IO 操作而被阻塞。

为了消除传统集中式事务日志技术的性能瓶颈，支持数据库系统多核可扩展、高性能的事务处理能力，本项研究提出了一种可扩展、自适应的事务日志技术 Laser。Laser 的核心思想是利用高并发访问的数据结构——追踪表以及原子操作来解决集中式事务日志技术的缓冲区竞争问题。此外，为了保证数据库系统能够在动态变化的负载中保持稳定的性能，本项研究提出了一种自适应的组提交协议。它能够根据实时的负载情况动态地调整组提交的触发条件。最后，本项研究基于

开源的分布式数据库系统 CEDAR [16] 实现了该事务日志技术，并使用 YCSB 基准测试 [88] 对其进行了性能评估。

综上所述，本项研究主要包含了以下贡献：

1. 本项研究提出了一种可扩展的事务日志技术，它使用高性能的原子指令和并行的日志填充方法提高了传统集中式事务日志的可扩展性。

2. 本项研究利用生产者/消费者模型对事务日志执行流程进行建模，然后分析了负载变化与事务性能之间的关系，最后提出了一种负载自适应的组提交协议。

3. 本项研究在开源分布式系统 CEDAR 中实现了可扩展、自适应的集中式事务日志技术 Laser 并对其进行了验证。实验结果显示，与已有的集中式事务日志技术相比，Laser 具有多核可扩展性，并且事务吞吐量提升了 1.4 倍，事务提交时延降低了 86%。

3.2 可扩展事务日志

本节主要介绍了一种解决集中式日志缓冲区竞争的可扩展事务日志技术。本节首先描述了可扩展事务日志技术的整体架构以及它面临的技术挑战，然后详细地从数据结构和执行流程两个方面介绍了它的具体实现。

3.2.1 整体架构

为了解决传统集中式事务日志存在的日志缓冲区竞争问题，本项研究提出了一种可扩展的集中式事务日志技术。图 3.1给出了它的整体框架。该事务日志技术主要包含一个集中式日志缓冲区和三种线程：多个工作线程、一个日志线程以及一个分组线程。工作线程负责把生成的日志记录并行地写入集中式日志缓冲区，分组线程主要将日志缓冲区分成多个大小不相等的日志块/日志组，日志线程负责将日志组按顺序写入磁盘。可扩展的事务日志执行流程主要包括三个阶段：日志准备阶段，日志分组阶段以及日志持久化阶段。

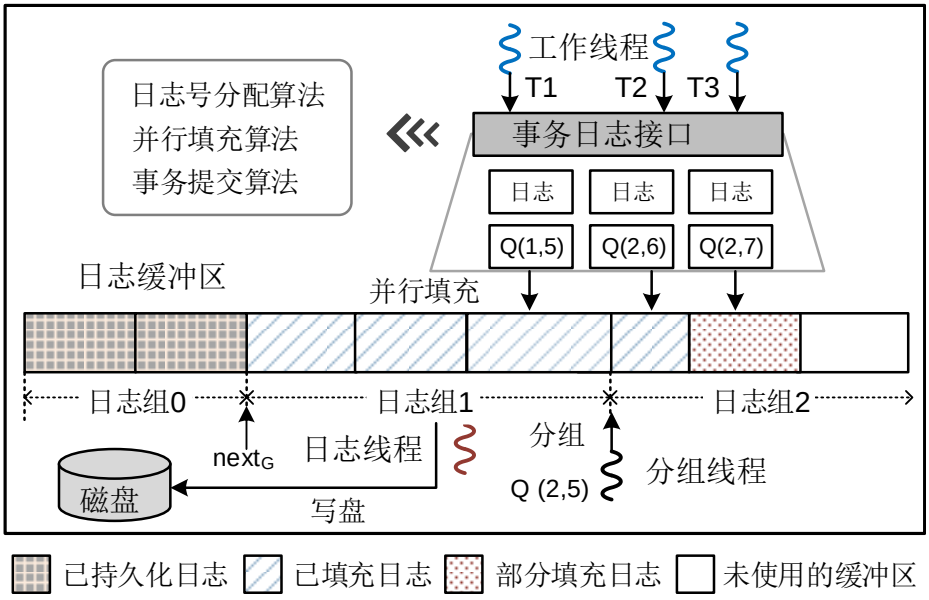


图 3.1: 可扩展性事务日志的整体架构

在日志准备阶段，工作线程为它们各自处理的事务获取唯一的日志序列号 LSN（日志缓冲区上的位置），并将事务日志记录填充到对应的日志缓冲区空间。在可扩展事务日志技术中，日志填充采用了并行的执行方式，并且 LSN 分配使用了高性能的原子指令 comapre-and-swap（CAS）。但是，并行的 LSN 分配以及日志填充会不可避免地导致日志空洞（例如，部分填充的日志记录）。如果将存在空洞的事务日志写入存储设备，那么数据库系统将出现数据丢失并且不能恢复到一个正确的状态。因此，可扩展事务日志技术的最大挑战是如何高效地追踪日志缓冲区上的空洞，从而避免数据丢失并且保证系统的正确性和可恢复性。

为了解决这个问题，本项研究动态地把集中式日志缓冲区分为多个日志组。每个日志组看作一个追踪单元以及一个持久化单元。当且仅当一个组里不存在日志空洞时，这个日志组里的日志记录才能被写入存储设备。此外，为了使事务日志具有更好的灵活性，每个日志组的大小是不固定的。

在日志分组阶段，分组线程根据组提交协议将集中式日志缓冲区分成多个日志组。在可扩展事务日志技术中，组提交采用每隔 D 时间触发一次持久化操作，即每 D 时间内产生的日志记录形成一个日志组。在日志持久化阶段，日志线程按顺序依次检查每个组中是否存在日志空洞。如果日志组中不存在日志空洞，那么日

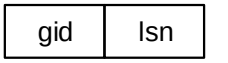
志线程将日志组内的日志记录写入磁盘。随后，日志线程唤醒持久化事务对应的工作线程并由工作线程完成事务提交。

3.2.2 数据结构

为了解决因全局锁导致的串行瓶颈，可扩展事务日志设计了允许并发访问的数据结构——追踪表和日志号。它们的具体结构如图 3.2所示。

gid	state	slsn	len	filled	closed
0	Available	0	0	0	false
1	Durable	2	3	3	true
2	Available	5	0	1	false
3	Available	0	0	0	false
4	Available	0	0	0	false
...	...	...	...	...	...

追踪表



日志号

图 3.2: 追踪表和日志号数据结构

追踪表。跟踪表（G）记录了每个日志组的重要信息。事务日志技术可以根据这些信息来追踪每个日志组里是否存在日志空洞。如图 3.2所示，追踪表是由多个日志组组成的数组结构。每个日志组包含五个成员变量（$state$，$slsn$，len，$filled$，$closed$），其中$state$ 表示日志组的状态，$slsn$ 表示日志组在集中式日志缓冲区中的起始位置，len 表示日志组中日志记录的总长度，$filled$ 表示日志组中已填充的日志记录长度，$closed$ 表明日志组是否已创建（如果 $closed$ 的值为真则表示日志组已创建成功）。日志组包含两种状态：可使用状态（Available）和可持久化状态（Durable）。当日志组处于可使用状态时，组里的日志记录才能被允许填充到集中式日志缓冲区；当日志组处于可持久化状态时，组里的日志记录才能被写入磁盘。在系统初始态时，所有的日志组都处于可使用状态。当且仅当$closed$ 的值为真并且len 的值与$filled$ 的值相等时，日志组的状态才会被修改为可持久化。这种方式保证了持久化的日志记录中不会出现日志空洞。

追踪表不仅能够避免日志空洞问题还提供了较好的事务处理性能。虽然追踪表中有许多共享变量，但是这些共享变量能够被所有线程并行地访问。表 3.1描述

表 3.1: 不同线程在共享变量上的访问模式

线程类型	共享变量				
	$state$	$slsn$	len	$filled$	$closed$
工作线程	写	无	无	写	无
分组线程	写	写	写	无	写
日志线程	写	读	读	读	读

了所有工作线程在追踪表上的访问模式。对任意一个共享变量来说，同一时间只有一种线程能够对它进行修改。追踪表的设计避免了共享变量上的写写冲突，提高了事务执行性能。

日志号。日志号（Q）主要用于为事务分配日志序列号（日志缓冲区空间）。根据日志号，事务能够确定日志记录所属的日志组。如图 3.2所示，日志号是一个 128 位的数据结构，它包含两个成员变量（gid，lsn），其中gid 表示日志记录所属的日志组，lsn 指向日志记录在集中式日志缓冲区上的填充位置。为了实现并行的日志号分配，可扩展事务日志技术采用了 128 位的 CAS 原子操作，该操作保证了gid 和lsn 的计算能够在同一条指令中完成。

3.2.3 执行流程

接下来，本节将结合追踪表和日志号的数据结构详细地描述可扩展事务日志技术的执行流程和算法实现。

算法 1: 日志号分配

输入: 事务日志长度 $|T|$
输出: 新生成的日志号

```
1 do
2 |   oldQ ← Q ;
3 |   newQ.gid = oldQ.gid ;
4 |   newQ.lsn = oldQ.lsn + |T| ;
5 while 0 ! = 128CAS(Q, oldQ, newQ);
6 return newQ;
```

日志号分配。当事务T 进入日志准备阶段时，工作线程读取当前的日志号并使用 CAS 原子操作生成一个新的日志号 $Q(gid, lsn = lsn + |T|)$，其中 $|T|$ 表示事

务T 的日志记录长度。算法 1给出了日志号分配的伪代码。在该算法中，第 2 行到第 4 行描述了工作线程根据当前的日志号和事务日志长度计算出了一个新的日志号。然后，第 5 行描述了工作线程用新的日志号原子替换当前的日志号。如果 CAS 函数的返回值为真，那么直接返回新生成的日志号，否则，工作线程重新执行第 2 行到第 4 行的代码。如图 3.1所示，假设所有事务的日志记录长度为 1，那么事务 T_1、T_2 和 T_3 的日志号分别为 $Q_1(1,5)$、$Q_2(2,6)$、$Q_3(2,7)$。

算法 2: 并行填充

输入: 当前事务的日志号 $newQ$，事务日志长度 $|T|$
输入: 日志缓冲区长度 $|L|$
输入: 追踪表G，追踪表大小 $|G|$
输出: 日志填充成功或者失败

```
1  i = newQ.gid % |G|;
2  while G_i.state == Available && log space of L is enough do
       /* 逻辑地址转换成物理地址 */
3      start = (newQ.lsn − |T|) % |L|;
4      end = newQ.lsn % |L|;
5      if end < start then
           /* 日志记录跨越日志缓冲区尾部和头部 */
6          Fill(start,|L|);
7          Fill(0,end);
8      else
9          Fill(start,end);
10     end
       /* 修改日志组已填充的日志长度和的状态 */
11     G_i.filled = G_i.filled + |L|;
12     if G_i.closed == true && G_i.filled == G_i.len then
13         G_i.state = Durable;
14     end
15 end
```

并行填充。日志号分配成功之后，工作线程根据日志号确定事务日志记录所属的日志组。如果追踪表中日志组的状态为可使用（Available）并且集中式日志缓冲区中有足够的空间，那么工作线程将日志记录填充到日志缓冲区，然后增加该日志组中已完成填充的日志长度。如果当前日志组已经创建成功并且日志组中已填充的日志长度等于日志组的总日志长度，那么工作线程将日志组的状态修改为可持

久化（Durable）。算法 2描述了并行填充算法的伪代码。由于日志号中lsn 只记录了日志记录在缓冲区上的逻辑地址，所以在填充时，该算法需要将逻辑地址转换成物理地址，如算法中的第 3 行和第 4 行。同时算法考虑了日志记录跨越日志缓冲区尾部和头部的情况，如算法中的第 5 行到第 7 行。如图 3.1所示，T_1 的日志记录属于日志组 1（G_1），T_2 和 T_3 的日志记录属于日志组 2（G_2）。结合图 3.2的追踪表，G_1 和 G_2 的状态都为 Available，三个事务可以并行地将日志记录填充日志缓冲区。当 T_1、T_2 完成日志填充之后，它们分别将日志组的$filled$ 加 1。由于当前 G_1 的$closed$ 为真并且len 等于$filled$，因此 T_1 将 G_1 的状态修改为 Durable。这表示日志组 1 中所有日志记录都已完成日志填充且不存在日志空洞。

算法 3: 日志分组

输入: 追踪表G，追踪表大小 $|G|$
输出: 日志分组成功或者失败
/* 每隔时间 D 创建一个新的日志组 */
1 *Sleep(D)*;
2 **do**
3 $oldQ \leftarrow Q$;
4 $newQ.gid = oldQ.gid + 1$;
5 $newQ.lsn = oldQ.lsn$;
6 **while** *0* ! = 128CAS (Q, $oldQ, newQ$);
7 $i = oldQ.gid$ % $|G|$;
8 $j = newQ.gid$ % $|G|$;
/* 修改新创建日志组的内容 */
9 **while** $G_i.state == Available$ **do**
10 $G_i.closed = true$;
11 $G_i.len = oldQ.lsn - G_i.slsn$;
12 **if** $G_i.closed == true$ && $G_i.filled == G_i.len$ **then**
13 $G_i.state =$ Durable;
14 **end**
15 **end**
/* 设置下一个日志组 */
16 **while** $G_j.state == Available$ **do**
17 $G_j.slsn = newQ.lsn$;
18 **end**

日志分组。每隔一段时间（D），分组线程会创建一个新的日志组。它首先将当前的日志号Q 更新为 $Q'(gid = gid + 1, lsn)$，然后修改Q 对应的日志组为已创建

（$closed$ 为真）并设置该日志组的日志记录总长度。此时，如果Q 对应日志组中的日志总长度等于已填充的日志长度，那么分组线程将日志组的状态修改为可持久化（Durable）。最后，如果Q' 对应日志组的状态为 Available，那么分组线程将该日志组的起始位置设置为Q' 中的 lsn。算法 3给出了日志分组算法的伪代码。在该算法中，分组线程的任务主要分为三个步骤：第一步，分组线程每隔 D 时间生成一个新的日志号，如算法中的第 1 行到第 6 行。D 时间内生成的日志记录看作一个日志组。第二步，分组线程修改新创建日志组的日志总长度和对应的$closed$ 变量，如第 9 行到第 15 行。这两个字段主要用于检查日志组中是否存在日志空洞。如果没有日志空洞，日志组的状态被修改为可持久化。第三步，分组线程设置下一个日志组在日志缓冲区中的开始位置，如第 16 行到第 18 行。如图 3.1所示，分组线程将 T_1 的日志记录和前两个日志记录组成日志组 G_1 并将日志号修改为 $Q(2,5)$。之后的 T_2 和 T_3 基于 $Q(2,5)$ 进行分配日志号。最后分组线程将日志组 G_2 的日志总长度和起始位置$slsn$ 分别设置为 3 和 5。

日志持久化。在可扩展事务日志技术中，日志持久化由单独的日志线程负责。而工作线程在完成日志记录填充之后可以直接处理下一个事务请求。这种流水线式执行方法避免了传统 ARIES 日志存在的上下文切换问题和 CPU 资源浪费问题。日志线程记录了下一个即将被持久化的日志组（用$next_G$ 来表示）。当开始执行持久化操作时，日志线程首先检查$next_G$ 对应日志组的状态。如果该日志组的状态为可持久化，那么日志线程将组里的日志记录写入磁盘。然后，日志线程重置日志组的所有变量并更新$next_G$ 的值。算法 4给出了日志持久化的伪代码。在该算法中，第 1 行到第 10 行描述了日志线程找到下一个将被持久化的日志组并把它的日志记录写入存储设备。由于日志组的起始位置采用了逻辑地址，因此算法需要将它转换成物理地址之后再进行持久化。同时算法还考虑了日志组中日志记录跨越日志缓冲区的尾部和头部的情况，如第 5 行到第 8 行。第 11 行到第 16 行描述了日志线程完成日志持久化之后重置日志组的状态并修改$next_G$。如图 3.1所示，当前的$next_G$ 等于 1。由于当前 G_1 的状态为 Durable，因此日志线程可以将日志组 1 中的日志记

算法 4: 日志持久化

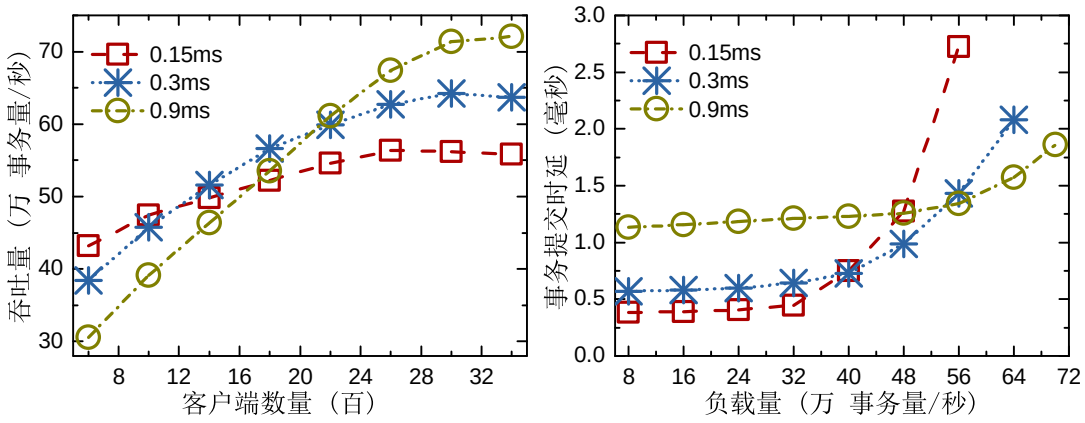
输入: 日志缓冲区长度 $|L|$
输入: 追踪表G，追踪表大小 $|G|$
输入: 事务日志长度 $|T|$
输出: 日志持久化成功或者失败

```
/* 确定下一个将被持久化的日志组并执行持久化操作 */
i = next_G % |G|;
while G_i.state == Durable do
    /* 逻辑地址转换成物理地址 */
    start = (G_i.slsn) % |L|;
    end = (G_i.slsn + G_i.len) % |L|;
    if end < start then
        /* 日志组中的日志记录跨越日志缓冲区尾部和头部 */
        Flush(start,|L|);
        Flush(0,end);
    else
        Fill(start,end);
    end
    /* 重置日志组 */
    G_i.slsn = Available; G_i.slsn = 0;
    G_i.len = 0; G_i.filled = 0;
    G_i.closed = false;
    next_G = next_G + 1;
end
```

录写入磁盘，然后更新$next_G$ 的值并且重置 G_1 的所有成员变量。

3.3 自适应组提交

上一节介绍了一种可扩展的事务日志技术。该技术使用了固定的组提交协议，即它把每固定 D 时间内产生的日志记录看作一个持久化/提交单元（在本文中，一个单元就是一个日志组）。然而，本项研究发现传统的组提交不适用于动态变化的应用负载。因此，本项研究提出了一种自适应的组提交协议。本节首先详细地描述了组提交时间对事务日志性能产生的影响；然后阐述了借助生产者/消费者模型对日志性能的分析过程；最后介绍了自适应组提交的计算规则。



(a) 事务吞吐量　　(b) 事务提交时延

图 3.3: 组提交时间对事务日志性能的影响

3.3.1 问题描述

通过实验测试，本项研究发现了两个有趣的现象：（1）组提交条件会影响事务日志的吞吐量；（2）组提交条件会影响事务的提交时延。图 3.3显示了不同的组提交时间对数据库系统事务吞吐量和事务时延的影响。本次实验使用了一个自定义的基准测试，该基准测试主要包含一张测试表和一种只写负载。测试表由一个 64 位的整型列（主键）和一个 10 字节的字符串列组成。表中默认存储了 1 亿条记录。只写负载包含一个写事务，该写事务每次只修改测试表中的一条记录。只写负载服从均匀分布。在测试过程中，可扩展的事务日志技术分别设置了三种不同的组提交时间：$0.15ms$，$0.3ms$ 和 $0.9ms$，然后通过改变客户端的连接数和负载量来观察系统吞吐量和事务平均时延的变化。

如图 3.3(a)所示，在不同的组提交时间下，事务日志技术获得了不一样的事务吞吐量。当事务日志技术的组提交时间为 $0.9ms$ 时，数据库系统获得了最高的吞吐量。而组提交时间为 $0.15ms$ 时，系统的吞吐量最小。因此组提交协议会影响系统的最大事务吞吐量。如图 3.3(b)所示，随着负载量的升高，三种固定组提交协议展现了不同的事务提交时延，没有任何一种组提交能够在所有的负载下一直保持最优的事务提交时延。当客户端每秒产生低于 40 万的负载请求时，组提交时间为 $0.15ms$ 的事务日志技术获得了最小的时延。当客户端每秒产生的负载请求量在 40

万到 56 万之间时，组提交时间为 $0.3ms$ 的事务日志技术拥有最好的提交时延。而当客户端每秒产生的负载请求继续增加时，组提交时间为 $0.9ms$ 的事务日志技术能够获得更小的提交时延。因此，固定组提交协议不适用于动态变化的负载。

3.3.2 建模分析

为了使得数据库系统能够获得最大的事务吞吐量，并且能在动态变化的应用负载中始终保持最低的事务提交时延，本项研究首先使用经典的生产者/消费者模型来模拟事务日志流程。然后分析了组提交协议对数据库事务日志的影响。本项研究将可扩展事务日志技术中的分组线程和工作线程看作生产者，日志线程看作消费者。然后将事务日志执行流程抽象为：生产者每隔 D 时间产生一个日志组（任务），然后将日志组压入一个还有剩余空间的队列。消费者从非空的队列里获取一个日志组，然后将日志组里的日志记录写入磁盘。在此过程中，生产者和消费者分别使用了不同的硬件资源，其中生产者主要利用 CPU 资源，而消费者主要利用磁盘资源。这些硬件资源决定了生产者和消费者的处理能力，如：CPU 的计算能力决定了生产者每秒产生的日志记录数和日志组数量；磁盘的处理能力决定了消费者每秒能处理的日志组数以及日志记录总量。表 3.2显示了所有硬件资源的最大能力值。只有充分地利用硬件资源，事务日志才能获得最好的性能。

表 3.2: 硬件资源属性及其影响

硬件资源	影响	最大值
CPU	生产者每秒产生的日志记录数（每个日志组的大小）	100%
IOPS	消费者每秒能处理的日志组数	3 万
带宽	消费者每秒能处理的日志记录总量	$600M/s$

表 3.3: 建模分析中使用的符号

符号	意义
D	组提交时间
$\|\bar{T}\|$	每条日志记录的长度（平均值）
λ	工作线程每秒产生的日志记录数
μ	日志线程每秒持久化的日志记录数

接下来，本节利用生产者/消费者模型分析了组提交协议对事务日志性能的影响。表 3.3给出了分析过程中使用的符号。在事务日志技术框架中，λ 主要由客户端每秒产生的负载量（本项研究假设每个事务只生成一条日志记录）和事务日志号分配的性能共同决定。因此工作线程每秒产生的日志记录数存在一个上限：

$$\lambda \leq 1/t_{CAS} \tag{3.1}$$

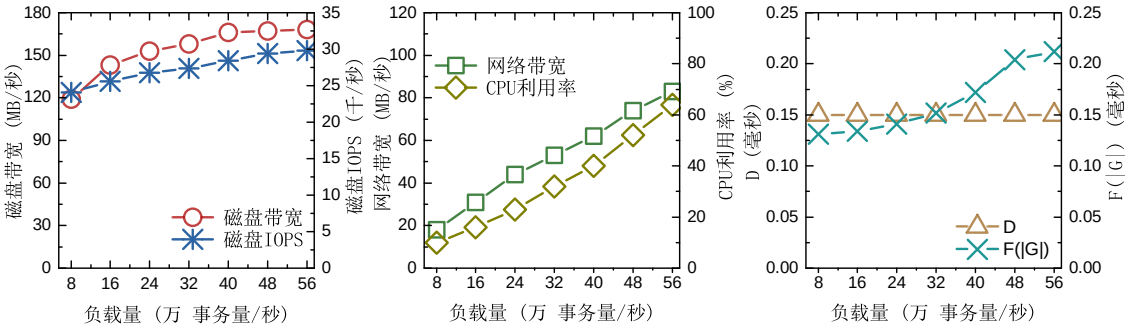
μ 主要由日志组里的日志记录数以及持久化一个日志组所需的磁盘时延决定。此处用$\mathcal{F}(|\bar{G}_i|_D)$ 来表示日志线程持久化一个日志组所需的时延，其中$|\bar{G}_i|_D$ 表示一个日志组里日志记录的总长度。日志组是由分组线程每隔 D 时间创建的。假设工作线程每秒产生的日志记录数服从泊松分布，那么日志组包含 $\lambda * D$ 条日志记录。因此日志线程每秒能持久化的日志记录数可以计算为：

$$\mu = \frac{\lambda * D}{\mathcal{F}(|\bar{G}_i|_D)} \quad w.r.t. \quad |\bar{G}_i|_D = |\bar{T}| * D * \lambda \tag{3.2}$$

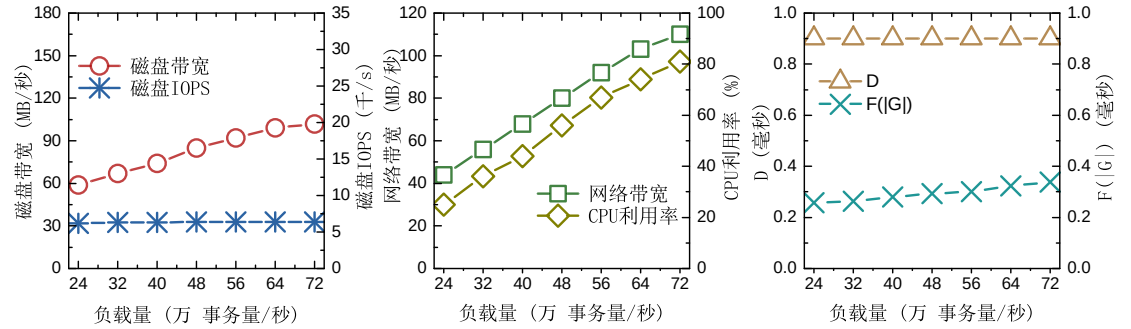
吞吐量。本节首先分析了组提交时间对事务吞吐量的影响。在生产者/消费者模型中，生产率（λ）和消费率（μ）存在以下两种场景：

场景 1：$\lambda > \mu$，即 $D < \mathcal{F}(|\bar{G}_i|_D)$（组提交时间小于持久化时间）。在这种情况下，日志组的生成速率远远大于日志组被持久化的速率，也就是说日志线程无法及时处理掉已生成的日志组，因此大量日志组被堆积在队列中。此时，系统的最大吞吐量由日志线程的处理能力决定。然而，由于日志线程的处理能力受限于磁盘的 IOPS，因此日志线程每秒能持久化的日志组数量是有限的。当采用一个极小的组提交时间时，日志组持久化的时间也会相应变小。极端情况下，如果一条日志记录组成一个日志组，那么日志组持久化的时间将远远小于 $\frac{1}{IOPS}$，因此系统的吞吐量无限接近于 IOPS 值。

场景 2：$\lambda <= \mu$，即 $D >= \mathcal{F}(|\bar{G}_i|_D)$（组提交时间大于等于持久化时间）。在这种情况下，日志线程处于相对比较空闲的状态，即日志线程能够及时处理工作线程生成的日志组。因此，系统的最大吞吐量由生产者的处理能力决定。由于生产者



(a) 当 $D = 0.15ms$ 时的硬件资源情况（$\lambda > \mu$）



(b) 当 $D = 0.9ms$ 时的硬件资源情况（$\lambda < \mu$）

图 3.4: 不同组提交时间下的硬件资源使用情况

的处理能力由 CPU 和客户端共同决定，因此当有足够多的物理核和客户端时，系统将能取得较好的吞吐量。

本项研究使用自定义的基准测试验证了不同组提交条件对系统吞吐量产生的影响。本次实验选择了两种组提交时间：$D = 0.15ms$ 和 $D = 0.9ms$，然后通过改变客户端的负载量来观察系统硬件资源情况。本次实验主要监控了四种硬件资源：磁盘带宽、磁盘 IOPS、客户端网络带宽以及 CPU 利用率。磁盘带宽表示每秒写入磁盘的日志记录量，磁盘 IOPS 表示每秒执行的 IO 次数（每秒持久化日志组的个数），客户端网络带宽是指每秒客户端与数据库系统之间传输的数据量，CPU 利用率是指每秒使用的 CPU 比例。图 3.4显示了实验结果。随着负载量的不断增加，D 与 $\mathcal{F}(|\bar{G}_i|_D)$ 之间的关系也发生了相应的改变。当 $D >= \mathcal{F}(|\bar{G}_i|_D)$ 时，日志线程处于空闲状态，因此磁盘带宽以及磁盘 IOPS 处于未饱和状态，从而不会影响系统性能。从图 3.4(b)中可以看出，当 $D = 0.9ms$ 时，数据库系统的磁盘没有成为性能瓶颈，因此事务日志技术获得了最高的吞吐量。此时的系统吞吐量最终受限于客户端

与数据库之间的带宽（最大值为 $125M/s$）。然而，当 $D < \mathcal{F}(|\bar{G}_i|_D)$ 时，持久化操作频率会迅速增加，最终将达到磁盘 IOPS 的最大值（3 万）。如图 3.4(a)所示，当 $D = 0.15ms$ 时，数据库系统的事务处理性能最终受限于磁盘 IOPS。

通过理论与实验分析，本项研究得出以下结论：为了获得较高的吞吐量并且避免磁盘 IO 对系统事务处理性能的影响，D 与 $\mathcal{F}(|\bar{G}_i|_D)$ 之间的关系应该始终保持 $D >= \mathcal{F}(|\bar{G}_i|_D)$ 并且 $D > 1/IOPS$。

提交时延。接下来，本节分别从两个场景讨论组提交时间对提交时延的影响。

场景 1：当组提交时间 D 较大时（$D > \mathcal{F}(|\bar{G}_i|_D)$），日志记录等待被分组的时间也会相应地变大，从而增大了事务的提交时延。结合图 3.3(b)和图 3.4(b)来看，当 $D = 0.9ms$ 时，系统中不存在 CPU 和磁盘瓶颈。但在处理低负载应用时，由于大量事务日志记录处于等待被持久化和提交的状态，因此事务的提交时延明显高于其他组提交协议的事务提交时延。

场景 2：当组提交时间 D 较小时（$D < \mathcal{F}(|\bar{G}_i|_D)$），不仅事务的日志填充操作会因为没有足够的日志缓冲区空间而被阻塞，并且事务的持久化操作也会因为磁盘 IO 瓶颈而被阻塞，事务阻塞将增加事务的提交时延。结合图 3.3(b)和图 3.4(a)来看，在处理高负载应用时，大量日志记录会因磁盘 IO 瓶颈而被阻塞，因此事务提交时延急剧升高。

通过上述分析，本项研究发现：组提交时间过大或者过小在动态变化的应用负载中都可能导致事务提交时延变差。因此为了能一直获得最小的事务提交时延，组提交时间 D 应等于$\mathcal{F}(|\bar{G}_i|_D)$，即 $D = \mathcal{F}(|\bar{G}_i|_D)$。

自适应组提交时间。总之，组提交时间对系统的吞吐量和事务提交时延都存在影响。当 $D << \mathcal{F}(|\bar{G}_i|_D)$ 时，日志线程会频繁地进行持久化操作并且大量的日志记录会被阻塞，因此系统吞吐量受限于磁盘 IO，并且事务提交时延会急剧升高。当 $D >> \mathcal{F}(|\bar{G}_i|_D)$ 时，日志线程会出现空闲的状态，虽然系统中不存在任何瓶颈，但是由于磁盘资源没有得到充分的使用并且大量的日志记录处于等待状态，因此事务的提交时延会特别高。结合在吞吐量和事务提交时延上的分析，为了同时获

得较高的吞吐量和较小的事务时延，组提交时间应满足以下条件：

$$D = max(\mathcal{F}(|\bar{G}_i|_D), \frac{1}{IOPS}) \tag{3.3}$$

由于日志组持久化时间$\mathcal{F}(|\bar{G}_i|_D)$会随着负载的变化而改变，因此某一固定时间的组提交协议不能时刻满足公式 3.3。为了能在动态变化的负载下始终保持较高的吞吐量和较小的事务时延，本项研究提出了一种自适应的组提交协议。它通过监控实时的$\mathcal{F}(|\bar{G}_i|_D)$，然后不断调整得到一个新的组提交时间。它的调整规则如下：

$$D = 1/2 * D + 1/2 * max(\mathcal{F}(|\bar{G}|_D), \frac{1}{IOPS}) \tag{3.4}$$

通过不断地调整，组提交时间将无限接近于一次日志组持久化的时间，从而保证了数据库系统事务处理性能的稳定性。最后，本项研究实现了自适应组提交协议并将其集成到可扩展事务日志技术中。

3.4 实验与分析

本项研究基于开源分布式数据库系统 CEDAR 实现了可扩展、自适应事务日志技术（Laser）。CEDAR 具有可扩展的数据存储能力以及并发的事务处理能力。本节评估了可扩展事务日志技术和自适应组提交协议并得出以下结论：

1. 得益于高效原子操作和支持高并发访问数据结构的使用，Laser 展现了良好的多核可扩展性。

2. 负载自适应组提交协议使得 Laser 在动态变化的负载中始终保持了最高吞吐量和最低的事务提交时延。

3.4.1 实验配置

实验环境。本实验将 CEDAR 数据库系统部署在了一台 Linux 服务器上。这台服务器配置了两个 Interl Xeon E5-2630@2.20GHZ 的 CPU 处理器（每个处理器具有

10 个物理核），一个 256G 的内存和一个 SATA 接口的 SSD。

实验对比项。本实验主要对比了以下四种不同的事务日志技术：

1. CEDAR。它表示 CEDAR 数据库系统中采用的集中式日志技术。该技术采用一个日志线程来执行日志序列号分配、日志填充以及日志持久化。

2. AETHER。它是一种比较流行的集中式事务日志技术 [35]。它实现了基于锁的日志序列号分配、并行日志填充方法以及固定的组提交协议。

3. SCAL。它表示本项研究提出的可扩展事务日志技术。该日志方法中采用了传统的固定组提交协议。

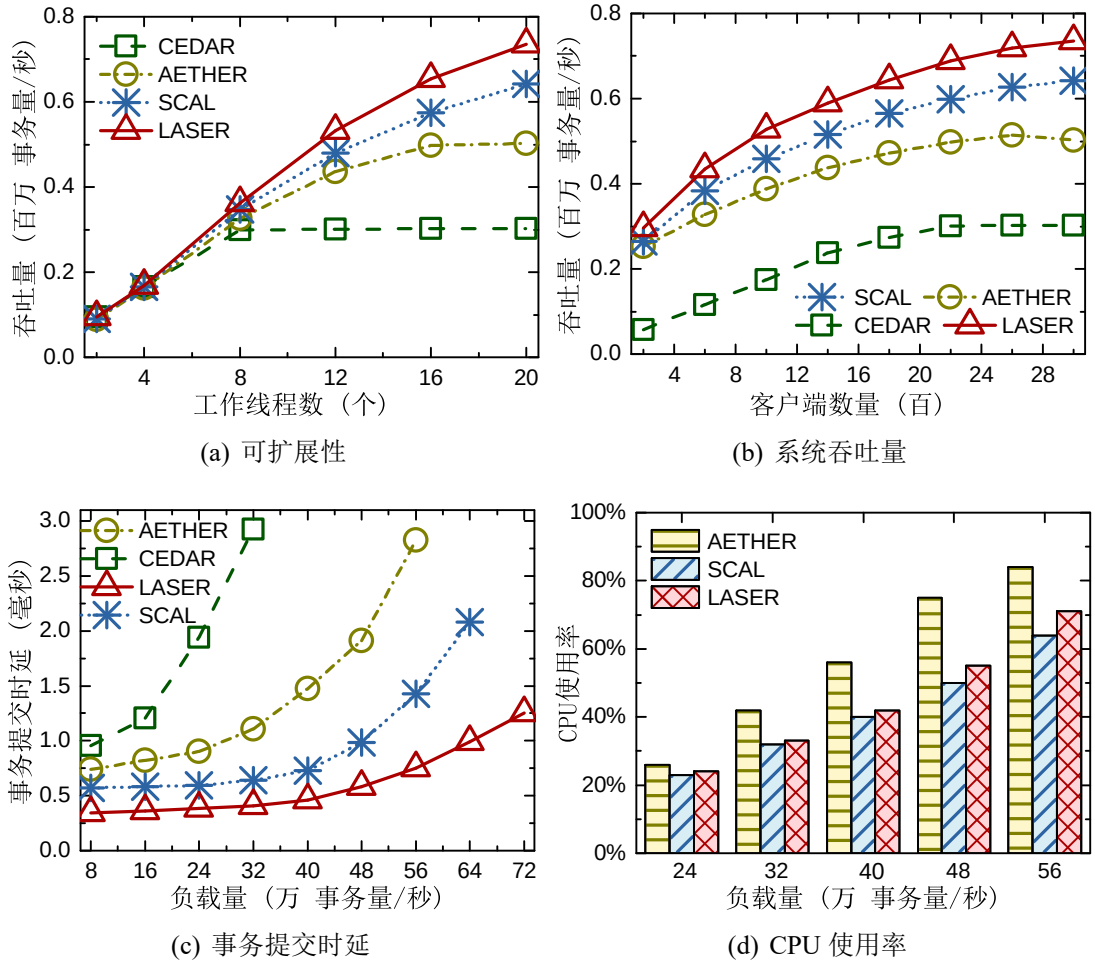
4. LASER。 LASER 结合了可扩展事务日志技术以及自适应的组提交协议。

3.4.2 工作负载

本实验主要采用 YCSB 基准测试 [88] 来验证不同事务日志技术的性能。YCSB 是一种流行的测试标准并广泛地运用于评估云服务系统的读/写性能。由于事务日志技术不涉及事务的 ACID 属性，因此 YCSB 基准测试很适合用于评估数据库系统日志性能。YCSB 基准测试主要包括一张测试表和五种测试负载。本实验对测试表和负载做了一些调整使其更适用于事务日志技术的测试。其中，测试表由一个 64 位的整型列（主键）和一个 10 字节的字符串列组成。表中默认存储了 1 亿条数据记录。本实验设计了一种只包含一个写事务的负载。该负载每次只修改测试表中的一条记录并且事务对测试表的修改服从均匀分布。

3.4.3 实验结果与分析

本实验首先对比了 CEDAR、 AETHER、 SCAL 和 LASER 四种事务日志技术的多核可扩展性、系统吞吐量、事务提交时延以及 CPU 资源利用率，然后验证了自适应组提交协议的有效性。本实验主要涉及三个实验配置项：工作线程数、客户端数量以及组提交时间。默认情况下，各组实验中的工作线程数为 20，客户端数量为 3200，组提交时间为 $0.3ms$。



(a) 可扩展性　(b) 系统吞吐量

(c) 事务提交时延　(d) CPU 使用率

图 3.5: 不同事务日志技术的验证结果

可扩展性。为了测试不同事务日志技术的多核可扩展性，本组实验通过不断地调整工作线程数来观察系统事务吞吐量的变化。图 3.5(a)给出了不同事务日志技术吞吐量的实验结果。最初的时候，随着工作线程数的增加，所有事务日志技术的吞吐量都呈线性增长。然而，当工作线程数量增加到某一值之后，CEDAR 和 AETHER 的吞吐量将不再升高。事务吞吐量达到饱和值的原因是：（1）CEDAR 采用一个日志线程来处理所有的日志操作，如日志序列号分配、日志填充以及日志持久化。随着负载量的不断增多，单个日志线程的处理能力成为了系统的主要瓶颈，从而限制了事务吞吐量的升高；（2） AETHER 使用了基于锁的方式来分配日志序列号。当事务负载增多时，事务日志记录在集中式日志缓冲区上的竞争会变得越来越激烈。

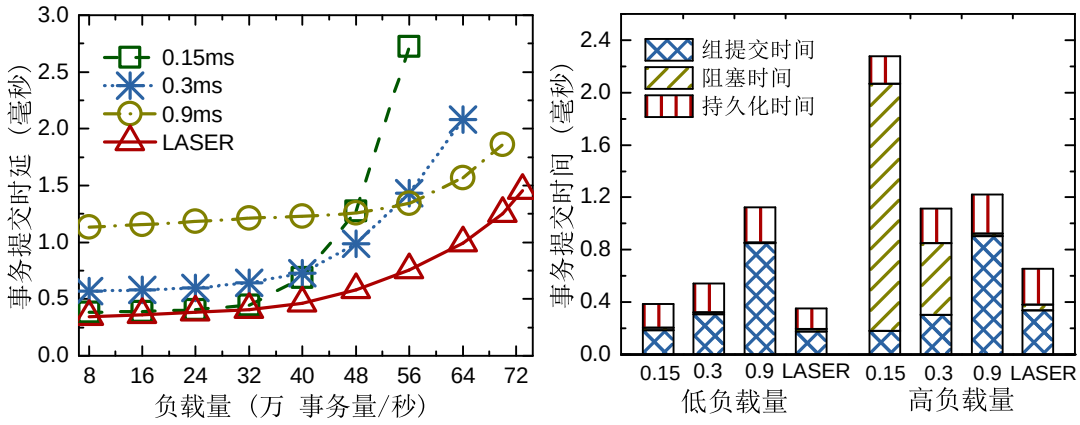
最终日志缓冲区的竞争成为了性能的主要瓶颈，从而限制了系统性能的提升。然而，由于 SCAL 和 LASER 采用了基于原子操作的日志号分配以及工作线程并行日志填充方法，因此它们的吞吐量随着工作线程数的增加呈线性扩展。此外，LASER 实现了负载自适应的组提交协议从而获得了最高的吞吐量。

吞吐量。接下来，本组实验测试了不同事务日志技术的最高吞吐量。图 3.5(b)显示了 CEDAR、AETHER、SCAL 和 LASER 的实验结果。随着客户端数量的增加，所有事务日志技术的事务吞吐量都得到了不断的提升。从图中可以看出，由于 LASER 消除了集中式事务日志中存在的日志缓冲区竞争以及固定组提交问题，因此它始终获得了最大的吞吐量。当客户端数量为为 3200 时，LASER 每秒能处理 73.5 万的负载请求，SCAL 每秒能够处理 64.2 万的负载请求，而 CEDAR 和 AETHER 每秒只能处理 30.1 万和 51.4 万的负载。LASER 的吞吐量是 CEDAR 的 2.4 倍，是 AETHER 的 1.2 倍，是 SCAL 的 1.14 倍。

提交时延。接下来，本组实验测试了不同事务日志的事务提交时延。图 3.5(c)给出了在不同负载量下，CEDAR、AETHER、SCAL 和 LASER 的事务提交时延结果。随着负载量的不断升高，CEDAR、AETHER、SCAL 的事务提交时延急剧增大，而 LASER 的提交时延呈现缓慢的增长趋势。在任何负载量下，LASER 始终获得了最小的事务提交时延，因此它适应于动态变化的负载。当客户端每秒发起 32 万的负载请求时，LASER 的事务提交时延只有 $0.41ms$，而 SCAL、CEDAR 和 AETHER 分别为 $0.64ms$、$2.92ms$ 和 $1.1ms$。在相同负载请求下，LASER 的提交时延比 CEDAR 减少了 86%，比 AETHER 减少了 73%，比 SCAL 减少了 52%。在图 3.5(c)中 CEDAR 的事务提交时延增长最快，这主要是因为 CEDAR 事务日志技术中的单线程成为了系统性能瓶颈从而导致大量事务被阻塞。AETHER 事务提交时延的急剧增长主要是因为集中式日志缓冲区竞争导致了大量事务回滚重试。而 SCAL 提交时延的急剧增长主要是因为固定组提交协议导致大量事务处于等待状态。

CPU 利用率。接下来，本组实验比较了不同事务日志技术的 CPU 使用情况。图 3.5(d)展示了实验结果。图中没有列举出 CEDAR 的 CPU 使用情况。当客户端每

秒产生大于 30 万的负载量时，CEDAR 的 CPU 利用率一直保持在 30% 左右。这主要是因为此时单一日志线程成为了系统瓶颈，从而限制了整体 CPU 的资源利用率。比较 AETHER、 SCAL 和 LASER 的 CPU 使用率，从图中可以看出： AETHER 的整体 CPU 使用率高于 SCAL 和 LASER。然而，AETHER 的最大吞吐量却是最低的。这是因为 AETHER 中存在的大量锁竞争导致产生了额外的 CPU 计算，从而浪费了大量 CPU 资源。 SCAL 的 CPU 使用率低于 LASER，这是因为固定组提交协议导致大量事务处于等待被持久化状态，从而导致大量 CPU 资源被浪费。



(a) 事务提交时延　　(b) 执行时间分析

图 3.6: 自适应组提交协议的验证结果

自适应组提交。接下来本组实验验证了自适应组提交协议的有效性。本组实验比较了自适应组提交与传统固定组提交在动态变化负载下的事务提交时延。本组实验设置了三种固定组提交时间 $0.15ms$，$0.3ms$ 和 $0.9ms$，然后通过不断改变负载量来观察固定组提交协议和自适应组提交协议的事务提交时延。图 3.6(a)给出了它们的测试结果。与传统固定组提交协议相比， LASER 始终能获得最小的事务提交时延。而传统的组提交协议只能在某一个负载量范围内能取得相对较好的事务提交时延。这是因为随着负载量的改变，组提交时间与持久化时间之间的关系会发生变化。当组提交时间远大于日志组持久化时间时，大量事务会处于等待被持久化的状态从而导致事务时延增大。当组提交时间远远小于持久化时间时，大量事务会因没有足够的日志缓冲区空间或磁盘 IO 过载而被阻塞从而导致事务提交时

延升高。因此固定组提交协议不能适用于动态变化的应用负载。而 LASER 实现的自适应组提交协议能够根据实时应用负载动态地调节组提交时间，使得组提交时间和持久化时间时刻保持相等。因此 LASER 能一直获得最优的事务提交时延。

为了进一步验证自适应组提交结论的正确性，本组实验监控了不同组提交协议的事务日志执行时间，并将它们分成三个部分：组提交时间、持久化时间以及事务被阻塞的时间。图 3.6(b)分别显示了事务日志技术处理低负载请求和高负载请求时的事务执行时间。从图中可以看出，当组提交时间大于持久化时间时，组提交时间是组成事务执行时延的最主要部分。这种情况主要表现在组提交时间为 $0.9ms$ 的组提交协议中。当组提交时间小于持久化时间时，事务阻塞的时间会突然增加从而导致事务时延升高。这种情况主要出现在组提交时间为 $0.15ms$ 的提交协议中。而 LASER 在任何负载下都保证了组提交时间与持久化时间相等，因此它不存在事务被阻塞或者事务处于等待的状态从而始终获得最低的提交时延。

3.5 本章小结

本项研究首先总结了传统集中式 ARIES 日志技术在多核环境下存在的两个性能瓶颈：（1）基于锁的日志序列号分配方式使得事务在集中式日志缓冲上的竞争变得更加激烈；（2）传统采用固定时间的组提交协议不适用于动态变化的负载。为了解决上述性能瓶颈，本项研究提出了一种可扩展、自适应的事务日志技术。可扩展事务日志技术主要采用并行日志填充的方式和基于原子操作的日志号分配方法来解决传统事务日志的集中式日志缓冲区竞争问题。为了解决并行填充方式带来的日志空洞问题，本项研究实现了一个支持高并发访问的数据结构来追踪日志记录中的空洞，从而保证了数据库系统的正确性。此外，自适应组提交协议根据实时的工作负载来调整系统的组提交时间，从而保证数据库系统在动态负载中能够一直获得最大的吞吐量和最小的事务提交时延。最后本项研究基于开源分布式数据库系统实现了该事务日志技术并对其进行验证。实验结果表明：可扩展、自适应事务日志技术比传统的日志技术具有更好的多核可扩展性和事务处理性能。

第四章　面向可扩展存储的并行事务日志技术

4.1　引言

上一项研究实现了一种可扩展、自适应的事务日志技术，该技术解决了传统集中式事务日志技术存在的日志缓冲区竞争和固定组提交问题，从而提高了数据库系统的可扩展性。然而，随着可扩展事务处理能力的不断提升，集中式事务日志技术面临了一个新的性能瓶颈：有限的磁盘 IO 带宽。为了保证数据库系统的可靠性，集中式事务日志技术串行地将所有日志记录存入一块磁盘 [21, 35]。磁盘的顺序写性能决定了数据库系统事务处理的最大吞吐量。在多核（众核）平台下，基于可扩展事务日志技术的数据库系统每秒能够产生超过 1.1GB 的日志量 [15, 20]。然而，一块磁盘每秒最多只能处理几百兆的顺序写请求。因此有限的磁盘 IO 带宽限制了数据库系统的性能。

为了解决集中式事务日志技术的有限 IO 带宽瓶颈，本项研究提出了一种面向可扩展存储的并行事务日志技术。该技术使用多个日志缓冲区和多块磁盘来代替原来的集中式设计，并且允许并行地填充和持久化日志记录。然而，在数据库系统中实现一个并行事务日志技术并不容易。它主要面临两个挑战：（1）如何保证事务之间的执行顺序以及事务的正确提交；（2）日志记录如何均匀地分布到多个存储磁盘。对于第一个挑战，本项研究提出了一个新的事务日志号（GSN）和一种持久化的组提交协议（persistent group commit）。GSN 追踪了事务之间的偏序顺序并保证了事务的先写后读依赖（RAW）、写后写依赖（WAW）和先读后写依赖（WAR）。持久化组提交协议规定只有当某个事务的所有日志记录以及它所依赖事务的日志记录都已持久化之后，该事务才能被提交。对于第二个挑战，本项研究采用基于数据的日志分区策略，即修改同一数据项的日志记录存储在相同的缓冲区和磁盘中。这种日志分区策略便于 GSN 计算并且有利于实现并行的日志恢复。但是，这种分区策略存在一个问题：当应用负载存在倾斜或者热点数据时，系统性能会受到影

响。为了解决这个问题，本项研究提出了一种负载感知的日志分组策略。它能根据实时的应用负载选择一个最合适的分组策略，从而使得日志记录能够均匀地分布到所有的存储设备中。综上所述。本项研究主要做出了以下贡献：

1. 本项研究发现随着事务日志技术可扩展性的提升，事务日志的主要瓶颈已由集中式日志缓冲区竞争和固定组提交转变为有限的磁盘 IO 带宽。为此，本项研究提出了一种面向可扩展存储的的并行事务日志技术。

2. 本项研究实现了一个偏序的事务日志号（GSN）和持久化的组提交协议保证了数据库系统的正确性和可恢复性，并且提出了一种负载感知的日志分区策略避免了负载倾斜和跨分区事务对系统事务处理性能的影响。

3. 本项研究结合并行事务日志技术、乐观多版本并发控制协议以及并行恢复方法实现了一个内存事务处理原型系统（Plover）。此外，本项研究使用 YCSB 和 TPC-C 基准测试评估了并行事务日志技术的性能。实验结果表明，与集中式事务日志相比，并行日志技术的性能能够随着磁盘数量的增加呈线性扩展。

4.2 系统框架

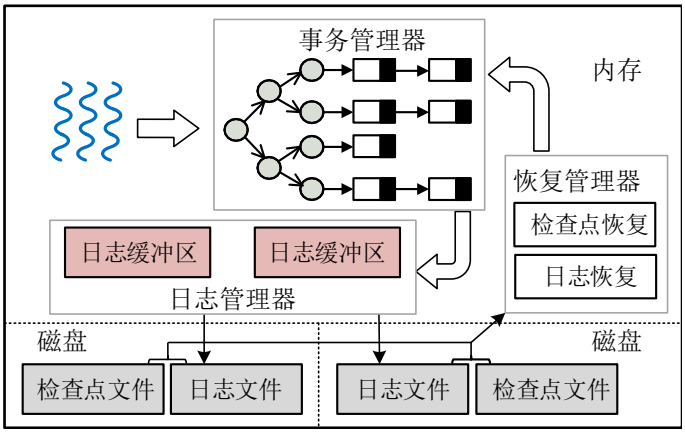


图 4.1: 内存事务处理系统原型框架

本节主要介绍了内存事务处理原型系统 Plover 的整体框架。如图 4.1所示，它主要包含一个事务管理器、一个日志管理器和一个恢复管理器。事务管理器实现

了一种乐观的多版本并发控制协议，并且提供了可序列化的事务隔离级别。数据的修改存储在一个内存数据表结构中。该表由一个索引结构和链表结构组成，数据的修改版本存储在链表结构中。日志管理器实现了一种并行的事务日志技术。它将所有日志记录并行地写入多个日志缓冲区和多块存储设备，并实现了一种基于偏序事务日志号的持久化组提交协议，该协议保证了事务的正确提交。恢复管理器实现了一种并行的日志恢复方法。它首先将产生的检查点文件并行地写入多块磁盘，然后在系统发生故障后能够并行地从多块磁盘中读取检查点文件和日志文件，最后将数据库系统恢复到故障之前的正确状态。

接下来，本章将分别介绍并行事务日志技术，以及基于并行事务日志的并发控制协议和日志恢复方法的具体设计与实现。

4.3 并行事务日志

并行事务日志技术的目的是为了消除数据库系统中集中式事务日志的性能瓶颈。它主要采用将事务日志记录并行写入多个日志缓冲区的方式来解决集中式日志缓冲区的竞争问题，同时采用将日志记录存储到多块存储设备的方式来解决单一磁盘有限的 IO 带宽问题。本节首先介绍了并行事务日志技术的整体框架，然后详细地描述了它的执行流程，最后介绍了一种负载感知的日志分区策略。

4.3.1 整体架构

并行事务日志技术主要包含多个日志缓冲区、多块存储设备以及两种线程：多个工作线程和多个日志线程。工作线程和日志缓冲区之间存在一个多对多的关系，而日志线程和日志缓冲区以及存储设备存在一个一对一的关系。

日志分区。该技术采用了一种基于数据的日志分区策略将日志记录填充到多个日志缓冲区中。该日志分区策略首先将数据项按主键划分为多个数据分区，然后将修改同一数据分区的事务日志记录写入相同的日志缓冲区和相同的存储设备。

日志流程。并行事务日志技术的执行流程主要分为三个阶段：日志准备阶段，

日志持久化阶段以及事务提交阶段。在日志准备阶段，每个工作线程根据事务访问的数据项生成一条或者多条事务日志记录，然后根据分区策略将日志记录写入相应的日志缓冲区中。也就是说，如果一个事务访问了两个分区中的数据，那么工作线程会产生两条日志记录然后将它们写入两个不同的日志缓冲区。在日志持久化阶段，每个日志线程分别将对应日志缓冲区中的日志记录写入一块磁盘。在事务提交阶段，工作线程根据事务提交协议将可提交事务的结果返回给客户端。提交协议规定：只有当一个事务的所有日志记录以及它依赖的所有事务日志记录都已持久化之后，这个事务才能称为可提交事务。此外，并行事务日志技术还实现了上项研究提出的日志空洞检测方法和自适应组提交协议。

主要挑战。并行事务日志技术主要存在两个挑战：

挑战 1：如何在分散的日志记录中追踪事务的执行顺序。由于日志序列号（LSN）只能表示事务在一个日志缓冲区上的先后顺序，并不能追踪事务在多个日志缓冲区上的相对顺序，因此传统事务日志技术的日志序列号（LSN）不再适用于并行事务日志技术。为此，本项研究提出了一个新的事务日志号（GSN）。GSN 是基于逻辑时钟 [89] 实现的，它保证了事务之间的偏序执行顺序，即 GSN 追踪了事务之间的写后写依赖（WAW）、先写后读依赖（RAW）和先读后写依赖（WAR）。

挑战 2：如何保证事务提交的正确性。在并行事务日志技术中，同一个事务的日志记录或者相互依赖事务的日志记录可能写入不同的日志缓冲区。由于每个日志缓冲区的日志记录是由一个单独的日志线程进行持久化的并且每个日志线程并不知道其他线程的持久化情况，因此日志线程无法判断当前已持久化日志对应的事务是否能够提交。为了解决此问题，本项研究提出了一种持久化的组提交协议。该协议使用一个单独的监控线程周期地访问所有日志线程的持久化情况，然后根据持久化情况确定哪些事务是可提交事务，从而保证了事务提交的正确性。

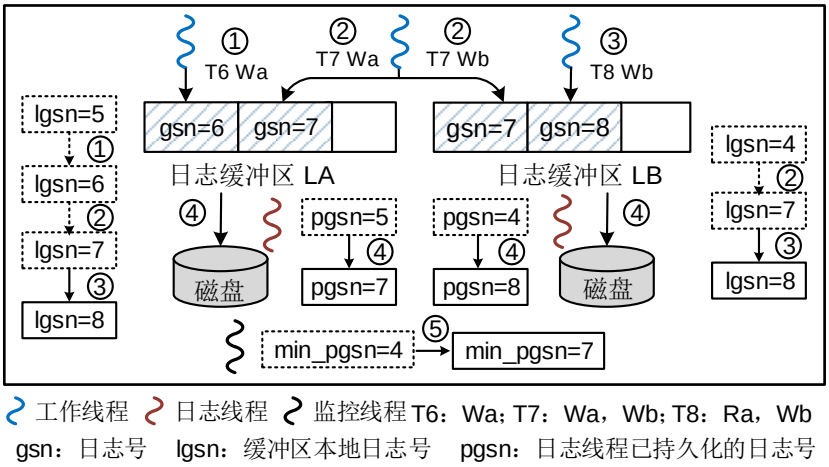


图 4.2: 并行事务日志执行流程

4.3.2 执行流程

接下来，本节结合 GSN 和持久化组提交协议详细地介绍了并行事务日志技术的执行流程。图 4.2给出了并行事务日志技术的整体框架和具体执行流程。图中主要包含了两个日志缓冲区 LA 和 LB 以及两块存储设备。每个日志缓冲区和存储设备对应一个日志线程。假设数据简单地按数据项 a，b 进行划分，其中 $a \in LA$，$b \in LB$。事务 T_6、T_7 和 T_8 的执行过程描述如下：

日志填充。当事务进入日志准备阶段时，工作线程根据事务访问的数据分区生成相应的日志记录。由于 T_6 和 T_8 只修改了一个分区上的数据，因此它们只产生一条日志记录，而 T_7 修改了两个分区上的数据项，因此它需要产生两条日志记录。

在并行填充日志记录之前，工作线程需要为每条日志记录分配一个日志号 GSN。并行事务日志技术在每个日志缓冲区上维护了一个本地的缓冲区日志号（$lgsn$）。日志记录的 GSN 根据日志缓冲区上的日志号进行分配。比如，如果一个事务访问 m 个数据分区并产生 m 条日志记录，那么工作线程为这个事务的 m 条日志记录分配一个相同的日志号（$Max_{1 \le i \le m}(lgsn_i) + 1$），然后更新 m 个日志缓冲区的本地日志号并将 GSN 保存在事务和事务修改的数据项中。

如图 4.2中的步骤①，由于事务 T_6 只修改了数据项 a，因此在分配日志号时，它只需要获取日志缓冲区 LA 的本地日志号$lgsn_a$ 然后计算日志号为 $lgsn_a + 1 = 6$，最

后将 LA 的本地日志号和数据项 a 的日志号更新为 6。对于同时修改数据项 a，b 的事务 T_7 来说，它需要同时获取日志缓冲区 LA 和 LB 的本地日志号$lgsn_a$，$lgsn_b$，然后为它生成的两条日志记录计算日志号 GSN 为 $max(lgsn_a = 6, lgsn_b = 4) + 1 = 7$，最后将两个缓冲区的本地日志号都设置为 7, 如步骤②所示。对于事务 T_8 来说，虽然它只修改数据项 b 且只生成一条日志记录，但是由于它读取了数据项 a，因此在分配日志号时，它仍需要同时访问日志缓冲区 LA 和 LB 的本地日志号，然后设置日志号 GSN 为 $max(lgsn_a = 7, lgsn_b = 7) + 1 = 8$ 并将两个日志缓冲区的本地日志号也设置成 8，如步骤③所示。在多核数据库系统中，多个事务可能同时修改日志缓冲区的本地日志号，因此事务日志技术实现了基于 latch 的方式来避免事务之间的写写冲突。算法 5给出了事务日志号 GSN 分配的伪代码。算法第 2 行到第 13 行首先利用原子指令获取事务对应日志缓冲区上的最大本地日志号$lgsn$，然后计算事务的日志号以及日志记录在缓冲区中的存储空间，最后更新日志缓冲区的本地日志号。算法第 14 到 16 行更新了事务访问数据项上的日志号。

日志持久化。当每个日志缓冲区中积累一定量的日志记录后，日志线程各自采用自适应的组提交协议将日志记录写入对应的存储设备。完成持久化操作之后，每个日志线程记录当前已持久化日志记录中最大的日志号（$pgsn$）。如图 4.2的步骤④，两个日志线程分别将日志缓冲区 LA，LB 中的日志记录写入对应的存储设备，然后分别更新它们本地的最大已持久化日志号 $pgsn$ 为 7 和 8。

事务提交。在并行事务日志技术中，工作线程无法根据某一个日志线程的最大已持久化日志号 $pgsn$ 来判断哪些事务是可提交事务。因此，本项研究提出了一种可持久化的组提交协议。该协议使用了一个监控线程周期性地监测所有日志线程的持久化情况（$pgsn$），然后计算出一个最小的已持久化的日志号（min_pgsn）。之后，工作线程根据min_pgsn 来判断事务是否能够提交。当且仅当日志号小于等于min_pgsn 的事务才能被提交。如图中的步骤④，监控线程首先读取两个日志线程的$pgsn$，然后将min_pgsn 更新为它们中最小的值 7，之后日志号小于等于 7 的事务可以被提交，即事务 T_6 和 T_7 可以提交。

算法 5: 日志号 GSN 分配

```
输入: 事务T，数据项集合 E
输入: 事务所属的日志缓冲区集合 LS
输入: 事务日志记录大小 len
输入: 数据项 e，日志缓冲区 l
输出: 日志号分配成功或者失败
1  base = 0
2  while CAS( LS.latch, false, true ) do
       /* 获取事务所属日志缓冲区中最大的本地日志号 */
3      for l in LS do
4          base = max(l.lgsn, base);
5      end
       /* 计算事务日志号 */
6      T.gsn = base + 1;
7      for l in LS do
8          l.lgsn = T.gsn;
9          FETCH_ADD( l.offset, len );
10     end
11     COMPILER_BARRIER();
12     LS.latch = false;
13 end
   /* 更新事务访问数据项的日志号 */
14 for e in E do
15     e.gsn = T.gsn
16 end
```

4.3.3 日志分区

前文介绍了并行日志技术采用了一种基于数据的日志分区策略。然而，这种分区策略存在一个明显的缺陷：如果负载中存在大量的跨分区事务或者负载中存在访问倾斜，那么基于分区的数据库系统性能将受到影响。本节详细地分析了基于数据的日志分区策略对系统产生的具体影响，然后提出了一种负载感知的日志分区策略，使得负载能够均匀地分布到各个日志分区上。

问题描述。基于数据的日志分区策略使得系统可能遭受两个性能问题：（1）跨分区事务。在并行事务日志技术中，每个日志缓冲区对应一个数据分区。这里把只访问一个数据分区的事务称为单分区事务，而把访问多个数据分区的事务称为跨分区事务。对于单分区事务，由于它们在事务日志执行过程中只涉及一个日志缓冲

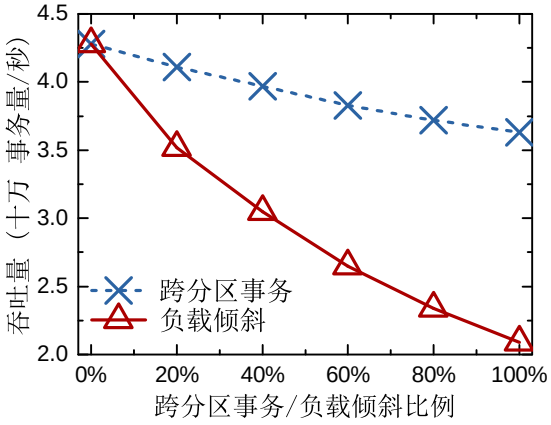


图 4.3: 跨分区事务和负载倾斜对性能的影响

区并且相互之间不存在冲突，因此它们能够被并行地执行。而对于跨分区事务来说，由于它们需要同时访问多个日志缓冲区来分配日志号，因此它的计算代价会更高甚至会阻塞其他事务的执行。（2）负载倾斜。如果应用负载在日志分区之间分布不均匀，那么大多数事务日志记录可能写入同一个日志缓冲区和同一块存储设备。这会使得事务在某一个日志缓冲区上的竞争变得激烈，甚至会使得某一存储设备遭受过多的 IO 压力从而限制了系统的性能。

为了更加详细地阐述跨分区事务和负载倾斜对系统性能造成的影响，本项研究使用 YCSB 基准测试来测试并行事务日志技术的吞吐量。图 4.3给出了测试结果。在本组实验中，并行日志技术采用了两个日志缓冲区和两块存储设备（SSD）。此外，实验使用了一个包含 1000 万条记录的测试表和一个包含两个写事务的负载。本组实验主要通过调整跨分区事务比例和负载倾斜比例来观察系统吞吐量的变化。默认情况下，它们的值设置为 0 并且每组实验只修改其中一个影响因素的值。从图中可以看出，随着跨分区事务比例和负载倾斜比例的增加，系统吞吐量逐渐减少。当比例达到 100% 时，系统性能比不存在跨分区事务时的吞吐量降低了 15%，比不存在负载倾斜时降低了 52%。

分区设计。为了解决基于数据的日志分区策略所带来的性能问题，本项研究提出了一种负载感知的日志分区策略。它主要根据实时的负载情况来创建一个均衡的日志分区。实现这种分区策略主要包括两个步骤：（1）对实时的负载进行建

模；（2）根据模型生成一个均衡的分区策略。对于第一个步骤，类似于之前的工作 [90]，本分区策略采用了图 $G = (V, E)$ 来模拟实时的应用负载，其中每个顶点 $v \in V$ 表示一个数据项，顶点 v_i 和顶点 v_j 之间的边 $e_{ij} \in E$ 表示这两个数据项被同一事务访问。每条边上记录了一个权重值 w_e 用来表示连接的两个数据项被一起访问的频率。对于第二个步骤，在图建立以后，负载感知分区策略使用 K 路平衡最小割的分区方法 [91] 将图G 分成 k 个独立的分区。该方法保证了基于生成的日志分区，负载中存在最少的跨分区事务同时负载能够均匀地分布在所有的日志分区。在实现过程中，日志分区策略主要采用了 METIS 包 [92]，它具有高效的基于图的计算能力并且能够生成均衡的日志分区。

但值得注意的是，生成的均衡日志分区可能会随着应用负载的变化而变得不再适用。当跨分区事务的数量或者负载倾斜的比例超过某一阀值时，系统需要重新生成一个日志分区。重新生成的日志分区会使得日志记录写入一个新的日志缓冲区，这会带来另一个问题：对于单分区事务，它的日志记录只会写入一个日志缓冲区，而基于新日志分区策略生成的日志号可能小于之前基于旧分区产生的日志号。为了解决这个问题，并行事务日志技术在产生新日志分区之前执行一个分布式事务，即将所有日志缓冲区的本地日志号（$lgsn$）更新为它们中的最大值。这样保证了修改同一数据项的日志记录一定遵循全序关系。

4.4 并发控制与恢复

本节主要介绍内存事务处理原型系统 Plover 中的并发控制协议和日志恢复方法。本节首先描述了与并行事务日志技术相结合的多版本乐观并发控制协议的设计与实现，然后介绍了基于并行事务日志技术设计的并行恢复方法。

4.4.1 并发控制

在实现并行事务日志技术的数据库系统中可以采用任意的并发控制协议。但是，本项研究主要实现了一个乐观多版本并发控制（MVOCC）协议 [93, 94] 并提

供了可串行化的事务隔离级别。MVOCC 通常采用一种集中式的方式来分配事务提交时间戳。该提交时间戳确定了事务之间的提交顺序 [1]。由于传统集中式的方式会影响数据库系统的可扩展性，因此本项研究实现了一种分布式的事务提交时间戳分配方法。该方法直接用 GSN 来表示事务的提交时间戳。

为了保证在发生事务故障或者系统故障后系统能够恢复到一个正确的状态，数据库系统需要保证事务的执行遵循可串行化和严格性 [1]。严格性要求事务的提交顺序必须满足写后写依赖（WAW）、先写后读依赖（RAW）和先读后写依赖（WAR）。由于日志号 GSN 保证了这三种依赖，因此本项研究可以直接用 GSN 来代替提交时间戳。接下来，本节将详细地描述如何结合 GSN 来实现并发控制协议。

乐观多版本并发控制协议主要包含三个阶段。在读取阶段时，事务 T 在内存数据上执行读取和修改操作。对于读操作，事务 T 首先读取访问数据分区上的$lgsn$，然后以所有数据分区中最大的$lgsn$ 作为事务的开始时间，最后基于事务开始时间读取正确的数据项版本。对于写操作，它首先对要修改的数据项进行加锁，加锁成功之后创建数据项的一个新版本。锁的使用主要是用来避免事务的写写冲突。

在验证阶段，事务管理器需要判断事务的执行是否满足可串行化。如果事务的执行满足以下两个条件：（1）事务 T 能够获得所有修改数据项的写锁；（2）事务 T 读取的数据项没有被其他事务修改，那么事务 T 允许提交。否则，T 被回滚。一旦事务 T 验证成功，它就会生成一个新的 GSN 然后进入事务日志执行阶段。

在写阶段，事务 T 释放掉所有的写锁使得它生成的新版本能够对其他事务可见。每个数据项会存储一个 GSN 来表示它的版本信息。

4.4.2 日志恢复

接下来，本节将详细地介绍恢复管理器中的并行恢复方法。当发生系统故障（断电或者操作系统损坏）之后，数据库系统主要依靠检查点和日志文件将数据库恢复到一个正确的状态。

检查点。为了加速系统恢复，数据库系统通常会定期地将数据库状态存入磁

盘。这些存储的数据库状态信息被称为检查点。与事务日志技术一样，检查点也采用相同的策略进行分区然后将每个分区的数据信息写入对应的一块存储设备。每个检查点分区由一个检查点线程负责处理。

基于内存多版本的数据表结构，系统在生成检查点的时候不会阻塞正常事务的执行。当开始执行检查点时，检查点管理器首先读取当前系统中最小的已持久化日志号min_pgsn并标记为$cgsn$，$cgsn$表示可以被写入检查点文件的最大数据版本号。然后，管理器启动n个检查点线程，其中n等于检查点分区数量。每个检查点线程将自己对应分区的快照版本写入m个检查点文件中。当所有检查点完成持久化之后，检查点管理器将$cgsn$存储元信息文件中。

恢复流程。当系统发生故障之后，数据库系统通过重构最新检查点文件中的数据以及日志文件中的日志记录来恢复数据库状态。恢复检查点时，恢复管理器首先从元信息文件中读取$cgsn$，它表示日志恢复的起始点，然后启动$n*m$个线程并行地恢复所有最新存储的检查点文件。恢复日志记录时，所有恢复线程并行地从多块存储设备中读取日志文件，然后并行地恢复日志号GSN大于$cgsn$的日志记录。

但是，在并行事务日志技术中可能存在以下风险：假设一个事务产生了两条日志记录，在系统发生崩溃时，只有一条日志记录被持久化了，而另一条日志记录未被持久化。在日志恢复阶段，如果日志恢复线程恢复了该事务持久化的日志记录，那么数据库系统将处于不正确的状态。为了解决这个问题，持久化组提交协议中的监控线程在计算出min_pgsn之后将其写入存储设备。在执行日志恢复时，恢复线程只需要重构日志号满足$cgsn < GSN \leq min_pgsn$的日志记录即可。

4.5 实验与分析

本项研究实现了一个内存事务处理原型系统Plover，该原型系统中主要包含一个并行事务日志技术、一个乐观的多版本并发控制协议和一个并行的日志恢复方法。本项研究基于原型系统进行一些对比实验，并得出了以下结论：

1. 基于多个日志缓冲区和多块存储设备的设计，并行事务日志的处理性能能够

随着磁盘数量的增多呈线性增长。

2. 得益于负载感知的日志分区策略，并行事务日志技术在拥有倾斜和跨分区事务的应用负载中仍能获得较好的吞吐量。

3. 基于并行日志恢复方法，数据库系统能够更快地从故障中恢复并提供服务。

4.5.1 实验配置

实验环境。本实验运行在一台 Linux 服务器上。该服务器包含了两个 Intel Xeon E5-2630 v4 @2.20GHZ 的 CPU 处理器（总共 20 个物理核），256GB 的 DRAM 和 4 块 SATA 接口的 SSD。所有的应用负载数据都存储在内存中。

实验对比项。本实验主要对比了以下五种不同的事务日志技术：

1. CENTR。它表示一种集中式事务日志技术。该技术使用了一个集中式的日志缓冲区和一块存储设备来存储日志记录，并且实现了基于 latch 的日志序列号分配和串行日志填充方法。

2. COMM。它表示一种新型的逻辑事务日志技术 [76]。该技术使用一个集中式日志缓冲区存储基于事务逻辑形式的日志记录。该日志记录不存储数据项的更新值，而只记录事务的存储过程标识符以及事务的请求参数。

3. NVM-D。它表示一种基于非易失内存（NVM）的并行日志技术。该技术允许工作线程直接将日志记录写入 NVM 中。为了避免频繁的 IO 操作，本实验为每个工作线程分配了一个私有的日志缓冲区，然后允许日志记录被批量地写入普通的存储设备（SSD）。

4. PARA+HASH。它表示本项研究提出的事务日志技术。该技术使用多个日志缓冲区和多块存储设备来并行地存储事务日志记录。此外，该技术实现了一种基于哈希的日志分区策略。

5. PARA+OPT。它表示集成负载感知日志分区策略的并行事务日志技术。

默认情况下，所有的对比项（除了 CENTR 和 COMM）配置了两块 SSD。在所有实验中，存储设备的数量等于日志缓冲区的数量和日志线程的数量。

4.5.2 工作负载

本实验使用了 YCSB 和 TPC-C 两种基准测试来评估并行事务日志技术。

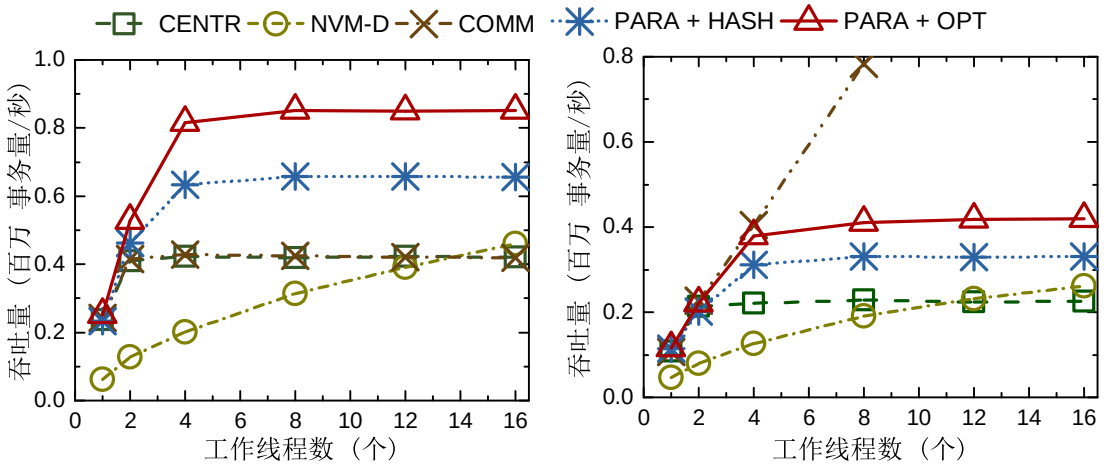
YCSB。YCSB 基准测试 [88] 是由雅虎公司开发的，它主要用于测试键值对系统的读写性能。YCSB 基准测试包含一张测试表，该表是由一个 64 位的整型列（主键）和十个 100 字节的字符串列组成。在本实验中，测试表中包含了 1000 万条数据记录。此外，本实验使用了一种只写负载。该负载中只包含一个写事务并且每个事务每次只修改测试表中的一条数据记录。只写负载服从 Zipfian 分布，其中负载倾斜比例为 $\zeta = 0.6$（0.6 表示大约有 40% 的写事务会集中修改 10% 的数据记录）。

TPC-C。 TPC-C 是一种常用于评估在线事务处理系统的基准测试，它模拟了一个以仓库为中心的订单处理系统 [95]。TPC-C 基准测试包含 9 张测试表和 5 种事务（三种读写事务和两种只读事务），其中读写事务的比例达到了 92%。本实验设置了 20 个仓库数量并且只使用了三种读写事务。

4.5.3 实验结果与分析

本实验首先对比了 CENTR、 COMM、 NVM-D、 PARA+HASH 和 PARA+OPT 五种事务日志技术的系统吞吐量和可扩展性，然后验证了并行恢复方法的有效性。

吞吐量。本节比较了不同事务日志技术的吞吐量。图 4.4显示了所有对比项在 YCSB 和 TPC-C 基准测试中的实验结果。在 YCSB 负载中，随着工作线程数量的增加， CENTR、 COMM、 PARA+HASH 和 PARA+OPT 的吞吐量先逐步升高，然后趋于稳定（达到一个饱和值）。呈现这种趋势的主要原因是当事务量增多时，SSD 的带宽成为了限制系统性能的主要瓶颈。由于 CENTR 只使用了一块 SSD，有限的带宽使得 CENTR 的吞吐量最低。同样地， COMM 的吞吐量也受到了单个 SSD 的限制。虽然 COMM 的日志只记录请求参数和存储过程标志符，但是在 YCSB 这种逻辑简单的负载中， COMM 仍然产生了与传统事务日志技术相



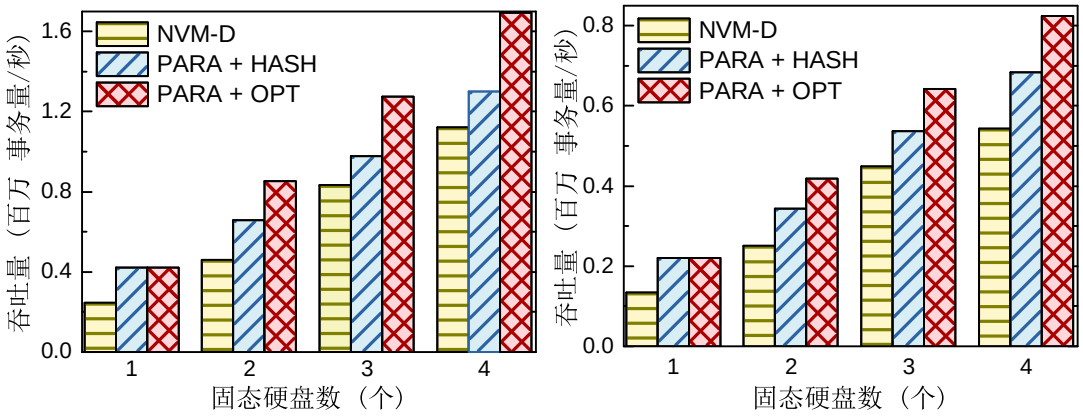
(a) 在 YCSB 负载下的吞吐量　　(b) 在 TPCC 负载下的吞吐量

图 4.4: 不同事务日志技术的吞吐量

同的日志量。因此， COMM 表现出了和 CENTR 一样的性能趋势。得益于使用了两块 SSD， PARA+HASH 和 PARA+OPT 具有更高的吞吐量，其中 PARA+OPT 的性能是 CENTR 的 2 倍。与 PARA+HASH 相比较， PARA+OPT 展现出了更高的吞吐量。这是因为 PARA+OPT 实现了负载感知的日志分区策略，该分区策略能够使倾斜的 YCSB 负载均匀地分布到两块 SSD 上。然而，在基于哈希的日志分区策略中，大量的日志记录会集中地写入某一块 SSD 和某一个日志缓冲区。因此，日志缓冲区竞争和有限的 IO 带宽最终降低了 PARA+HASH 的吞吐量。对于 NVM-D 来说，尽管它的吞吐量随着工作线程数的增加而升高，但是它的吞吐量始终无法与 PARA+OPT 相媲美。甚至，当工作线程数量少于 10 时， NVM-D 的吞吐量比集中式事务日志的性能还要低。 NVM-D 之所以展示出较低的吞吐量是因为 NVM-D 由工作线程负责日志持久化，这导致了大量的上下文切换和额外的调度代价。

在 TPC-C 负载中，随着工作线程数的增加，CENTR、PARA+HASH 和 PARA+OPT 的吞吐量先不断地升高然后趋于稳定。与 YCSB 的测试结果一样， PARA+OPT 获得了比 PARA+OPT 和 NVM-D 更高的吞吐量。CENTR 的性能仍然受限于单个 SSD 的 IO 带宽，并且最大吞吐量只有 PARA+OPT 的一半。然而， COMM 展现了最好的性能。这主要是因为在 TPC-C 这种负载下，基于逻辑日志产生的日志量特别小

并且不会对存储设备造成太大的压力。尽管 COMM 具有更好的日志性能，但是由于它严重依赖了事务存储过程，因此不能广泛地运用到商业数据库中。除此之外，COMM 最大的问题是在系统崩溃之后，数据库系统需要按照日志记录顺序重新执行所有的事务 [96, 97]，因此它的恢复性能会特别糟糕。



(a) YCSB 中的可扩展性　　(b) TPCC 中的可扩展性

图 4.5: 不同事务日志技术的可扩展性

可扩展性。接下来，本组实验比较了所有事务日志技术在 YCSB 和 TPCC 负载下的可扩展性。本实验主要通过改变 SSD 的数量来观察事务日志技术的吞吐量变化。其中，SSD 的数量等于系统内存中的日志缓冲区个数和日志线程数量。本组实验没有给出 CENTR 和 COMM 的实验结果。因为它们的性能不会随着 SSD 数量的改变而改变，因此它们不具有可扩展性。图 4.5给出了 PARA+OPT、 PARA+HASH 和 NVM-D 的实验结果。在 YCSB 和 TPCC 负载下，随着 SSD 数量的增多，所有事务日志技术的吞吐量都得到了提升，但 PARA+OPT 表现出了最好的可扩展性。对于 NVM-D 来说，它主要是基于新型非易失存储内存（NVM）来设计并实现的。它采用的工作线程直接持久化日志的方式并不适用于部署在普通存储设备（如 SSD，HDD）上的系统。对于 PARA+HASH 来说，YCSB 和 TPC-C 负载在基于哈希的日志分组策略上存在跨分区事务和负载倾斜问题，因此它获得了较差的吞吐量。

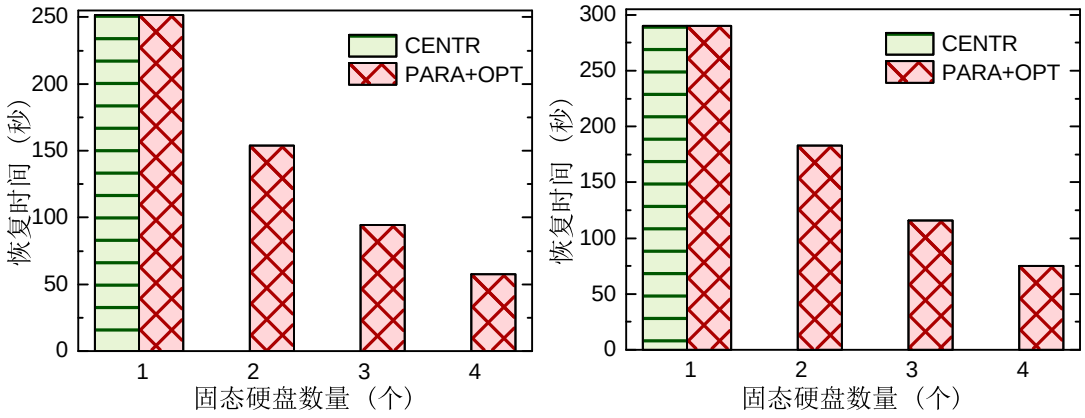
恢复性能。最后，本组实验验证了并行日志恢复方法的高效性。本组实验首先使用了 YCSB 和 TPC-C 来测试 CENTR 和 PARA+OPT 的恢复时间，然后比较了它

表 4.1: YCSB 负载中的日志恢复性能

事务日志技术	检查点恢复时间	日志恢复时间	总恢复时间
CENTR	81.72s	169.73s	251.45s
PARA+OPT	50.02s	103.87s	153.89s

表 4.2: TPCC 负载中的日志恢复性能

事务日志技术	检查点恢复时间	日志恢复时间	总恢复时间
CENTR	38.22s	251.63s	289.85s
PARA+OPT	24.14s	158.85s	182.99s



(a) 在 YCSB 负载中的可扩展性　　(b) 在 TPC-C 负载中的可扩展性

图 4.6: 不同事务日志技术的恢复性能

们的可扩展性。本组实验启动了 20 个线程来执行检查点恢复和日志恢复。

在测试两种事务日志技术的恢复时间时， PARA+OPT 采用了两块 SSD 来存储检查点文件和日志文件。表 4.1和表 4.2分别显示了事务日志技术在 YCSB 和 TPCC 中的恢复时间。在 YCSB 负载中，系统中存在 26G 的检查点文件和 54G 的日志文件， CENTR 和 PARA+OPT 要读取所有的检查点文件和日志文件并将它们恢复在系统中。从表中可以看出，无论是恢复检查点还是恢复日志文件， PARA+OPT 所花的时间都比 CENTR 少。这主要是因为在 CENTR 事务日志技术中，所有的检查点文件和日志文件都存储在一块存储设备上。在执行恢复时，单 SSD 有限的 IO 带宽决定了 CENTR 的恢复性能。而由于能够并行地执行恢复， PARA+OPT 使用了更少的恢复时间。它的恢复性能是 CENTR 的 1.63 倍。在 TPC-C 负载中， CENTR 和 PARA+OPT 需要恢复 12G 的检查点文件和 79G 的日志文件。和 YCSB 负载中的结

果一样， PARA+OPT 使用了更少的恢复时间。

为了验证不同事务日志技术的恢复可扩展性，本实验主要通过改变 SSD 的数量来观察它们总恢复时间的变化。所有用于恢复的检查点文件和日志文件均匀地分布在所有的存储设备 SSD 中。图 4.6给出了它们的实验结果。在 YCSB 负载和 TPC-C 负载中，当只使用一块 SSD 来存储检查点文件和日志文件时，所有事务日志技术的恢复时间都一样。随着 SSD 数量的增加，由于 PARA+OPT 能够并行地从多个 SSD 中读取检查点和日志文件，因此它的性能呈线性增加。因此 PARA+OPT 的日志恢复具有较好的可扩展性。

4.6 本章小结

本项研究发现有限的磁盘 IO 带宽已经成为多核可扩展集中式事务日志技术最主要的性能瓶颈。为了解决这个问题，本项研究提出了一个面向可扩展存储设备的并行事务日志技术。该技术采用多个日志缓冲区和多块存储设备来分担了高通量事务日志的压力。事务根据负载感知的日志分区策略将日志记录写入对应的日志缓冲区和磁盘。这种日志分区策略避免了负载倾斜和跨分区事务对系统性能的影响。为了保证数据库系统的正确性和可恢复性，本项研究实现了偏序的日志号和持久化组提交协议。此外，本项研究实现了一个并行日志恢复方法来加速系统的恢复过程。并行事务日志技术和并行恢复方法集成到了一个实现乐观并发多版本协议的内存事务处理原型系统 Plover 中。最后，本项研究在原型系统上用 YCSB 和 TPC-C 基准测试对事务日志技术和恢复方法进行了评估。实验结果显示，并行事务日志技术和恢复方法比传统集中式技术具有更好的吞吐量和可扩展性。

第五章　支持系统可恢复性的偏序事务日志技术

5.1　引言

前两项研究解决了传统集中式事务日志技术的缓冲区竞争、固定组提交和有限磁盘 IO 带宽瓶颈，并取得了较大的系统性能提升和可扩展性。然而，传统事务日志技术的顺序性约束问题仍然存在。为了使得在发生事务故障或者系统故障后，数据库系统能够恢复到一个正确的状态，现有的事务日志技术严格地保证了事务的执行顺序，即事务日志记录的日志序列号、日志的持久化以及事务的提交全都遵循相同的全序关系。这种顺序性约束使得事务日志的所有操作必须以串行的方式执行，从而极大地影响了事务日志技术的并行性和可扩展性。

事实上，本项研究发现事务日志技术完全不需要遵循如此严苛的约束条件。尤其是对没有任何依赖的事务来说，它们的任何乱序操作不会对数据库系统的正确性产生任何的负面影响。为此，本项研究首先全面地分析了在保证数据库系统正确性和可恢复性的基础上事务日志技术需要遵循的必要约束条件，然后给出了可恢复的事务日志定义并验证了它的正确性。事务日志技术的可恢复性规定事务的提交顺序需要遵循事务之间的先写后读（RAW）依赖，并且事务日志记录的日志号需要遵循事务之间的写后写（WAW）依赖。

基于事务日志的可恢复性定义，本项研究提出了一种支持系统可恢复性的偏序事务日志技术 Poplar。该技术允许（1）日志记录能够并行地写入多个日志缓冲区和多块磁盘；（2）只有具有先写后读（RAW）依赖的事务才按事务执行顺序提交；（3）事务日志记录的日志号只追踪事务之间的写后写（WAW）依赖。为了避免产生额外的开销，本项研究使用了一种可扩展的偏序事务日志号（SSN）来追踪事务之间的 WAW 和 RAW 依赖并根据 SSN 实现了一种快速的事务提交协议。最后，本项研究在开源数据库原型系统 DBx1000 [22, 98] 中实现了支持可恢复性的偏序事务日志 Poplar，然后对比了 Poplar 和几种当前流行的事务日志技术。实验结果

表明，在 YCSB 和 TPC-C 基准测试中，Poplar 的性能随着磁盘数量的增加呈线性扩展并且 Poplar 同时拥有最高的吞吐量和最低的事务提交时延。

综上所述，本项研究主要包括以下贡献：

1. 本项研究明确了在保证系统可恢复性的基础上事务日志技术需要具备的必要约束条件，然后根据当前事务日志技术遵循的不同约束定义了三种事务日志级别：顺序性、严格性和可恢复性，并证明了可恢复事务日志技术的正确性。

2. 基于事务日志的可恢复性定义，本项研究提出了一种支持系统可恢复性的偏序事务日志技术 Poplar。该技术消除了传统事务日志存在的所有性能瓶颈——集中式日志缓冲区竞争、固定组提交、有限 IO 带宽和顺序性约束问题。

3. 本项研究基于开源数据库原型系统 DBx1000 实现了支持可恢复性的偏序事务日志技术 Poplar 并对其进行了性能评估。实验结果表明，当部署两块固态硬盘时，Poplar 的事务吞吐量是其他事务日志技术的 2 倍——280 倍，事务提交时延比其他日志技术减少了 5 倍。

5.2 事务日志级别

本节主要介绍了事务日志技术在保证系统可恢复性和正确性的基础上需要保证的约束条件。本章首先给出了三种事务日志级别的定义，然后证明了事务日志的可恢复性的正确性，最后按日志级别对当前流行的事务日志技术进行了归类。

5.2.1 定义

为了使系统能够在崩溃（系统故障或者掉电）之后恢复到一个正确的状态，数据库系统的恢复管理器需要满足以下两个条件：（1）能够确定在系统崩溃之前确定哪些事务处于已提交状态，哪些事务处于未提交状态；（2）能够按一个正确的顺序来重做（REDO）已提交事务的日志记录以及撤销（UNDO）未提交事务的日志记录。对于第一个条件来说，当且仅当某一个事务的日志记录已持久化并且它所

依赖事务是已提交的，那么这个事务才能处于已提交状态。而其余的事务处于未提交状态。换句话说就是，在事务执行过程中，一个事务需要在它依赖事务提交之后才能提交。对于第二个条件来说，来自冲突事务的日志记录必须遵循它们的执行顺序，即日志记录的顺序需要保证事务之间的依赖关系。为了保证系统恢复的正确性，事务提交和事务日志记录顺序需要满足一定的约束条件。根据它们遵循的不同约束，本项研究为事务日志定义了三种级别：顺序性、严格性和可恢复性。为了便于阐述，此处用C_i表示事务T_i的提交操作，L_i表示事务T_i日志记录的日志号，$C_i \prec C_j$表示C_i发生在C_j之前。三种事务日志级别的定义如下：

定义 5.2.1 (可恢复性). 如果事务日志是可恢复的，那么它需要满足以下条件：对于任意两个事务 T_i，T_j（$i \neq j$），如果 T_j 先修改了数据项 x，然后 T_i 读到了 T_j 对数据项 x 的修改，那么它们的提交顺序需要满足 $C_j \prec C_i$。此外，如果 T_j 先修改数据项 x，T_i 后修改数据项 x，那么它们的日志记录顺序需要满足 $L_j < L_i$。

定义 5.2.2 (严格性). 如果事务日志是严格的，那么它需要满足以下条件：对于任意两个事务 T_i，T_j（$i \neq j$），如果它们之间存在写写冲突或者是读写冲突并且 T_j 先于 T_i 执行，即 T_j 先写或者先读数据项 x，T_i 后读或者后写数据项 x，那么它们的事务提交顺序满足 $C_j \prec C_i$ 并且它们日志记录顺序满足 $L_j < L_i$。

定义 5.2.3 (顺序性). 如果事务日志是顺序的，那么它不仅需要满足严格性还需要满足以下条件：对于任意两个不存在任何依赖关系的事务 T_i, T_j（$i \neq j$），它们的事务提交顺序和日志记录顺序要么满足 $C_j \prec C_i$，$L_j < L_i$，要么满足 $C_i \prec C_j$，$L_i < L_j$。

总的来说，事务日志的可恢复性要求事务的提交顺序遵循事务之间的先写后读（RAW）依赖并且事务日志记录的日志号遵循事务之间的写后写（WAW）依赖。事务日志的严格性要求事务的提交顺序和事务日志记录的日志号满足事务之间的先写后读（RAW）、写后写（WAW）以及写后读（WAR）依赖。而事务日志的顺序性要求事务的提交顺序以及事务日志记录的日志号必须是全序的。很明显，当事务提交顺序和日志号的约束越多时，事务日志技术的设计与实现会越简单，但这些限制也降低了事务日志执行的并行性和可扩展性。

值得注意的是，在事务日志技术中讨论事务提交的顺序是有意义的。为了追求更高的性能，数据库系统通常采用提前锁释放（Early Lock Release）[35] 来减少事务冲突。提前锁释放（ELR） [66, 67] 允许事务在日志记录未被持久化之前就释放数据项上的写锁。这种事务被称为预提交事务。提前锁释放会导致其他事务读到预提交事务的修改结果。如果预提交事务还未提交，但是后续读到它修改的事务却提交了，那么此时一旦发生系统崩溃，数据库系统将恢复到一个不正确的状态。因此数据库系统的恢复管理器必须保证预提交事务先于其他事务提交，即事务日志的事务提交顺序需要遵循事务的先写后读依赖。

5.2.2 举例论证

本节主要论证了可恢复的事务日志如何保证数据库系统的正确性。图 5.1给出了四个事务在事务日志执行过程中可能的执行方案。事务之间的依赖关系如图中的箭头（$\longrightarrow$）所示。图中所有的事务都属于预提交事务，它们的日志记录都未写入磁盘。图中的正确符号（✓）表示遵循该执行方案的事务日志能够在系统故障之后使系统恢复到一个正确的状态，而图中错误符号（×）表示了相反的意思。

事务执行调度	事务日志执行方案
$W_1(x)$；$R_2(x)$；$W_2(y)$ $T_1 \xrightarrow{RAW} T_2$	方案（a）：$C_1 < C_2$；$L_1 < L_2$ √ 方案（b）：$C_1 < C_2$；$L_2 < L_1$ √ 方案（c）：$C_2 < C_1$；$L_2 < L_1$ ×
$R_2(x)$；$W_2(y)$；$W_3(y)$ $T_2 \xrightarrow{WAW} T_3$	方案（d）：$C_2 < C_3$；$L_2 < L_3$ √ 方案（e）：$C_2 < C_3$；$L_3 < L_2$ × 方案（f）：$C_3 < C_2$；$L_2 < L_3$ √
$R_2(x)$；$W_2(y)$；$W_4(x)$ $T_2 \xrightarrow{WAR} T_4$	方案（g）：$C_2 < C_4$；$L_2 < L_4$ √ 方案（h）：$C_4 < C_2$；$L_4 < L_2$ √

图 5.1: 事务日志执行方案

接下来本节将分别从先写后读（RAW）依赖事务、写后写（WAW）依赖事务和先读后写（WAR）依赖事务三个方面来证明事务日志可恢复性的正确性。

先写后读依赖事务。图中的事务 T_1 和事务 T_2 之间存在 RAW 依赖，即 T_2 读取了 T_1 在数据项 x 上的修改值。一方面，如果两个事务的提交顺序不遵循 RAW

依赖，那么 T_2 可以先于 T_1 提交（$C_2 \prec C_1$）。它们的执行情况如图中的方案（c）。如果在 T_2 提交之后并且 T_1 提交之前系统发生了崩溃，那么数据库状态将无法恢复到一个正确的状态。这是因为在恢复过程中只有 T_2 的执行结果能够被恢复（T_2 的日志记录被持久化了），而 T_1 的执行结果不能被恢复（T_1 的日志记录没有被持久化），这将导致 T_2 读到一个不存在的数据项 x 值。因此，T_1 和 T_2 的提交顺序必须遵循 RAW 依赖，即 $C_1 \prec C_2$。

另一方面，如果只是 T_1 和 T_2 的日志记录顺序没有保证 RAW 依赖，那么崩溃的数据库系统仍能恢复到一个正确的状态。如图中的方案（b），T_1 先于 T_2 提交（$C_1 \prec C_2$），但是 T_2 的日志号小于 T_1（$L_2 < L_1$）。如果在 T_1 提交之后 T_2 未提交之前系统崩溃了，那么在恢复时数据库系统只重构 T_1（T_1 的日志记录被持久化而 T_2 的日志没有）。此时的数据库状态是正确的。当数据库系统在两个事务都提交之后崩溃了，由于 T_2 的日志号较小因此它可能先于 T_1 被恢复。即便如此，数据库状态仍然是正确的。这主要是因为 T_2 在数据项 x 上只有读操作，不管它先执行还是后执行都不会产生任何的负面影响。因此，对于存在 RAW 依赖的 T_1 和 T_2 来说，它们的日志记录顺序不需要遵循 RAW 依赖。

写后写依赖的事务。图中的事务 T_2 和事务 T_3 存在 WAW 依赖，即 T_2 先修改了数据项 y，T_3 后修改了数据项 y。一方面，如果两个事务的提交顺序不遵循 WAW 依赖，那么 T_3 可以先于 T_2 提交（$C_3 \prec C_2$）。它们的执行情况如图中的方案（f）。如果在 T_3 提交之后并且 T_2 提交之前系统发生了故障，那么数据库系统仍能恢复到一个正确的状态。此时的数据库状态仿佛 T_2 从未发生过一样。因此，T_2 和 T_3 的提交顺序不需要遵循 WAW 依赖。

另一方面，如果 T_2 和 T_3 的日志记录顺序没有保证 WAW 依赖，那么发生系统故障之后数据库系统不能恢复到一个正确的状态。如图中的方案（e），T_3 的日志号小于 T_2（$L_3 < L_2$）。如果数据库系统在两个事务都提交之后发生了故障，那么 T_2 和 T_3 将按照它们日志号顺序进行恢复，即 L_3 先于 L_2 恢复。由于后恢复的 T_2 会覆盖掉 T_3 在数据项 y 的值，从而导致恢复之后的状态与系统崩溃之前的状态不

一致。因此，对于存在 WAW 依赖的 T_2 和 T_3 来说，它们的日志记录顺序必须遵循 WAW 依赖，即 $L_2 < L_3$。

先读后写依赖事务。图中的事务 T_2 和事务 T_4 存在 WAR 依赖，即 T_2 读了数据项 x 之后 T_3 修改了数据项 x。一方面，如果两个事务的提交顺序不遵循 WAR 依赖，那么 T_4 可以先于 T_2 提交（$C_4 \prec C_2$）。它们的执行情况如图中的方案（h）。如果在 T_4 提交之后并且 T_2 提交之前系统发生了故障，那么只有 T_4 能够被恢复而 T_2 不会。恢复之后的数据库状态相当于系统从未执行过 T_2 一样，因此恢复后的状态与崩溃之前的状态是一致的。因此，T_2 和 T_4 的提交顺序不需要遵循 WAR 依赖。

另一方面，如果 T_2 和 T_4 的日志记录顺序没有保证 WAR 依赖，那么发生系统故障之后数据库系统仍能恢复到一个正确的状态。如图中的方案（h），T_4 的日志号小于 T_2（$L_4 < L_2$）。如果系统宕机发生在两个事务提交之后，那么 T_2 和 T_4 需要按照它们的日志号进行恢复，因此 T_4 会先于 T_2 被恢复。由于 T_2 在数据项 x 只是读取操作，后恢复 T_2 将不会改变数据项 x 的值。因此，对于存在 WAR 依赖的 T_2 和 T_4 来说，它们的日志记录顺序不需要遵循 WAR 依赖。

总的来说，为了保证系统在崩溃之后能够恢复到一个正确的状态，恢复管理器只需要保证事务的提交顺序遵循事务的 RAW 依赖，同时事务日志记录的日志号追踪事务的 WAW 依赖。因此，事务日志的可恢复性级别足以保证系统的正确性。

5.2.3 方法对比

事务日志的可恢复性定义彻底打破了集中式事务日志的顺序约束，这为在单节点数据库系统中实现并行的事务日志技术提供了理论依据。基于可恢复性事务日志级别，本项研究提出了一种支持系统可恢复性的并行事务日志技术（Poplar）。在此之前，已有许多研究提出了许多高性能的事务日志技术 [15, 18, 20, 35–38]。本节根据事务日志级别归纳并总结了它们的优缺点。表 5.1给出了比较结果。接下来，本节首先详细地分析了当前流行的并行事务日志技术存在的缺点，然后介绍了 Poplar 的设计思想和与其他技术的不同之处。

表 5.1: 与现有事务日志技术的比较结果

日志技术	日志号顺序	日志填充方式	日志持久化方式	事务提交顺序	恢复方式	事务日志级别
ARIES [6]	全序	串行	串行	全序	串行	顺序性
Aether [35] ELEDA [36]	全序	并行	串行	全序	串行	顺序性
H-Store [18]	全序	并行	串行	全序	串行	顺序性
Silo [37]	粗粒度全序	并行	并行	粗粒度全序	并行	粗粒度顺序性
NV-Log [38]	全序	无	并行	全序	并行	顺序性
NVM-D [39]	RAW WAW WAR	无	并行	RAW WAW WAR	并行	严格性
Poplar	RAW WAW	并行	并行	RAW	并行	可恢复性

Silo [37, 51] 是一种并行的事务日志技术，它属于 5.2.1节中定义的事务日志顺序性。Silo 使用多个日志缓冲区来并行地存储事务的日志记录，同时允许日志记录被并行地写入多块存储设备。为了避免内存数据库系统中所有的集中式瓶颈点，它采用 epoch 来表示事务的事务号以及追踪事务之间的依赖。Epoch 具有全局递增性并且是周期性地更新的。这种粗粒度的方式提高了内存数据库系统的事务处理性能和可扩展性。但是，由于处于同一个 epoch 内的事务无法区分它们的依赖关系，因此 Silo 必须保证一个 epoch 内的所有事务要么都提交要么都不提交。这种提交方式被称为 s 基于 epoch 的提交协议。显而易见，epoch 粒度越大，数据库系统的事务吞吐量会越大，但是事务的提交时延也会越高。

NV-Log [38] 是一种基于非易失内存设备（NVM）设计的非集中式事务日志技术。它在 NVM 上为每一个工作线程分配了一个私有的日志缓冲区，然后允许日志线程并行地将事务日志写入非易失内存设备中。虽然 NV-Log 支持了并行的日志持久化，但是它仍然采用了传统集中式的日志序列号（LSN）来追踪事务日志的顺序以及事务的提交顺序。此外，当系统崩溃之后，数据库系统需要按照日志序列号（LSN）的顺序恢复事务日志记录从而使得系统恢复到一个正确的状态。因此，NV-Log 属于一个满足顺序性的事务日志。

NVM-D [39] 是一种基于非易失内存设备设计的并行事务日志技术。它在 NVM 上分配了与工作线程数量相等的日志缓冲区，然后允许工作线程按照不同的日志分区策略（按数据分区方式或者按事务分区方式）将事务日志记录并行地存入对应的日志缓冲区。此外，它采用了一个新的事务日志号（GSN）来追踪事务之间的先写后读、写后写、先读后写依赖。事务的日志记录顺序、事务的提交顺序以及事务的恢复顺序全都遵循 GSN 的顺序，因此它属于满足严格性的事务日志。

虽然基于 NVM 的事务日志技术 NV-Log 和 NVM-D 能够提供系统的并行性以及事务日志的持久化时间，但是在这些技术中，工作线程频繁地使用 mfence 指令执行持久化操作。这种方式严重地降低了系统的事务处理性能并且不适用了基于普通存储设备（HDD，SSD）的数据库系统。

设计思想。为了解决集中式事务日志技术存在的所有性能瓶颈，本项研究设计了一种通用的事务日志技术。它具有可扩展性、并行性和可恢复性。这种技术在本文被称为 Poplar。它的主要设计思想如下：它使用了多个日志缓冲区和多块存储设备来并行地存储事务日志记录，然后使用了一个偏序事务日志号（SSN）来追踪事务之间的写后写（WAW）依赖，最后允许事务的提交顺序仅遵循事务的先写后读（RAW）依赖。为了避免追踪事务提交依赖产生额外的开销，本项研究直接使用偏序事务日志号来记录事务的 RAW 依赖，然后事务按照 SSN 的顺序进行提交。SSN 采用了一种分布式的方式进行分配，因此它具有较好的多核可扩展性。

方法比较。与 Silo 相比，Poplar 使用了一个细粒度的 SSN 来保证事务日志的可恢复性。它不仅具有较好的事务吞吐量而且拥有比 Silo 更小的事务提交时延。与 NVM-D 和 NV-Log 相比，Poplar 使用了单独的日志线程来负责事务日志记录的持久化。日志线程采用了高效的组提交协议 [34, 62] 避免了频繁的磁盘 IO 操作。这种方式使得 Poplar 不论在新型硬件 NVM 还是普通的存储设备上都能展现出很好的事务处理性能。此外，与 NVM-D 和 NV-Log 的日志号相比，SSN 不用保证事务之间的 WAR 依赖。因此，在计算 SSN 时 Poplar 不用修改只读数据项上的特殊字段（如数据项上存储的 SSN），从而能够在混合应用负载中获得更好的性能。

5.3 可恢复的事务日志

本节中介绍了支持系统可恢复性的事务日志技术的详细设计。本节首先概括了事务日志技术的整体框架和它遇到的技术难题，然后分别详细地描述了事务日志技术的设计与实现。

5.3.1 整体架构

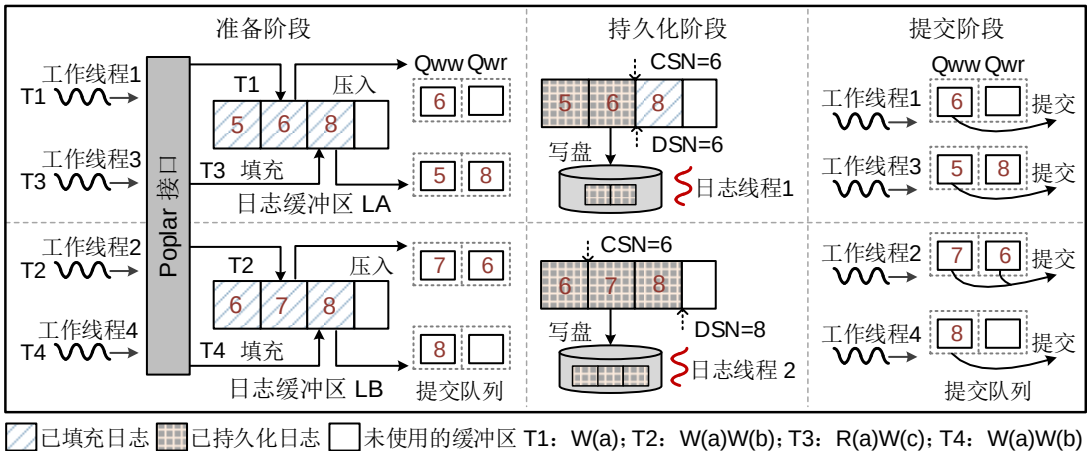


图 5.2: 可恢复性事务日志的整体框架

可恢复的事务日志技术 Poplar 主要使用了多个日志缓冲区和多块磁盘来存储事务日志记录。它主要包含了两种类型的处理线程：工作线程和日志线程。日志线程和日志缓冲区存在一个一对一的映射关系，而工作线程和日志缓冲区之间存在一个多对一的映射关系。每个工作线程生成事务日志记录之后将它填充到对应的日志缓冲区中，然后日志线程将该缓冲区中的记录写入对应的存储设备。图 5.2给出了可恢复事务日志技术的整体架构。图中包含了两个日志缓冲区 LA 和 LB、两个日志线程和四个工作线程。其中，工作线程 1、3 对应日志缓冲区 LA，工作线程 2、4 映射到日志缓冲区 LB。每个工作线程有两个私有提交队列（Qww 和 Qwr）。Qww 称为写写队列，它主要用于存储存在 WAW 依赖的事务。Qwr 称为读写队列，它用于存储包含 RAW 依赖的事务。Poplar 的执行流程主要包含三个阶段：准备阶段、持久化阶段和提交阶段。

在准备阶段，每个工作线程首先为处理的事务生成日志记录，然后将日志记录写入对应的日志缓冲区。在填充日志记录之前，每个工作线程会为事务和该事务的日志记录分配一个偏序日志号（SSN）。这里假设每个事务只产生一条日志记录。SSN 保证了事务之间的偏序顺序并跟踪了事务之间的 RAW 和 WAW 依赖。当日志记录被填写到日志缓冲区之后，该事务进入提交队列并等待提交。如果它是只读或读写事务，那么工作线程将它推送到 Qwr 队列中。如果事务只包含写操作，那么工作线程将它推入 Qww 队列。图 5.2中的准备阶段描述了四个事务的执行过程。事务 T_1、T_2、T_3、T_4 分别由工作线程 1、2、3、4 负责执行。其中，T_1 首先分配日志号 SSN 等于 6，然后工作线程 1 将它的日志记录写入日志缓冲区 LA 中。填充完成之后，由于 T_1 只包含了一个写操作，因此工作线程 1 将它推送到自己的写写队列（Qww）。同样地，T_2 和 T_4 的日志号 SSN 分别为 7、8，它们的日志记录被工作线程写入日志缓冲区 LB 中，最后压入对应工作线程的 Qww 中。而对于事务 T_3 来说，它的日志号 SSN 为 8。由于 T_3 包含了读写操作，因此在日志记录填充完成之后，它被推送到工作线程 3 的读写队列 Qwr。

在持久化阶段，每个日志线程首先独立地将各自对应日志缓冲区中的日志记录写入绑定的存储设备中，然后更新自己本地已持久化的 SSN。这里用 DSN 来表示已持久化的日志号。也就是说，每个日志线程或日志缓冲区拥有一个 DSN，它表示每个日志线程最新被持久化日志记录的日志号。如图 5.2所示，日志缓冲区 LA 的 DSN 为 6，日志缓冲区 LB 的 DSN 为 8。由于在准备阶段中，Poplar 采用了 SSN 和并行日志填充方法，因此日志缓冲区不可避免地会出现部分填充的日志记录，即日志空洞。如果事务日志技术将存在日志空洞的日志记录写入磁盘，那么在系统崩溃之后数据库系统将无法恢复到一个正确的状态。为了避免出现这个问题，事务日志技术实现了基于日志段的日志空洞追踪方法和日志持久化。只有当一个日志块里的所有日志记录都已填充完成，它们才能被写入存储设备。

在提交阶段，Poplar 需要确定哪些事务可以被提交。根据 5.2.1节中定义的可恢复性，可提交的事务需要满足两个条件：（1）它的日志记录已经被持久化了；（2）

与它存在 RAW 依赖的事务已经提交。因此存在 RAW 依赖关系的事务只有满足了条件（1）和条件（2）之后才能被提交，而其他事务只需要满足条件（1）就能被提交。检查事务是否满足第一个条件可以简单地对比事务的日志号 SSN 与对应日志缓冲区的持久化日志标识 DSN。然而，由于存在 RAW 依赖事务的日志记录可能分布在不同的日志缓冲区中，因此检查事务是否满足第二个条件就没有那么简单了。为此，事务日志技术使用了一个全局可提交的日志号（用 CSN 来表示）。CSN 是系统判断存在 RAW 依赖事务是否能被提交的依据。基于 CSN，工作线程可以直接比较可提交日志号与读写提交队列（Qwr）中事务日志号的大小，然后提交那些日志号小于 CSN 的事务。而处于写写提交队列（Qww）中的事务，它们需要对比日志号与对应日志缓冲区的持久化日志号 DSN，然后提交那些日志号小于 DSN 的事务。在图 5.2中，CSN$=6$、LA 的 DSN$=6$、LB 的 DSN$=8$。通过比较 CSN 与所有 Qwr 中的事务日志号，T_3 不能被提交。而通过对比 DSN 和所有 Qww 中的事务日志号，T_1、T_2 和 T_4 都可以被提交。

5.3.2 事务日志号

根据可恢复性的定义，并行事务日志技术只需要关注事务的 WAW 和 RAW 依赖，即事务日志号需要保证事务的写后写依赖并且事务的提交顺序需要满足事务的先写后读依赖。为此，可恢复的事务日志技术提出了一种可扩展的事务日志号（SSN）。SSN 保证了事务之间的偏序关系，即追踪了事务之间的先写后读依赖和写后写依赖。不同于之前的日志分配方法，可恢复的日志技术没有采用一种集中式的分配方式来计算日志号 [15, 35–37]，而是采用了一种分布式的计算方式。事务日志号根据事务访问的数据项和日志记录存储的日志缓冲区的实际情况来计算。分布式计算方式避免了集中式的竞争，从而使得事务日志具有更好的多核可扩展性。

计算 SSN。在支持系统可恢复性的事务日志技术中，日志号 SSN 采用了一种协同存储的方式，即每个日志缓冲区和每个数据项都维护了一个本地的 SSN。日志缓冲区中的日志号等于最新写入的日志记录的 SSN，而数据项上记录的是最新

算法 6: 可扩展事务日志号 SSN 的计算

```
   输入: 事务T，事务读集合RS, 事务写集合WS
   输入: 日志缓冲区 L，事务日志记录大小 len
   输入: 数据项 e
   输出: 日志号计算成功或者失败
1  base = 0
   /* 获取事务访问数据项上最大的日志号                                  */
2  for e in RS∪WS do
3  |  base = max(e.ssn, base);
4  end
5  if WS 不为空 then
   |  /* 计算事务日志号                                               */
6  |  while CAS( L.latch, false, true) do
7  |  |  T.ssn = max(base, L.ssn) + 1;
8  |  |  L.ssn = T.ssn;
9  |  |  FETCH_ADD( L.offset, len);
10 |  |  COMPILER_BARRIER();
11 |  |  L.latch = false;
12 |  end
   |  /* 更新事务写集合中数据项的日志号                                */
13 |  for e in WS do
14 |  |  e.ssn = T.ssn
15 |  end
16 else
   |  /* 计算只写事务的日志号                                         */
17 |  T.ssn = base
18 end
```

修改它的事务的 SSN。算法 6给出了计算事务日志号的伪代码。算法的第 1 行到第 4 行获取了事务所有读写数据项上最大的日志号。算法的第 6 行到第 12 行首先获取数据项和日志缓冲区中最大的日志号，然后进行加 1 操作得到事务 T 的日志号，最后将日志缓冲区的本地日志号修改为生成的 SSN。为了避免并行事务在日志缓冲区上的写写冲突，该算法实现了一个基于 latch 的 CAS 指令。算法中的第 13 行到第 15 行更新了事务写集合中数据项的本地日志号。最后算法的第 17 行考虑了只读事务的情况。对于只读事务来说，它的事务号直接设置为读集合中最大的数据项日志号。除了分配事务的 SSN 之外，算法的第 9 行还计算了每个事务日志记录在日志缓冲区上的存储位置。

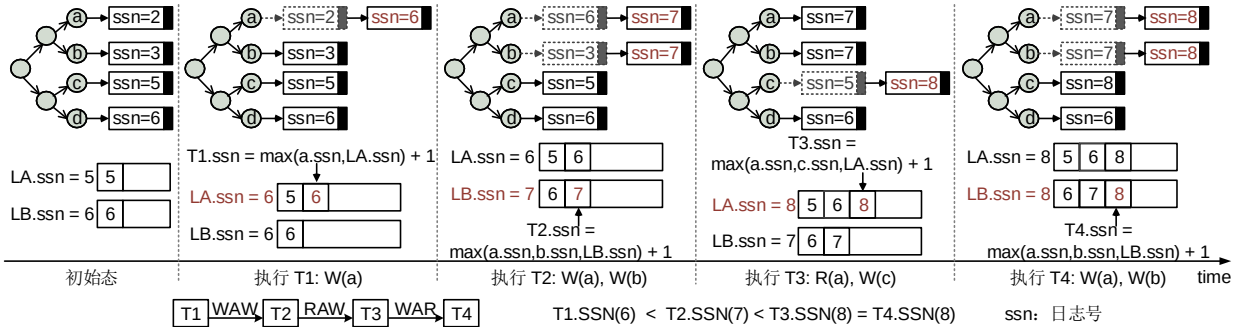


图 5.3: 可扩展事务日志号的计算过程

图 5.3给出了事务 T_1、T_2、T_3 和 T_4 计算日志号的详细过程。假设 T_1 只修改数据项 a 并且它的日志记录将写入日志缓冲区 LA 中。T_1 首先获取数据项 a 存储的本地日志号（a.SSN= 2）和日志缓冲区 LA 的日志号（LA.SSN= 5），然后计算事务的 SSN 为 $max(a.ssn, LA.ssn) + 1 = 6$，最后再更新日志缓冲区 LA 和数据项 a 的 SSN。假设 T_2 修改了数据项 a 和 b 并且它的日志记录将写入日志缓冲区 LB 中。由于事务 T_1 和 T_2 之间存在 WAW 依赖，因此 T_2 的日志号必须大于 T_1 的日志号。在计算事务 SSN 时，T_2 首先获取了数据项 a、b 和缓冲区 LB 中最大的 SSN，然后执行加 1 操作计算出 SSN= 7，最后将数据项 a 和 b 的本地 SSN 修改为 7。这种计算方式保证了事务 T_1 和 T_2 之间的 WAW 依赖。采用同样的方式，T_3 的 SSN 计算为 8。它的日志号大于 T_2 的日志号，因此事务日志号遵循了 T_2 和 T_3 之间的 RAW 依赖。不同的是，对于 T_3 来说，由于它在数据项 a 上执行读取操作，在数据项 c 上执行更新操作，因此只有数据项 c 的本地 SSN 需要修改为事务日志号 8，而数据项 a 的本地日志号不需要修改。最后，T_4 采用相同的日志号分配方式计算出 SSN= 8。虽然 T_3 和 T_4 之间存在读写冲突，但是由于它们之间是 WAR 依赖，因此 T_3 和 T_4 的事务日志号可以相同。

5.3.3 事务提交协议

Poplar 提出了一种快速的事务提交协议，它不需要事务按全序进行提交。在提交阶段中，写写队列 Qww 中的事务通过比较它们的事务日志号 SSN 和对应日志缓冲区的 DSN 来判断是否能被提交，而读写队列 Qwr 中的事务通过比较事务日志

号与 CSN 的大小来判断是否可以被提交。因此，每个日志缓冲区的 DSN 和全局的 CSN 在事务提交过程中占据了非常重要的地位。接下来，本节将介绍并行事务日志技术如何计算 DSN 和 CSN 以及如何保证系统的正确提交。

计算 DSN。为了解决由基于原子指令的 SSN 分配和并行的日志记录填充引起的日志空洞问题，本项研究提出了一种基于日志段的空洞检测方法。该方法将日志缓冲区逻辑地分成大小不同的日志段，其中每一个日志段看做一个空洞检测单元和持久化单元。只有当一个日志段里的所有日志记录都完成填充以后，日志线程才能将日志段里的记录写入存储设备。等日志记录持久化之后，日志线程将 DSN 更新为当前已持久化日志记录中最大的 SSN。

日志段数组

seg	ssn	allocated	filled	start	state
0	0	0	0	0	Open
1	8	64k	64k	0x2000	Closed
2	15	32k	12k	0x12000	Closed
3	0	16k	16k	0x1A000	Open
4	0	0	0	0	Open
...	...	...	...	...	...

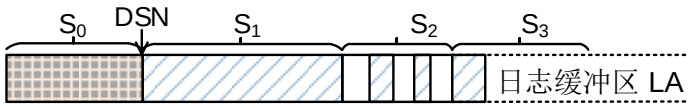


图 5.4: 日志段数组结构.

事务日志技术为每个日志缓冲区维护了一个包含多个日志段的数组结构（用S来表示）。图 5.4给出了它的数据结构。每个日志段包含五个成员变量（ssn，$allocated$，$filled$，$start$，$state$），其中ssn 记录的是日志段中日志记录最大的 SSN，$allocated$ 和$filled$ 分别表示日志段中已分配的日志记录长度和已填充完成的日志记录长度，$start$ 指向日志段在日志缓冲区中的起始位置，$state$ 表示日志段的状态（如果状态值为关闭，则表示当前日志段已创建完成）。

每个日志段可以由以下两种方式来产生：（1）如果当日志段中已分配的日志记录长度超过了设定的 IO 单元，那么工作线程就将当前日志段的状态由开启状态改为关闭状态；（2）日志线程每隔一段时间t 将当前正在使用的日志段状态修改为

关闭。IO 单元和时间t 可以根据数据库系统的硬件配置和实时的负载进行设置。这里将创建日志段的工作线程或者日志线程统称为分段线程。

基于日志段的数据结构，事务日志技术可以根据它的状态值来判断日志段是否创建成功并且根据已分配长度和已填充长度来判断已创建成功的日志段中是否存在日志空洞。图 5.4中，S_1 和 S_2 是已经创建成功的日志段，而 S_3 是正在创建的日志段。在已生成的日志段中，由于 S_1 的已分配日志记录长度等于它已填充的日志记录长度，因此 S_1 中不存在日志空洞。相反，S_2 中存在日志空洞。

算法 7: 已持久化日志号 DSN 和可提交日志号 CSN 的计算

```
输入: 日志段数组 S, 工作线程 Lg
输入: 日志缓冲区 L, 日志缓冲区集合 LBS
输入: IO 单元 IO, 分段时间 FT
输出: 计算成功或者失败
1  Procedure 创建日志段 ()
2      i = cur_generate_seg modulo S.size;
3      if (S[i].allocated ≥ IO || Lg.wait_time ≥ FT) && S[i].state != closed
         then
4          S[i].ssn = L.ssn;
5          S[i].state = closed;
6          S[i+1].start = L.offset;
7          FETCH_ADD(cur_generate_seg,1);
8      end
9  Procedure 计算 DSN()
10     i = cur_flush_seg modulo S.size;
11     while (S[i].allocated == S[i].filled) && (S[i].state == closed) do
12         flush(S[i].start, S[i].allocated);
13         L.dsn = S[i].ssn;
14         COMPILER_BARRIER();
15         FETCH_ADD(cur_flush_seg,1);
16         S[i].reset();
17     end
18 Procedure 计算 CSN()
19     for l in LBS do
20         csn = min(csn, l.dsn)
21     end
```

算法 7给出了创建日志段和计算 DSN 的伪代码。算法的第 1 行到第 8 行描述了分段线程如何创建一个新的日志段。算法为每一个日志段数组维护了一个变

量$cur_generate_seg$，它指向当前正在创建的日志段。一旦创建日志段的条件被触发后，分段线程修改当前正在创建日志段的状态值（$state$）、最大 SSN 以及它在日志缓冲区中的起始位置（$start$）。修改成功之后，分组线程以原子递增的方式修改$cur_generate_seg$。算法的第 9 行到 17 行介绍了如何计算 DSN。同样地，该算法为每一个日志段数组维护了一个变量cur_flush_seg，它指向即将被持久化的日志段。算法的第 10 行通过cur_flush_seg 找到数组中对应的日志段，然后算法的第 11 行判断当前日志段是否创建成功并且是否不存在日志空洞。只有当日志段的状态为关闭（closed）并且它已分配的日志记录长度等于已填充的日志长度，该日志段中才不存在日志空洞。当日志段中不存在日志空洞时，算法的第 12 行到 16 行首先将日志记录写入存储设备，然后更新日志缓冲区的 DSN 为当前日志段的最大ssn，最后日志线程更新cur_flush_seg 值并且将日志段恢复到初始值。

计算 CSN。 CSN 等于所有日志缓冲区中 DSN 的最小值。算法 7中的第 18 行到第 21 行给出了计算 CSN 的伪代码。当事务完成日志持久化之后，日志线程读取所有日志缓冲区的 DSN 然后将 CSN 设置成它们中的最小值。对于存在 RAW 依赖的事务来说，如果它们的日志号 SSN 小于 CSN，那么这些事务可以提交。假设 T_i 和 T_j 之间存在 RAW 依赖，即 T_i 先写了数据项 x 之后 T_j 读了数据项 x，那么 T_j 必须在 T_i 提交之后才能提交。由于它们的日志号始终满足 $\text{SSN}_i < \text{SSN}_j$，因此它们与 CSN 的关系只存在以下三种情况：（1）$\text{SSN}_i < \text{SSN}_j \leq \text{CSN}$；（2）$\text{SSN}_i \leq \text{CSN} < \text{SSN}_j$；（3）$\text{CSN} < \text{SSN}_i < \text{SSN}_j$。在第一种情况中，$T_i$ 和 T_j 都能被提交。在第二种情况中，只有 T_i 能提交而 T_j 不能提交。在第三种情况下，T_i 和 T_j 都不能被提交。因此，直接比较存在 RAW 依赖事务的 SSN 和 CSN 足以保证事务的正确提交。

5.4 实验与分析

本节主要评估了支持系统可恢复性的事务日志技术 Poplar 的性能。本项研究首先在一个开源内存数据库原型系统 DBx1000 [22] 上实现了 Poplar，然后对比了 Poplar 和其他事务日志技术，并得出了以下结论：

1. 由于采用了去中心化的日志号分配方法和并行的执行方式，Poplar 比其他事务日志技术获得了更高的吞吐量。

2. Poplar 实现了一种细粒度快速的事务提交协议。由于它仅要求存在 RAW 依赖的事务按顺序提交，因此它拥有更小的平均事务提交时延。

3. Poplar 彻底打破了传统集中式事务日志的顺序约束性，从而使得事务日志具有很好的可扩展性和并行性。

5.4.1 实验配置

实验环境。本实验运行在一台服务器上。该服务器配置了一块 256GB 的 DRAM、两个 Intel Xeon v4 E5-2630@2.20GHZ 的 CPU 处理器（总共包含 20 个物理核）和 4 块 1.6TB 的 PCIe 接口的 SSD。实验中没有使用超线程，每个 SSD 提供了每秒 1.2GB 的顺序写性能以及 21.5us 的访问时延（每次顺序写 16KB 的文件）。

NVM 模拟。当前市场上可用的非易失性内存设备 [99] 大多采用将 DDR4 DIMM 集成到配置有 NVDIMM 的服务器上。NVDIMM 提供了近似于 DRAM 的访问性能。因此，本项研究在实验中使用 DRAM 来模拟 NVM 的性能并且使用 CFLUSH 指令来模拟 NVM 的访问时延，其中 NVM 的存储时延配置成 DRAM 的 2 倍。

实验对比项。本实验主要对比了以下四种不同的事务日志技术：

1. CENTR。它表示一种集中式事务日志技术。该技术使用了一个集中式的日志缓冲区和一块存储设备来存储日志记录。 CENTR 采用了 fetch-add 原子指令来获取事务日志记录的 LSN 并允许事务能够并行地填充日志。由于采用了集中式的日志号，因此所有事务需要串行地提交。

2. NVM-D。它表示一种基于非易失内存（NVM）设计的并行事务日志技术 [39]。该技术允许工作线程直接将日志记录写入 NVM 中。 NVM-D 实现了以分布式的计算方式来分配每个事务记录的日志号 GSN。GSN 追踪了事务之间的所有依赖（RAW、WAW 和 WAR）。所有的事务按照 GSN 的顺序进行提交。

3. SILO。它表示一种实现了粗粒度提交协议的并行事务日志技术 [37]。该技术使用多块普通的存储设备来并行地持久化日志记录。 SILO 采用了一个周期性更新的 epoch（每 50ms 更新一次）来保证事务的全序关系。事务的日志号和提交顺序都遵循 epoch 的顺序。

4. POPLAR。它表示本项研究实现的支持系统可恢复性的并行事务日志技术。该技术使用多个日志缓冲区和多块磁盘来存储事务的日志记录。 POPLAR 实现了一种细粒度的偏序日志号（SSN）和快速提交协议。SSN 追踪了事务之间的 RAW 依赖和 WAW 依赖。提交协议允许只有存在 RAW 依赖的事务才按 SSN 的顺序提交，其他事务可以并行地提交。

所有的对比项都在开源内存数据库原型系统 DBx1000 中实现了。为了保证实验的公平性，所有对比项都采用了基于 Silo 的乐观并发控制协议。默认情况下，除 CENTR 之外，所有的对比项都配置了两块 SSD。在 SILO 和 POPLAR 中，SSD 的数量等于系统中日志缓冲区的数量和日志线程的数量。每个日志缓冲区的大小为 30MB。日志线程都实现了组提交协议：每隔 5ms 或者每当日志缓冲区内积累的日志记录长度超过 15MB 时，日志线程执行一次持久化操作。在模拟 NVM 的实验中，日志缓冲区的大小为 1MB 并且组提交协议的条件改为：每隔 5ms 或者当日志缓冲区内的日志记录长度累积到缓冲区的 1/10 时，工作线程执行日志持久化操作。

5.4.2 工作负载

本实验使用 YCSB 基准测试和 TPC-C 基准测试来验证事务日志技术的有效性。

YCSB。 YCSB [88] 是由雅虎公司提供的一种测试标准。它主要用于测试大规模在线云服务系统的读写性能。YCSB 基准测试主要包含了一张测试表和五种测试负载。该测试表包含一个 64 位的整型列（主键）和十个 100 字节的字符串列。初始时，测试表中包含 1000 万条数据记录。本实验主要采用了两种测试负载：

1. **只写负载**。这种负载包含一个写事务并且每个事务只修改一条数据记录。

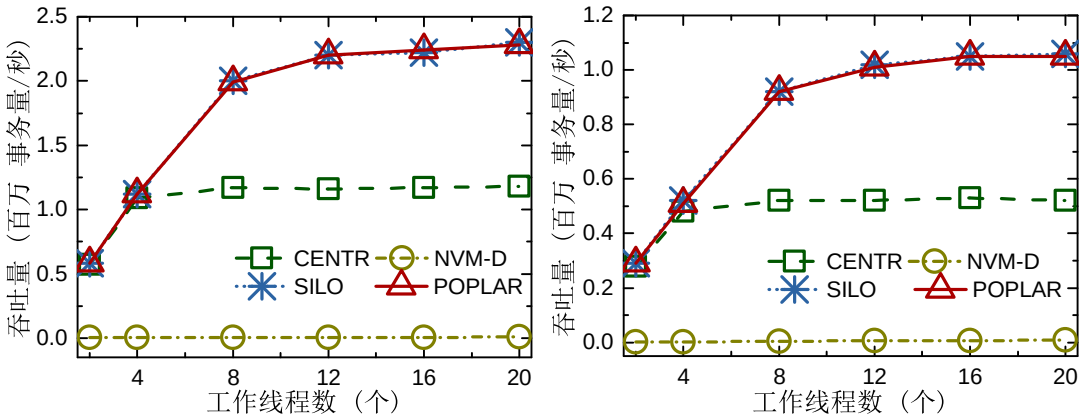
2. **混合负载**。该负载包含一个读写事务。每个读写事务由一个更新操作和一个范围查询构成。

默认情况下，所有的负载都服从均匀分布。这种均匀分布方式可以避免并发控制等模块对测试结果产生影响。在测试过程中，每组实验都进行了 5 次测试，然后取 5 次测试中的平均值作为实验结果。

TPC-C。 TPC-C [95] 是由事务处理性能委员会（TPC）提供的一种用于衡量在线事务处理系统性能的基准测试。它模拟了一个以仓库为中心的订单管理系统。TPC-C 基准测试包含了九张测试表和五个事务（三个读写事务和两个只读事务）。本实验主要使用了两个读写事务：50% 的 payment 和 50% 的 neworder。默认情况下，本实验总共配置了 20 个仓库并且每组实验进行了 5 次测试。

5.4.3 实验结果与分析

本实验首先对比了 CENTR、 NVM-D、 SILO 和 POPLAR 四种事务日志技术的事务吞吐量、磁盘带宽、事务提交时延、事务日志号分配方式以及可扩展性，然后验证了快速提交协议的有效性。



(a) 在 YCSB 中吞吐量　　(b) 在 TPC-C 中的吞吐量

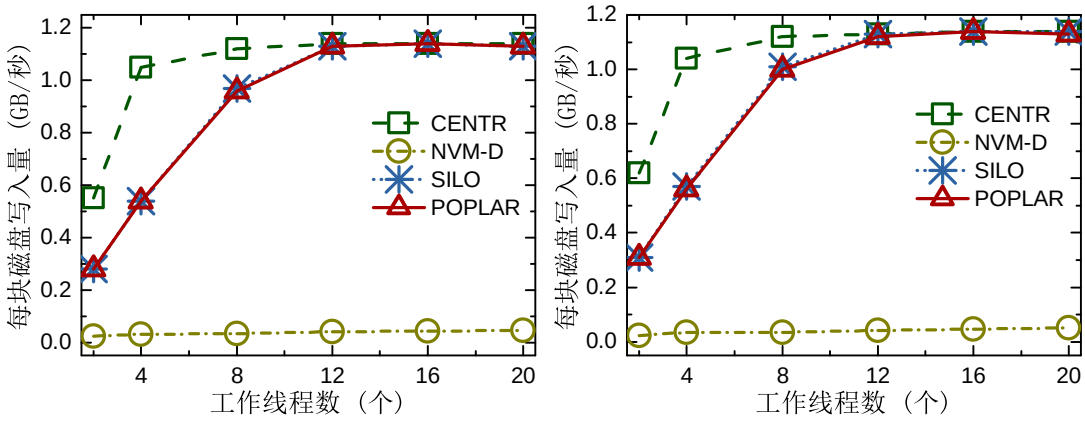
图 5.5: 不同事务日志技术的吞吐量

吞吐量。本组实验使用两块 SSD 来存储事务日志记录并通过改变工作线程的数量来观察每种事务日志技术的吞吐量变化情况。图 5.5显示了所有事务日志技术

在 YCSB 和 TPC-C 基准测试中的吞吐量实验结果。

在 YCSB 只写负载中，随着工作线程数量的增加，CENTR、SILO 和 POPLAR 的事务吞吐量先逐渐升高然后趋于稳定。事务吞吐量达到饱和值的原因是每秒产生的事务日志量达到了磁盘 IO 带宽的最大值。由于只使用了一块 SSD，因此 CENTR 最先遭遇 SSD 的 IO 带宽瓶颈从而获得了最低的吞吐量。由于使用了多块 SSD，POPLAR 和 SILO 展现出了更高的系统处理性能，它们的吞吐量是 CENTR 的 2 倍。对于 NVM-D 来说，随着工作线程数量的增加，它的吞吐量呈现出不断上升的趋势。但是，由于它的实验结果值太低，因此上升趋势在图中并不明显。比较 POPLAR 和 NVM-D 的性能，它们之间的吞吐量值相差了 280 倍。如此大的性能差距主要是因为 NVM-D 允许工作线程直接将日志记录写入 SSD 中。频繁地 IO 操作导致了大量的上下文切换和操作系统的调度开销。因此，基于新型非易失内存设备实现的事务日志技术并不适用部署在普通存储设备（SSD 和 HDD）上的数据库系统。

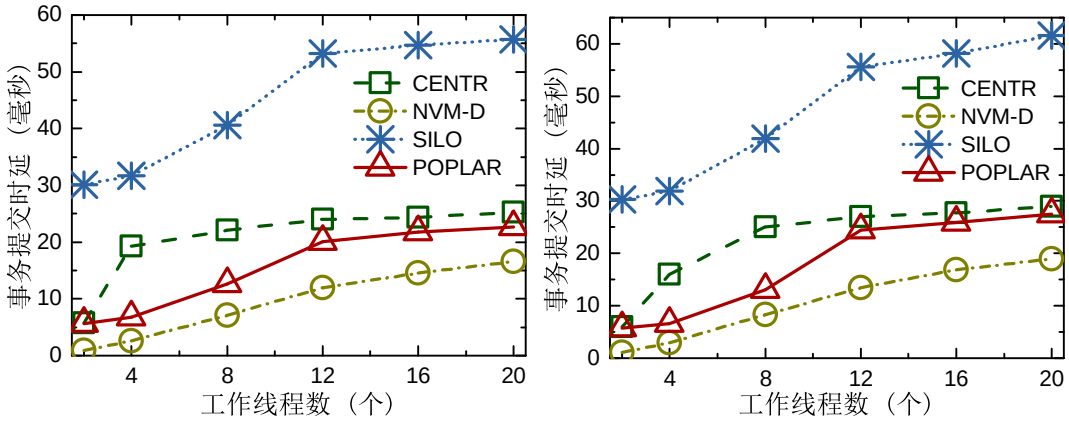
在 TPC-C 负载中，所有事务日志技术展现了与 YCSB 一样的性能趋势。CENTR、SILO 和 POPLAR 的吞吐量最终都受限于 SSD 的 IO 带宽。由于使用了两块 SSD，因此 POPLAR 和 SILO 的吞吐量是 CENTR 的两倍。如果增加 SSD 的个数， POPLAR 和 SILO 的事务吞吐量会继续升高。而 NVM-D 获得较低的吞吐量是因为频繁的持久化操作限制了系统的事务处理能力。



(a) 在 YCSB 负载中的结果　　(b) 在 TPC-C 中的结果

图 5.6: 不同事务日志技术的磁盘带宽情况

磁盘带宽。接下来，本组实验用 nmon 工具 [100] 监控了所有事务日志技术在测试中的磁盘 IO 带宽使用情况。图 5.6给出了每个 SSD 的带宽结果。在 YCSB 只写负载和 TPC-C 负载中，随着工作线程数量的增加， POPLAR、 SILO 和 CENTR 的单块 SSD 带宽先逐渐升高，然后达到一个稳定值。实验结果说明，事务日志技术的吞吐量最终受限于 SSD 的 IO 带宽。由于 NVM-D 的吞吐量特别低，因此它的每块 SSD 带宽使用率并不高。

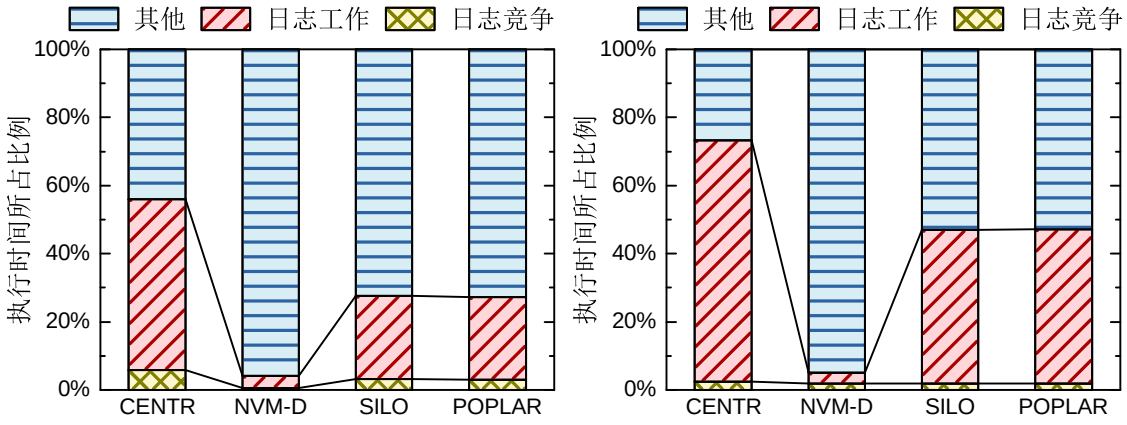


(a) 在 YCSB 负载中的时延　　(b) 在 TPC-C 负载中的时延

图 5.7: 不同事务日志技术的事务提交时延

提交时延。接下来，本组实验对比了所有事务日志技术的提交时延。图 5.7给出了它们在 YCSB 和 TPC-C 基准测试中的实验结果。在 YCSB 只写负载和 TPC-C 负载中， SILO 的平均事务提交时延明显高于其他事务日志技术。它的事务提交时延是 POPLAR 时延的 6 倍左右。 SILO 获得如此高的事务提交时延主要是因为它采用了基于 epoch 的事务提交协议。该协议要求只有当一个 epoch 内的所有事务日志记录都已持久化之后，事务才能提交。事务的提交时延由 epoch（$50ms$）决定。对于 CENTR 和 POPLAR 来说，在工作线程数较少的时候，它们的提交时延保持在 5ms 左右。此时事务提交时延基本上等于组提交协议中的组提交时间。随着工作线程数量的增多，由于每个日志段（一个日志持久化单元）里的日志记录会逐渐增多，因此日志持久化的时延会相应升高，从而导致事务时延变高。当工作线程的数量等于 12 时， POPLAR 和 CENTR 的磁盘 IO 带宽达到了最大值。此时系统的事务

提交时延也明显升高。这是因为大量的事务会因有限的 SSD 性能而处于等待状态。当工作线程超过 12 时，事务提交时延呈现出缓慢增长的趋势。这是因为工作线程因没有足够的日志缓冲区空间而被阻塞。而对于 NVM-D 来说，由于工作线程直接写盘，因此它的事务提交时延随着工作线程数量的增多而不断增加。除此之外，由于 NVM-D 没有采用组提交协议，而是每产生一条日志记录就进行持久化操作，因此它的提交时延小于其他事务日志技术的事务提交时延。

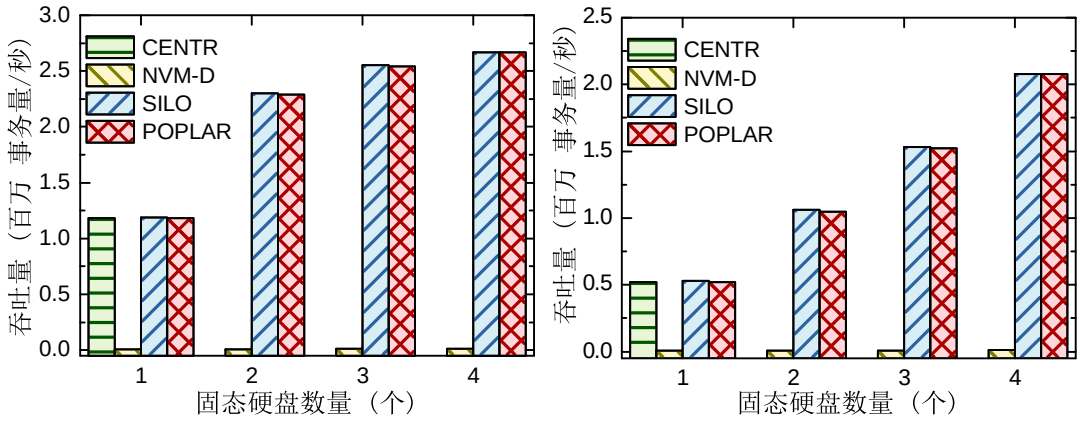


(a) 在 YCSB 负载中的结果　　(b) 在 TPC-C 负载中的结果

图 5.8: 不同事务日志技术的执行时间

执行时间分解。为了验证事务日志号分配的有效性，本组实验统计了事务在内存中的执行时间。执行时间主要由三部分组成：日志竞争、日志工作和其他。其中日志竞争主要包括事务日志技术中事务日志号（LSN、GSN、SSN）的分配时间；日志工作包括事务日志填充日志缓冲区的时间、事务等待提交的时间（主要有日志持久化时间构成）和事务提交时间；其他主要包括事务的逻辑处理时间和事务的执行验证时间。图 5.8给出了在 YCSB 和 TPC-C 基准测试中事务执行时间的分解结果。在 YCSB 只写负载和 TPC-C 负载中，由于所有事务日志技术都实现了高效的日志号分配方法，如基于原子指令的日志号分配以及去中心化的日志号分配，因此它们的日志竞争时间所占比例最小。本组实验统计的是 20 个工作线程时的事务执行时间。由于此时系统的 IO 已成为瓶颈，所以大量事务处于等待被持久化的状态。因此日志工作时间会相对比较高。对比 POPLAR 和 CENTR 的实验结果可以

发现增加 SSD 的数量能够减少事务的等待时间，从而可以提高系统的性能。通过这组实验结果可以得出 SSN 的分配方法是高效且可行的。

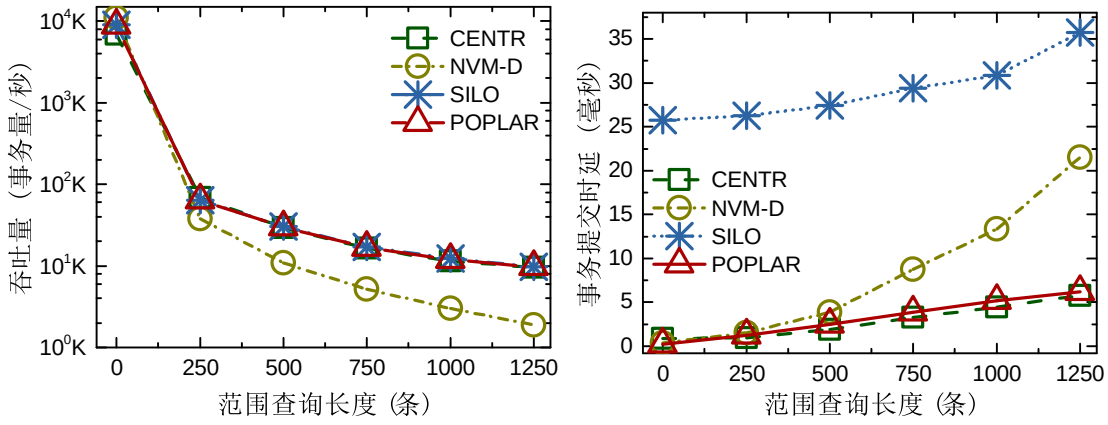


(a) 在 YCSB 负载中的结果　　(b) 在 TPC-C 负载中的结果

图 5.9: 不同事务日志技术的可扩展性

可扩展性。接下来，本组实验对比了所有事务日志技术的可扩展性。本组实验主要通过改变 SSD 的数量来观察各个事务日志技术的吞吐量变化情况。图 5.9给出了它们的实验结果。在 YCSB 只写负载和 TPC-C 负载中，当使用一块 SSD 时，POPLAR、SILO 和 CENTR 的事务吞吐量是一样的。但随着 SSD 数量的增多，POPLAR 和 SILO 的性能得到了不断提升。然而，由于 CENTR 采用的是集中式的日志缓冲区，因此 SSD 数量增多对它的性能没有任何影响。在 YCSB 负载中，当 SSD 数量超过 2 块时， POPLAR 和 SILO 的吞吐量并没有得到显著地提升。此时的性能主要受限于 CPU 的处理能力。如果增加 CPU 的核数，它们的吞吐量将得到提升。对于 NVM-D 来说，随着 SSD 数量的增多，它的吞吐量也在不断地升高。但由于它的吞吐量值太低，因此增长趋势在图中并不明显。

提交协议性能。最后，本实验分析了事务日志技术中不同事务提交协议对系统性能的影响。本组实验主要使用了 YCSB 的混合负载。该负载包含了一个写请求和一个范围查询请求。本组实验主要通过调节混合负载中范围查询的长度来测试各事务日志技术的事务吞吐量和事务提交时延。范围查询的长度决定了负载读取的数据项数量。为了避免磁盘 IO 带宽成为系统的瓶颈，本组所有实验都采用模拟



(a) 事务吞吐量　　(b) 事务提交时延

图 5.10: 不同提交协议的验证结果

的 NVM 来存储日志记录。在本组实验中，工作线程数量设置为 20， POPLAR 和 SILO 的日志线程数设置为 2。

图 5.10给出了所有事务日志技术在 NVM 上的事务吞吐量和事务提交时延。当范围查询的扫描长度等于 0 时（负载中只包含一个写请求），所有事务日志技术获得了相同的事务吞吐量，但是 SILO 的事务提交时延为 25.7ms，而其他技术只有 0.23ms。 SILO 的事务提交时延大约是其他技术的 112 倍。如此大的时延差距主要是因为 SILO 采用了一个基于 epoch 的粗粒度事务提交协议。该协议使得大量事务处于一个等待持久化状态，从而导致了事务提交时延升高。随着范围查询长度的增加，由于工作线程需要处理更多的事务逻辑，因此所有事务日志技术的事务吞吐量都逐渐下降并且事务提交时延逐渐升高。当范围查询长度增大后， NVM-D 的事务吞吐量和提交时延都比 POPLAR 差。这主要是因为在 NVM-D 事务日志技术中，事务日志号 GSN 需要追踪事务之间的 WAW、RAW 和 WAR 依赖。在分配事务日志号时， NVM-D 需要将新生成的 GSN 写入事务访问的数据项中（包括更新和只读数据项）。随着范围查询长度的扩大，由于事务读取的数据项会逐渐增多，因此计算 GSN 的代价也逐渐变大，从而导致事务吞吐量降低并且事务提交时延升高。而 POPLAR 的日志号 SSN 只需要追踪事务之间的 WAW 和 RAW 依赖，它不需要将新生成的 SSN 回写到只读数据项中，因此 POPLAR 的性能不受扫描长度的影响。

从实验结果可以看出， POPLAR 的细粒度事务提交协议能够拥有和 SILO 粗粒度事务提交协议一样的事务吞吐量，并且解决了 SILO 存在的高事务提交时延问题。此外，实现事务日志恢复性级别的 POPLAR 比实现事务日志严格性级别的 NVM-D 拥有更高的事务吞吐量和更低的事务提交时延。

5.5 本章小结

本项研究明确了在保证数据库系统正确性的基础上事务日志技术需要遵循的必要约束条件并给出了可恢复的事务日志定义。基于事务日志的可恢复性定义，本项研究实现了一种高效的事务日志技术（Poplar）。该技术消除了传统集中式日志的缓冲区竞争并允许事务日志记录被并行地写入多块磁盘。此外，它还彻底地打破了传统事务日志技术的顺序约束。在 Poplar 中，事务日志记录只需要保证事务之间的 WAW 和 RAW 依赖并且事务的提交顺序只需要遵循事务的 RAW 依赖。最后，本项研究在一个开源内存数据库原型系统中实现了 Poplar 并对其进行了性能评估。实验结果表明，Poplar 既拥有高事务吞吐量又拥有低事务提交时延，同时事务日志性能随着磁盘数量的增多呈线性扩展。

第六章　面向主备复制系统的自适应日志复制技术

6.1　引言

上一项研究消除了集中式事务日志技术的所有性能瓶颈，从而大幅度地提高了系统的事务处理能力。然而，随着事务处理性能的不断提升，传统的日志复制技术成为了限制主备复制数据库系统性能的最主要瓶颈。主备复制系统由一个主副本和多个备副本组成。为了保证系统的可用性，系统的主副本需要将生成的日志记录周期地通过以太网传输给异地的备副本。可扩展的并行事务日志技术使得主副本每秒能够产生超过 1.5GB 的日志记录量。而传统以太网，如万兆网，每秒最多只能处理 1.12GB 的数据量，这将导致大量的日志记录被阻塞在主副本。如果主备复制系统采用同步日志传输方法，那么系统的整体性能将急剧下降；如果系统采用异步日志传输方法，那么备副本的数据版本将远远落后于主副本。当主副本发生硬件媒介故障时，系统将丢失大量的数据。因此，无论采用哪种日志传输方法，有限的网络 IO 带宽都将限制主备复制系统的性能。

为了解决传统日志传输方法导致的性能瓶颈，本项研究提出了一种自适应的日志复制技术。该复制技术的核心思想是：主副本根据实时的负载情况选择传输事务日志记录或者数据增量给备副本。在处理低负载时（产生的事务日志量比较少时），主副本采用传统的日志传输方法；而在处理高负载时（产生的事务日志量比较多时），主副本采用增量传输方法。在密集型更新负载中，增量传输可以有效地减少网络传输量从而避免有限的网络 IO 成为系统的瓶颈。

本项研究结合自适应的日志复制技术和并行事务日志技术，实现了一个高性能的主备复制内存数据库原型系统（Plover）。然而，集成自适应日志复制技术和并行事务日志技术并不是一件容易的事。它主要面临了以下两个新的挑战：（1）在传统基于集中式事务日志技术的复制系统中，备副本接收到的是按全序排列的日志记录。通过回放全序的日志记录，备副本可以拥有和主副本一致的数据库快照。

然而，在并行事务日志技术中，事务的日志记录分布在多个日志缓冲区中且它们的日志号只提供了偏序顺序。如何将这些分散、偏序的日志记录整合成有序的快照版本成为了一个新的问题。（2）如何在传统日志传输方法和增量传输方法之间进行无缝切换，从而保证主备副本的数据一致性。为了解决这两个问题，本项研究实现了基于段的日志记录和增量数据合并算法。该算法动态地将日志记录和数据项划成多个复制段。每个复制段看作一个传输单元和一个回放单元。最后，本项研究使用 YCSB 和 TPC-C 基准测试评估了自适应日志复制技术的有效性。

综上所述。本项研究主要做出了以下贡献：

1. 本项研究提出了一种日志传输和增量传输相结合的自适应日志复制技术，该技术避免了有限网络 IO 带宽成为限制可扩展事务处理的性能瓶颈。

2. 本项研究结合自适应日志复制技术、并行事务日志技术和多版本并发控制协议实现了一个高性能的主备复制内存数据库原型系统（Plover）。

3. 本项研究验证了自适应日志复制技术的有效性。实验结果表明，在高负载应用中自适应日志复制技术产生的传输量是传统复制技术的 1/5，同时自适应日志复制技术在混合负载中能够始终获得最好的事务处理性能。

6.2 主备复制系统框架

本节介绍了主备复制内存数据库系统 Plover 的整体框架。如图 6.1所示，它主要包含一个主副本和一个备副本，主备副本之间通过万兆网络进行连接。主副本负责接收应用端的读写事务请求，然后周期性地将数据库状态通过网络传输给备副本。备副本接收到主副本的状态之后将它复制到自己的内存中。备副本可以提供一个通用的快照读 [47]。接下来，本节将分别介绍主备副本的功能模块：事务管理器、日志管理器和复制管理器。

主副本是在第四章的内存事务处理原型系统上进行的扩展。它主要包含一个多版本事务管理器、一个并行日志管理器和一个快速复制管理器。事务管理器实

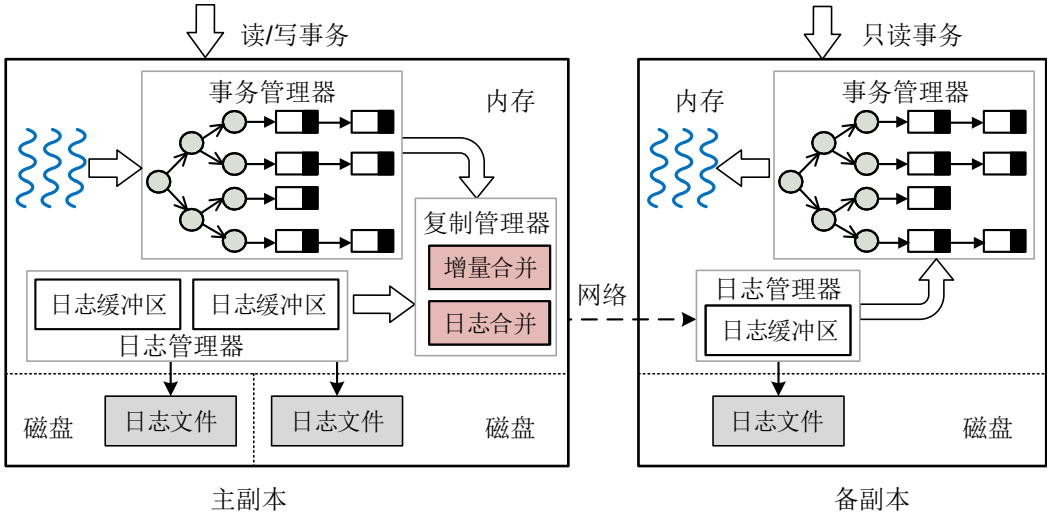


图 6.1: 主备高可用内存数据库系统框架

现了一种乐观多版本并发控制协议（MVOCC），它支持可串行化的事务隔离级别。所有的数据存储在一个多版本的内存表中。该表由一个 B 树和多个链表组成，B 树中的每个叶子节点连接一条链表。B 树中存储的是每个数据项的主键信息，它主要用于快速地检索数据项。每条链表存储了某一数据项的多个版本信息，链表中的每一个节点记录了数据项某一个版本的信息。

日志管理器实现了一种面向可扩展存储的并行事务日志技术。它使用多个日志缓冲区和多块存储设备来存储事务的日志记录。事务根据负载感知的日志分区策略将日志记录写入对应的日志缓冲区。每条日志记录分配了一个事务日志号 GSN。GSN 保证了事务之间的 RAW、WAW 和 WAR 依赖。此外，GSN 也存储在每个数据项中，它用来表示数据的版本信息，即版本号。

恢复管理器实现了一种自适应的日志复制技术。它主要实现了一种自适应的传输方法。该传输方法结合了两种传输手段：传统的日志传输和增量传输。日志传输方法是指主副本将多个日志缓冲区中的日志记录通过网络传输给备副本。增量传输方法是指主副本将内存中最新版本的增量数据传输给备副本。主副本可以根据实时的应用负载动态地选择一种合适的传输手段。在主备复制系统中，自适应传输方法采用了一种异步传输方式，即只要事务的日志记录持久化到主副本，事务就可以提交。此外，为了保证主备副本之间的数据一致性，日志复制技术实现了基

于段的日志记录/数据增量合并算法。该算法保证了主副本每次能够传输一个一致性的数据库快照给备副本。

备副本主要包含了一个集中式的日志管理器和一个多版本事务管理器。日志管理器将主副本传输的快照（日志记录或者增量）缓存在自己的集中式日志缓冲区中，然后再将它们串行地写入存储设备。事务管理器实现了基于段的回放方法。它按顺序将段里的快照版本信息并行地回放到内存多版本表中。只有当一个段里所有内容都回放成功之后，段对应的数据版本才能对外可见。

6.3 自适应日志复制

本节介绍了一种结合并行事务日志的自适应日志复制技术。本节首先概括了日志复制技术采用的三种关键方法：自适应的传输、基于段的合并以及基于段的回放，然后详细地描述了日志复制技术的执行流程。

6.3.1 整体架构

在基于异步传输方式的主备复制系统中，自适应的日志复制技术缩短了主副本与备副本之间的快照版本差距，从而使得备副本能够提供一个及时的快照读服务。为了消除因主副本产生大量日志记录而导致的网络 IO 带宽瓶颈，本项研究提出了一种自适应的传输方法。结合自适应传输方法和并行事务日志技术，本项研究实现了基于段的合并算法和基于段的回放方法。

自适应传输。随着事务处理性能的快速提升，主副本每秒能产生超过 10Gb 的事务日志记录。如此大的日志量远远超出了万兆网络（10Gbps）每秒能够处理的传输量（网络带宽）。因此，在基于日志记录传输的主备复制系统中，大量的事务日志记录将被阻塞在主副本中。随着被阻塞的事务越来越多，主备副本之间的数据版本差也将越来越大，最终甚至会导致备副本无法追赶上主副本的数据版本。在基于异步传输方式的主备复制系统中，如果主副本发生系统故障，那么数据库系统将面临丢失大量数据的风险。

表 6.1: TPC-C 负载中日志记录大小和增量大小的比较

事务类型	日志记录大小	增量大小
NewOrder	9500 字节	6292 字节
Payment	4180 字节	1256 字节
Delivery	820 字节	360 字节

为了减少主备复制系统中的网络传输量，本项研究提出了一种日志记录传输和增量传输相结合的传输方法。设计这种自适应传输方法的动机是：本项研究发现当处理密集型更新应用负载时，数据库系统在一段时间内产生的数据增量大小远远小于事务产生的日志记录量大小。因此，传输数据增量到备副本可以避免以太网成为系统性能瓶颈。为了验证这一现象，本项研究统计了一段时间内系统处理 TPC-C 应用负载时产生的数据增量大小和日志记录量大小。这里假设每 10 个事务中就有 2 个事务访问同一数据项。表 6.1给出了 TPC-C 中三种读写事务的比较结果。表中显示了系统处理 10 个事务时产生的日志记录量和数据增量。从表中可以看出产生的数据增量明显少于产生的事务日志记录量。因此，当主副本每秒产生的日志记录量超过了网络带宽时，主副本就选择增量传输的方式来减少网络传输量。当主副本每秒产生的日志记录量较少时，系统仍可以使用日志记录传输的方式。

基于段的合并。为了保证主备副本之间的数据一致性，主副本需要保证每次传输给备副本的日志记录或者数据增量来自一个一致性的数据库快照，同时主副本需要按照快照版本顺序依次传输给备副本。这种全序的一致性数据库快照在传统主备复制系统中是很容易实现的，因为日志记录按全序存储在一个集中式日志缓冲区中并且数据版本号也是基于一个集中式的时间戳来统一分配的。但是，在实现并行事务日志技术的主备复制系统中，由于日志记录的日志号和数据版本号是以分布式方式计算的（它们只提供了一个偏序关系）并且日志记录分布在多个日志缓冲区内，因此基于并行事务日志技术生成一个全序的快照版本就变得没那么容易了。此外，实现自适应传输方式的主备复制系统还需要保证日志记录产生的快照版本和数据增量产生的快照版本是一致的，这样才能在日志传输方法和增量传输方法切换时保证系统的正确性。

为了解决这些问题，本项研究提出了一种基于段的合并算法。该算法分别将日志缓冲区中的日志记录和内存中的增量数据根据它们的 GSN 分成多个复制段。GSN 既是日志记录的日志号也是数据项的版本号。一段连续 GSN 范围内的日志记录和增量属于同一个复制段。每个复制段被看作一个主副本的传输单元和一个备副本的回放单元。在实现过程中，合并算法结合并行事务日志技术中持久化的事务提交协议来生成复制段。也就是说，在并行事务日志技术的监控线程周期地计算出一个新的可提交事务日志号min_pgsn 之后，合并算法将两个新旧min_pgsn 之间的日志记录和数据增量组成一个复制段。虽然每个复制段里的日志记录和增量数据是无序的，但是由于复制段是满足全序关系，因此基于段的传输和回放可以保证主备副本之间的数据一致性。

基于段的回放。结合基于段的合并算法，本项研究在备副本上实现了一种基于段的回放方法。该方法允许回放线程并行地将复制段里的日志记录或者数据增量应用到内存表中。只有当一个复制段里的所有内容都回放完成之后，基于复制段生成的所有数据版本才能对外可见。此外，备副本按复制段的全序顺序进行回放。

6.3.2 执行流程

接下来，本节将结合自适应日志复制技术的三个主要方法来详细地介绍它的执行流程。图 6.2给出了日志复制技术的具体执行流程，它主要包含三个阶段：事务执行阶段、事务传输阶段和事务回放阶段。

事务执行。在主备复制系统的主副本中，事务首先生成一个新的数据版本并将数据版本追加到多版本表的链表尾部，然后生成事务日志记录并写入对应的日志缓冲区。当触发组提交时，日志线程将对应日志缓冲区内的日志记录写到磁盘，然后更新本地已持久化的日志号$pgsn$。之后，持久化组提交协议的监控线程周期地监测每个日志线程的$pgsn$，然后计算出它们中的最小值（min_pgsn）。min_pgsn 表示当前最大可提交的事务日志号。因此，事务日志号小于等于min_pgsn 的事务可以提交并且版本号小于等于min_pgsn 的数据项可以被其他事务访问。在图 6.2中，

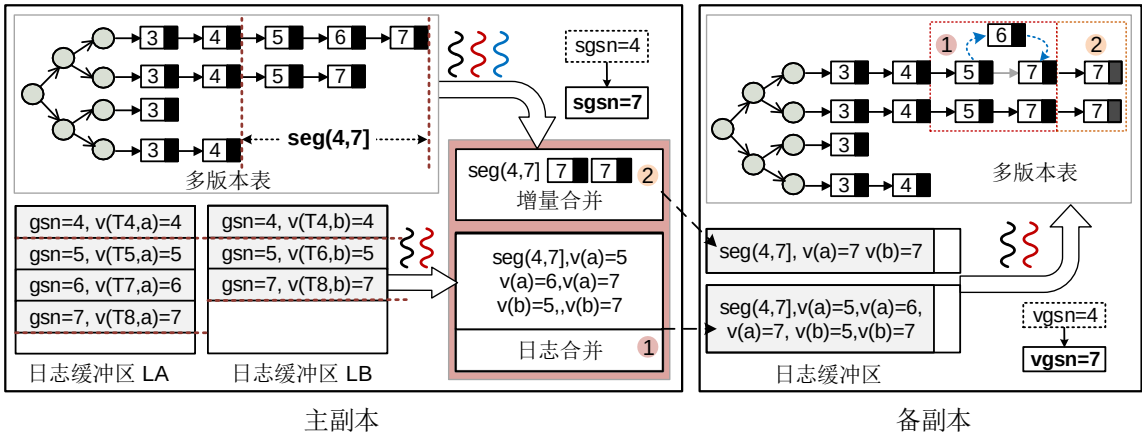


图 6.2: 日志复制的执行流程

当前的min_pgsn 等于 7。

事务传输。主副本首先结合min_pgsn 来生成复制段。这里将min_pgsn 标记为$sgsn$。它根据$sgsn$ 将数据增量和日志记录动态地分成多个复制段。处于两个相邻$sgsn$ 内的日志记录和数据增量属于一个复制段。如图 6.2所示，最新的一个复制段标记为$seg(4,7]$。事务日志号和版本号在 4 和 7 之间的日志记录和增量数据属于同一个复制段。然后，主副本根据当前每秒产生的日志记录量选择一种适合的传输方法。如果产生的日志记录量不大，那么它采用日志传输方法，即将属于$seg(4,7]$的日志记录传输给备副本。否则，它使用增量传输方法，即将属于$seg(4,7]$ 的最新数据增量传输给备副本。最后，主副本根据传输方式将来自多个日志缓冲区中的日志记录或者来自内存表中的增量数据合并成一个传输流并结合复制段信息一起传输给备副本。主副本采用了一个并行的合并方式。虽然这种方式可能使得传输流中的日志记录或者增量不是按序排列的，但是备副本实现的基于复制段的回放方法可以保证数据库处于一个一致性的状态。

事务回放。当接收到来自主副本的传输包之后，备副本首先解压传输包并将$seg(4,7]$ 里的日志记录或者数据增量填充到集中式日志缓冲区，然后将复制段中的内容写入存储设备。之后，备副本首先读取集中式日志缓冲区中的日志记录或者增量数据，然后使用多个回放线程并行地将其应用到多版本内存表中。备副本在本地维护了一个$vgsn$，它表示对外可见的数据项版本号。当$seg(4,7]$ 段里所有

的日志记录或者数据增量都回放完成之后，备副本将$vgsn$ 修改为 7，即版本号小于等于 7 的数据项可以被其他事务访问。

在实现过程中，日志复制技术考虑了两中容错的情况：网络数据包丢失和备副本崩溃。当出现网络数据包丢失时，备副本接收到的复制段是不连续的。当发现复制段不连续时，备副本主动从主副本中拉取丢失的复制段然后进行回放。当备副本发生系统故障时，备副本首先恢复本地日志文件中的内容，然后主动从主副本中拉取滞后的快照版本进行回放。最终保证主备副本之间的数据一致性。

6.4 实验与分析

本项研究实现了一个高性能的主备复制内存数据库原型系统 Plover，本节主要验证了自适应日志复制技术的有效性，并得出了以下结论：

1. 相比于传统日志记录传输方法，增量传输方法大大地减少了网络传输量，从而避免了有限的网络带宽导致系统性能降低。

2. 结合自适应的传输方法，主备复制系统拥有更好的回放性能。

3. 在混合负载下，自适应的日志复制技术能够有效地避免主备副本之间的数据版本差，从而使得备副本一直可以对外提供只读服务。

6.4.1 实验配置

实验环境。本实验分别将主备复制系统 Plover 部署在两台硬件配置相同的 Linux 服务器上。每台服务器包含两个 Intel Xeon E5-2630 v4 @2.20GHZ 的 CPU 处理器（总共 20 个物理核）、256GB 的 DRAM 和 2 块 SATA 接口的 SSD。服务器之间通过万兆（10Gb / s）以太网进行连接。

实验对比项。本实验主要对比了以下两种不同的日志复制技术：

1. SHIP。它表示基于传统日志记录传输方法的日志复制技术。该传输方法每隔 50ms 将日志记录传输给备副本。

2. ADAP。它表示基于自适应传输方法的日志复制技术。该技术根据实时的应用负载动态地选择使用日志记录传输方法或者数据增量传输方法。当使用数据增量传输方法时，主副本每隔 1s 执行一次传输。

默认情况下，所有日志复制技术都基于并行事务日志技术进行实现，并且采用了基于段的合并算法和基于段的备副本回放方法。

6.4.2 工作负载

本实验主要使用 YCSB 和 TPC-C 两种基准测试来测试自适应日志复制技术。

YCSB。 YCSB 基准测试 [88] 是一种典型的用于测试大规模在线云服务系统的读写性能。由于在复制过程中不需要考虑事务的 ACID 属性，因此 YCSB 适合用来评估主备复制系统的日志复制性能。YCSB 基准测试主要包含一张测试表和五种测试负载。该测试表由一个 64 位的整型列（主键）和十个 100 字节的字符串列组成。在本实验中，测试表中包含 1000 万条数据记录。此外，本实验主要使用了两种不同的工作负载：

1. 只写负载。此负载包含一个只写事务。该写事务每次只修改一条数据记录。

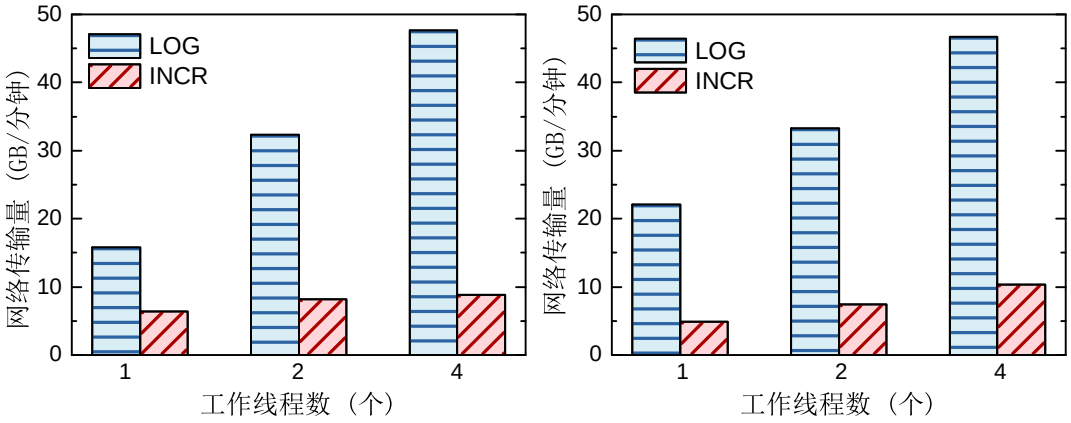
2. 混合负载。此负载包含一个只读事务和一个只写事务。只读事务可以在备副本上执行。本实验设置了三种不同的读/写事务比例：10/90，50/50 和 90/10。

所有的负载都服从 Zipfian 分布。其中，负载的倾斜比例 ζ 设置为 0.6，它表示大约有 40% 的读/写事务会集中访问 10% 的数据记录。

TPC-C。 TPC-C 基准测试是一种衡量在线事务处理（OLTP）系统的工业测试标准。它模拟了一个以仓库为中心的大型批发销售管理系统 [95]。本实验设置了 20 个仓库数。标准 TPC-C 包含了三个读写事务和两个只读事务。它属于一个更新密集型的应用负载，其中读写事务占了总事务的 90%。本实验只使用了三种读写负载：NewOrder、Payment 和 Delivery，它们的比例分别为 48%、48% 和 4%

6.4.3 实验结果和分析

本实验首先对比了 SHIP 和 ADAP 两种日志复制技术的网络传输量、日志滞留量以及回放性能，然后测试了基于两种复制技术的主备复制系统整体性能。

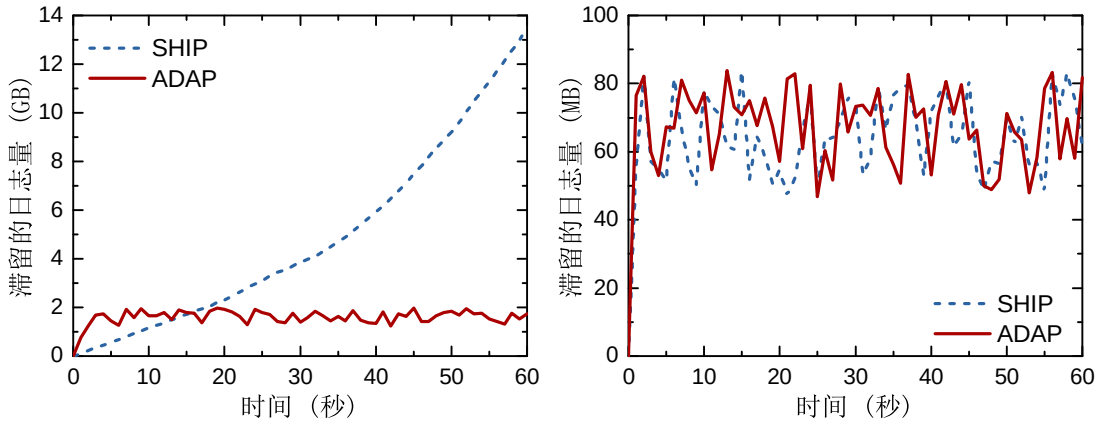


(a) 在 YCSB 中的网络传输量　　(b) 在 TPC-C 中的网络传输量

图 6.3: 不同传输方法的网络传输量

网络传输量。为了验证增量传输的有效性，本组实验比较了日志传输方法和增量传输方法产生的网络传输量。本组实验使用万兆（10Gbps）以太网来传输日志记录或者数据增量。万兆以太网理论上每秒能够传输 1.16GB 数据，但是在实际系统中它每秒只能传输 800MB 数据。图 6.3给出了在 YCSB 和 TPC-C 基准测试中不同传输方法每分钟产生的日志记录量（ LOG）和数据增量（ INCR）。在 YCSB 只写负载和 TPC-C 负载中，随着工作线程数量的增加，基于日志传输方式的主副本每分钟能够产生高达 46GB 以上的日志记录（ LOG）。如此高的网络传输量超出了万兆以太网的实际传输量。而基于增量传输的主副本每分钟只产生 10GB 左右的数据增量（ INCR）。实验结果表明，在 YCSB 只写负载中，日志传输方法产生的网络传输量是增量传输方法的 5.3 倍；在 TPC-C 负载中，日志传输方法产生的网络传输量是增方法的 4.5 倍。

日志滞留量。接下来，本组实验测试了不同传输方法在不同 TPC-C 负载下产生的日志滞留量。所谓的日志滞留量是指一直存储在主副本上没有传输给备副本的日志记录。为了测试日志滞留量，本组实验统计了每秒主副本产生的日志量和



(a) 在高负载 TPC-C 中的结果　　(b) 在低负载 TPC-C 中的结果

图 6.4: 不同传输方法产生的日志滞留量

已传输给备副本的日志量。图 6.4给出了 60 秒内日志记录传输方法和自适应传输方法产生的日志滞留量。

图 6.4(a)给出了在高负载 TPC-C 下两种传输方法的日志滞留量结果。在处理高负载应用时，主副本每秒大约能产生 821MB 的日志记录。这个日志量超过了万兆网的每秒传输量。当使用日志记录传输方法（ SHIP）时，日志记录会因有限的网络带宽被阻塞在主副本中。随着时间的流逝，滞留的日志量会越来越多。但是，当使用自适应传输方式（ ADAP）时，主副本根据当前的产生的日志量选择传输增量数据给备副本。由于主副本每秒只产生大约 262MB 的数据增量，因此主副本中几乎不会产生滞留的日志量。图中 ADAP 产生的日志滞留量一直保持在一个稳定值，这是因为本实验每隔 1 秒传输一次数据增量， ADAP 产生的日志滞留量等于主副本 1 秒内产生的日志记录量。

图 6.4(b)显示了在低负载 TPC-C 下两种传输方法的日志滞留量结果。在处理低负载应用时，主副本每秒大约只产生 377MB 的日志记录。产生的日志量没有超过万兆网的每秒传输量。因此，不论使用日志传输方法 SHIP 还是自适应传输方法 ADAP，主副本上都不会产生大量滞留的日志记录。

回放性能。接下来，本组实验测试了基于不同日志复制技术的备副本回放性能。为了避免网络因素影响实验结果，本组实验首先收集了主副本执行一千万个

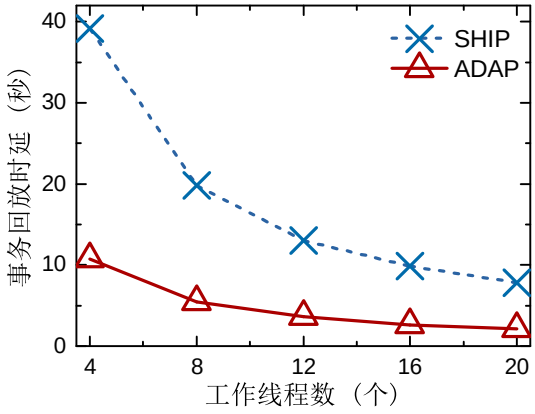


图 6.5: 不同传输方法的回放性能

TPC-C 事务之后产生的日志记录和数据增量，然后测试了备副本回放所有日志记录或者数据增量需要花费的时间。图 6.5显示了备副本的回放性能。当系统使用日志记录传输（ SHIP）时，备副本需要回放所有的日志记录。而当系统使用自适应传输（ ADAP）时，备副本只需要回放所有数据项的最新数据版本即可。因此，基于 ADAP 的备副本回放性能是基于 SHIP 的 3.7 倍。

整体性能。最后，本组实验使用 YCSB 的混合负载测试了主备复制内存数据库系统（Plover）的整体性能。该混合负载设置了三种读写事务比例：R/W = 10/90、R/W = 50/50 和 R/W = 90/10。本组实验规定只有当主备副本之间的数据版本相差不超过 10 秒时，只读事务才能发送到备副本进行处理。

图 6.6显示了基于不同日志复制技术的 Plover 在三种混合负载中的实验结果。在所有混合负载中，基于自适应传输方法（ ADAP）的主备复制系统始终保持了最高的事务吞吐量。而基于传统日志传输方法（ SHIP）的主备复制系统表现出了不同的性能趋势。接下来本小节将分别分析基于 SHIP 的主备复制系统在三种混合负载中事务吞吐量的变化原因。

当处理 R/W = 10/90 的混合负载时，基于 SHIP 的主备复制系统起初获得了和基于 ADAP 主备复制系统一样的事务吞吐量。然而，随着时间的流逝，基于 SHIP 的主备复制系统的性能会突然降低。这是因为主副本每秒产生的日志量超出了万兆网的每秒传输量，大量日志记录会被阻塞在主副本上。随着滞留日志量的增加，

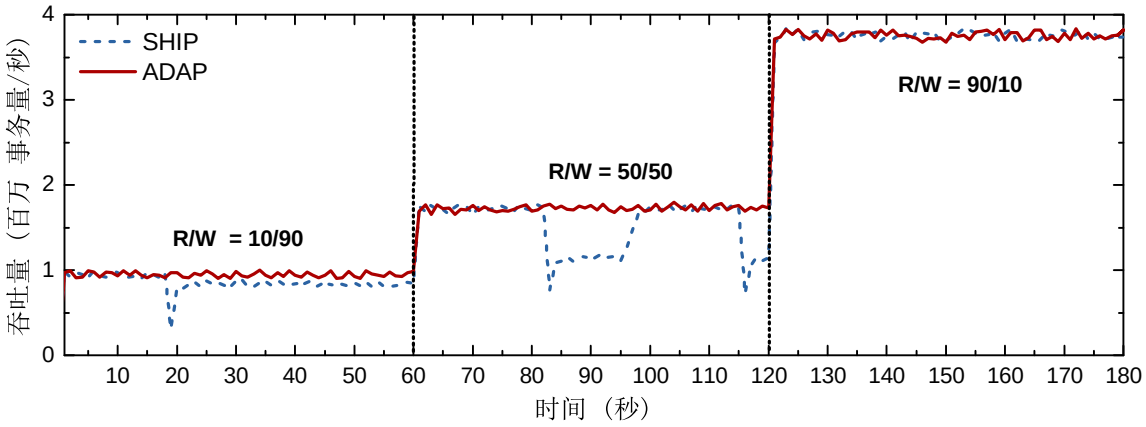


图 6.6: 主备复制系统的整体读写性能

主备副本上的数据版本差会越来越大。当它们之间的数据项版本差超过 10 秒时，备副本将不能提供读服务，因此基于 SHIP 的主备复制系统的整体性能会骤然下降。之后，所有的只读事务和只写事务都将发送给主副本进行处理，由于主副本有限的 CPU 处理能力，因此主备复制系统的性能会低于初始值。然而，由于在 R/W = 10/90 的混合负载中只写事务的比例比较高，即使系统的事务吞吐量比较低，主副本每秒产生的日志量仍然超出了网络的实际传输量，因此大量日志记录仍被阻塞在主副本中，从而导致主备复制系统的吞吐量一直处于较低的状态。

当处理 R/W = 50/50 的混合负载负载时，基于 SHIP 的主备复制系统会因为大量日志记录被滞留在主副本中从而导致事务吞吐量下降。当系统的事务吞吐量比较低时，由于主副本每秒产生的日志量也会相应地减少（远远低于网络带宽），因此大量被滞留的日志量将被传输给备副本。随着主备副本之间数据版本差缩小之后，只读事务能够发送给备副本进行处理，因此主备复制系统的事务吞吐量又将恢复到最初的状态。但是，随着系统事务吞吐量的升高，主备副本之间的数据版本差又将再一次扩大，从而导致主备复制系统性能又一次降低。

当处理 R/W = 90/10 的混合负载时，由于混合负载中只写事务的比例比较少，主副本每秒产生的日志量将不会超过万兆网的实际使用带宽，因此基于 SHIP 的主备复制系统将一直保持较高的事务吞吐量。

6.5 本章小结

为了保证系统的高可用性，主备复制系统通常采用日志传输方式将主副本上的数据库状态复制到备副本上。然而，随着主副本事务处理能力的提升，系统每秒将产生大量的日志记录。超高的日志量将远远超出以太网的带宽从而使得大量日志记录被滞留在主副本上。滞留的日志量将导致主备副本的数据版本差变大，从而降低系统的整体性能。为了解决这个问题，本项研究提出了一种自适应的日志复制技术。该技术能够根据实时的应用负载选择要么传输日志记录要么传输数据增量给备副本，从而避免了有限的网络带宽影响系统性能。此外，本项研究集成了自适应日志复制技术和并行事务日志技术并将它们实现在一个主备复制的内存数据库原型系统中。最后，本项研究验证了自适应复制技术的有效性。实验结果表明，自适应复制技术产生了更少的网络传输量，拥有更好的回放性能，同时在混合负载中能够具有更稳定的事务处理性能。

第七章　总结与展望

7.1　研究总结

从二十世纪七十年代中期以来，事务日志技术和日志复制技术一直是关系型数据库系统中不可或缺的事务处理技术，它们保证了数据库系统的可靠性和可用性。互联网的快速发展使得数据库系统需要具备可扩展、高性能的事务处理能力。随着多核 CPU 和大容量内存的出现，数据库系统由原来面向磁盘存储、单核 CPU 架构的繁重、串行的事务执行方式逐渐转变为面向内存存储、多核 CPU 的轻量级、并行的事务执行方式。这种新的设计架构大幅度地提高了系统的事务处理性能。然而，事务日志技术和日志复制技术仍采用传统的集中式设计和串行执行方式，从而严重影响了数据库系统的可扩展性和并行性。为此，本文全面、深入地分析了传统数据库日志技术在多核硬件平台中存在的性能瓶颈，并实现了新型的事务日志技术和日志复制技术来支持数据库系统可扩展、高性能、高通量的事务处理需求。本文的主要贡献主要包括以下四点：

1. 针对传统事务日志技术的集中式日志缓冲区竞争和固定组提交问题，本文提出了一种可扩展、自适应的集中式事务日志技术 *Laser*。该技术实现了基于原子指令的日志号分配方法和并行日志填充方法，从而提高了数据库系统的可扩展性。结合动态变化负载，该技术实现了自适应的组提交协议，从而保证了数据库系统在动态负载中能够一直保持最大的吞吐量和最小的事务提交时延。此外，本文基于开源分布式数据库系统 CEDAR 实现了新提出的事务日志技术并对其进行了性能评估。实验结果表明，Laser 具有多核可扩展性，并且事务吞吐量是传统集中式事务日志技术的 1.4 倍，事务提交时延比传统事务日志技术降低了 86%。

2. 针对传统事务日志技术串行持久化导致的有限磁盘 *IO* 带宽问题，本文提出了一种面向可扩展存储的并行事务日志技术。该技术使用多个日志缓冲区和

多块磁盘来分担高通量日志数据流对磁盘 IO 的负载压力。为了避免负载倾斜和跨分区事务对系统事务处理性能的影响，该技术实现了负载感知的日志分区策略。为了保证数据库系统的正确性和可恢复性，该项技术实现了偏序的日志号（GSN）和持久化组提交协议。此外，本文结合并行事务日志技术、乐观多版本并发控制协议以及并行恢复方法，实现了一个内存事务处理原型系统（Plover）并对其进行了性能评估。实验结果表明，Plover 的事务处理性能能够随着磁盘数量的增加呈线性扩展。

3. 针对传统事务日志技术的顺序性约束问题，本文提出了一种支持系统可恢复性的偏序事务日志技术（*Poplar*）。该技术首先明确了事务日志在保证系统可恢复性和正确性的基础上需要具备的必要约束条件，然后给出了事务日志可恢复性的定义。基于定义，该技术允许事务并行地将日志记录写入多个日志缓冲区和多块磁盘，并且实现了一种可扩展的偏序日志号（SSN）和快速的事务提交协议。此外，本文基于一个开源的内存数据库原型系统 DBX1000 实现了 Poplar 并对它进行了性能评估。实验结果表明，当部署两块固态硬盘时，Poplar 的事务吞吐量是其他事务日志技术的 2 倍——280 倍，事务提交时延相比其他事务日志技术降低了 5 倍。

4. 针对传统日志复制技术的有限网络 *IO* 带宽问题，本文提出了一种面向主备复制数据库系统的自适应日志复制技术。该技术能够根据实时的应用负载选择传输日志记录或者数据增量给远端的备副本。在处理密集型更新负载时，增量传输方式能够减少网络传输量，从而避免网络成为系统的性能瓶颈。此外，本文结合自适应传输方法和并行事务日志技术，实现了一个高性能的内存主备复制原型系统（Plover）。该系统采用基于段的日志和增量合并算法保证了主备副本之间的数据一致性。实验结果表明，在高负载应用中自适应日志复制技术产生的日志传输量是传统日志复制技术的 1/5 并且自适应日志复制技术在混合负载中表现出了更稳定的性能。

综上所述，为了解决传统事务日志技术和日志复制技术在多核 CPU 平台下存在的四个性能瓶颈，本文实现了一些新型的数据库日志技术，该技术满足了数据库系统高可扩展、高性能、高通量的事务处理需求。

7.2 未来展望

本文主要从数据库日志技术角度优化了数据库系统的事务处理性能。为了更加全面地提高数据库系统性能，后续还可以探索的方向包括且不局限于以下方面：

1. **混合隔离级别下的事务日志技术**。本文提出的所有事务日志技术都是针对可串行化隔离级别 [48] 进行设计的。然而，在实际应用过程中，大多数据库系统还实现了读已提交隔离级别和快照读隔离级别。由于读已提交和快照读隔离级别弱化了事务语义，因此事务日志技术也可以相应的减少依赖约束，从而进一步提高事务处理性能。因此，未来可以考虑实现支持低隔离级别的事务日志技术以及支持混合隔离级别的事务日志技术。

2. **基于 NVM 的数据库日志技术**。具有稳定性和低访问时延特性的新型非易失内存设备的出现为实现高吞吐量、低时延的事务处理带来了新的契机 [101]。数据库系统可以直接将数据存储在非易失内存设备中。这种基于单层存储层级的数据库架构打破了传统事务日志技术遵循的先写日志协议。因此，未来可以考虑实现面向 NVM 的数据库日志技术来保证系统的可靠性和可用性。

3. **高性能网络环境下的分布式事务处理技术**。远程直接内存访问（RDMA）技术的出现消除了传统网络对分布式数据库系统性能的影响 [102]。然而，简单地将数据库系统迁移到支持 RDMA 的高速网络下并不能充分地发挥高性能网络的优势。因此，未来可以考虑如何利用 RDMA 引入的新通信栈实现高性能、可扩展的分布式事务处理技术，如分布式提交协议、并发控制协议以及日志复制技术。

4. **高冲突下多热点数据的事务调度算法**。在处理高冲突、高并发的事务请求时，传统基于单队列先来先服务的事务调度算法严重影响了数据库系统的并行性 [103]。未来可以考虑使用多队列的事务调度算法来分别处理冲突事务和没有冲突的事务，从而提高数据库系统的并行度。此外，还可以结合存在多热点的应用负载，考虑如何动态的调整队列个数和硬件资源，从而使得数据库系统能够一直保持最优的事务处理能力。

5. **多级混合存储的自适应数据迁移技术**。在大容量硬件内存平台下，大多数据库系统实现了多级混合存储模式，即内存中存储热数据，磁盘中存储冷数据 [104]。然而，随着应用负载的变化，数据的访问热度会发生改变，从而导致原始的数据分布失效。频繁的数据失效将导致数据库系统性能急剧下降。因此，未来可以考虑结合应用负载动态地统计数据块之间的访问特征和冷热转换的触发条件，实现自动地冷热数据迁移。

参考文献

[1] WEIKUM G, VOSSEN G. Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery[M]. [S.l.]: Morgan Kaufmann, 2002.

[2] GRAY J, REUTER A. Transaction processing: Concepts and techniques[M]. [S.l.]: Morgan Kaufmann, 1993.

[3] BERNSTEIN P A, NEWCOMER E. Principles of transaction processing for systems professionals[M]. [S.l.]: Morgan Kaufmann, 1996.

[4] BERNSTEIN P A, HADZILACOS V, GOODMAN N. Concurrency control and recovery in database systems[M]. [S.l.]: Addison-Wesley, 1987.

[5] FRANKLIN M J. Concurrency control and recovery[M]//The Computer Science and Engineering Handbook. [S.l.]: CRC Press, 1997: 1058-1077.

[6] MOHAN C, HADERLE D J, LINDSAY B G, et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging[J]. ACM Trans. Database Syst., 1992, 17(1):94-162.

[7] 中国中央网络安全和信息化委员会办公室. 第 43 次中国互联网络发展情况统计报告[J]. 中国互联网信息中心, 2019, 2(1):1-135.

[8] 新华网. 2018 天猫“双 11”约两分钟成交额就突破百亿元[EB/OL]. 2018. http://www.xinhuanet.com/2018-11/11/c_1123694692.html.

[9] 罗乐，刘轶，钱德沛. 内存计算技术研究综述[J]. 软件学报, 2016, 27(8):2147-2167.

[10] HARIZOPOULOS S, ABADI D J, MADDEN S, et al. OLTP through the looking glass, and what we found there[C/OL]//SIGMOD. ACM, 2008: 981-992. https://doi.org/10.1145/1376616.1376713.

[11] 阿里巴巴数据库开发组. OceanBase[EB/OL]. 2014. https://github.com/alibaba/oceanbase.

[12] CORPORATION O. Mysql 8.0 reference manual[EB/OL]. 2019. https://dev.mysql.com/doc/refman/8.0/en/.

[13] GROUP I P G D. Postgresql:the world's most advanced open source relational database[EB/OL]. 1996. https://www.postgresql.org/.

[14] PingCAP 数据库开发组. TiDB 开源分布式关系型数据库[EB/OL]. 2018. https://www.pingcap.com/docs-cn/.

[15] DIACONU C, FREEDMAN C, ISMERT E, et al. Hekaton: SQL server's memory-optimized OLTP engine[C/OL]//SIGMOD. ACM, 2013: 1243-1254. https://doi.org/10.1145/2463676.2463710.

[16] CEDAR 项目组. CEDAR[EB/OL]. 2016. https://github.com/daseECNU/Cedar/.

[17] ZHU T, ZHAO Z, LI F, et al. Solar: Towards a shared-everything database on distributed log-structured storage[C/OL]//ATC. USENIX Association, 2018: 795-807. https://www.usenix.org/conference/atc18/presentation/zhu.

[18] STONEBRAKER M, MADDEN S, ABADI D J, et al. The end of an architectural era (it's time for a complete rewrite)[C/OL]//PVLDB. ACM, 2007: 1150-1160. http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf.

[19] STONEBRAKER M, MADDEN S, ABADI D J, et al. Voltdb[EB/OL]. 2015. https://github.com/VoltDB/voltdb.

[20] LEVANDOSKI J J, LOMET D B, SENGUPTA S, et al. High performance transactions in deuteronomy[C/OL]//CIDR. www.cidrdb.org, 2015. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper15.pdf.

[21] KIM K, WANG T, JOHNSON R, et al. ERMIA: fast memory-optimized database system for heterogeneous workloads[C/OL]//SIGMOD. ACM, 2016: 1675-1687. https://doi.org/10.1145/2882903.2882905.

[22] YU X, BEZERRA G, PAVLO A, et al. Dbx1000[EB/OL]. 2014. https://github.com/yxymit/DBx1000.

[23] COMMUNITY I. Ibm soliddb[EB/OL]. 2014. http://www.ibm.com/software/data/soliddb.

[24] CORPORATION O. Timesten[EB/OL]. 2013. https://www.oracle.com/database/technologies/related/timesten.html.

[25] LAHIRI T, NEIMAT M, FOLKMAN S. Oracle timesten: An in-memory database for enterprise applications[J]. IEEE Data Eng. Bull., 2013, 36(2):6-13.

[26] O'NEIL P E, CHENG E, GAWLICK D, et al. The log-structured merge-tree (lsm-tree)[J]. Acta Inf., 1996, 33(4):351-385.

[27] THOMSON A, DIAMOND T, WENG S, et al. Calvin: fast distributed transactions for partitioned database systems[C/OL]//SIGMOD. ACM, 2012: 1-12. https://doi.org/10.1145/2213836.2213838.

[28] YU X, PAVLO A, SÁNCHEZ D, et al. Tictoc: Time traveling optimistic concurrency control[C/OL]//SIGMOD. 2016: 1629-1642. https://doi.org/10.1145/2882903.2882935.

[29] LIM H, KAMINSKY M, ANDERSEN D G. Cicada: Dependably fast multi-core in-memory transactions[C/OL]//SIGMOD. ACM, 2017: 21-35. https://doi.org/10.1145/3035918.3064015.

[30] NEUMANN T, MÜHLBAUER T, KEMPER A. Fast serializable multi-version concurrency control for main-memory database systems[C/OL]//SIGMOD. ACM, 2015: 677-689. https://doi.org/10.1145/2723372.2749436.

[31] LEVANDOSKI J J, LOMET D B, SENGUPTA S. The bw-tree: A b-tree for new hardware platforms[C/OL]//ICDE. IEEE Computer Society, 2013: 302-313. https://doi.org/10.1109/ICDE.2013.6544834.

[32] ARULRAJ J, LEVANDOSKI J J, MINHAS U F, et al. Bztree: A high-performance latch-free range index for non-volatile memory[J]. PVLDB, 2018, 11(5):553-565.

[33] ASTRAHAN M M, BLASGEN M W, CHAMBERLIN D D, et al. System R: relational approach to database management[J]. ACM Trans. Database Syst., 1976, 1(2):97-137.

[34] HAGMANN R B. Reimplementing the cedar file system using logging and group commit[C/OL]//SOSP. ACM, 1987: 155-162. https://doi.org/10.1145/41457.37518.

[35] JOHNSON R, PANDIS I, STOICA R, et al. Aether: A scalable approach to logging [J]. PVLDB, 2010, 3(1):681-692.

[36] JUNG H, HAN H, KANG S. Scalable database logging for multicores[J]. PVLDB, 2017, 11(2):135-148.

[37] TU S, ZHENG W, KOHLER E, et al. Speedy transactions in multicore in-memory databases[C/OL]//SOSP. ACM, 2013: 18-32. https://doi.org/10.1145/2517349.2522713.

[38] HUANG J, SCHWAN K, QURESHI M K. Nvram-aware logging in transaction systems[J]. PVLDB, 2014, 8(4):389-400.

[39] WANG T, JOHNSON R. Scalable logging through emerging non-volatile memory [J]. PVLDB, 2014, 7(10):865-876.

[40] GOSWAMI H S. Microsoft sql server 2008 high availability[J]. 2011.

[41] COMMUNITY I. High availability through log shipping[J]. 2015.

[42] CORPORATION O. Chapter 16 replication[J]. 2015.

[43] CORPORATION O. Chapter 16 replication[EB/OL]. 2015. https://dev.mysql.com/doc/refman/5.7/en/replication.html.

[44] GROUP T P G D. Chapter 26. high availability, load balancing, and replication[J]. 2019.

[45] GROUP I P G D. Postgresql high availability[EB/OL]. 1996. https://www.postgresql.org/docs/11/high-availability.html.

[46] HELLERSTEIN J M, STONEBRAKER M, HAMILTON J R. Architecture of a database system[J]. Foundations and Trends in Databases, 2007, 1(2):141-259.

[47] ELNIKETY S, ZWAENEPOEL W, PEDONE F. Database replication using generalized snapshot isolation[C/OL]//SRDS. IEEE Computer Society, 2005: 73-84. https://doi.org/10.1109/RELDIS.2005.14.

[48] BERENSON H, BERNSTEIN P A, GRAY J, et al. A critique of ANSI SQL isolation levels[C/OL]//SIGMOD. ACM Press, 1995: 1-10. https://doi.org/10.1145/223784.223785.

[49] HONG C, ZHOU D, YANG M, et al. Kuafu: Closing the parallelism gap in database replication[C/OL]//ICDE. IEEE Computer Society, 2013: 1186-1195. https://doi.org/10.1109/ICDE.2013.6544908.

[50] SCHWARTZ B. Mysql limitations:single-threaded replication[EB/OL]. 2010. https://www.percona.com/blog/.

[51] ZHENG W, TU S, KOHLER E, et al. Fast databases with fast durability and recovery through multicore parallelism[C/OL]//OSDI. USENIX Association, 2014: 465-477. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting.

[52] QIN D, GOEL A, BROWN A D. Scalable replay-based replication for fast databases[J]. PVLDB, 2017, 10(13):2025-2036.

[53] WANG T, JOHNSON R, PANDIS I. Query fresh: Log shipping on steroids[J]. PVLDB, 2017, 11(4):406-419.

[54] GRAY J. Notes on data base operating systems[C/OL]//Operating Systems: volume 60. Springer, 1978: 393-481. https://doi.org/10.1007/3-540-08755-9_9.

[55] 陈娟，胡庆达，陈游旻，陆游游，舒继武，杨晓辉. 一种基于微日志的持久性事务内存系统[J]. 计算机研究与发展, 2018, 55(9):2029-2037.

[56] GRAY J, MCJONES P R, BLASGEN M W, et al. The recovery manager of the system R database manager[J]. ACM Comput. Surv., 1981, 13(2):223-243.

[57] 宋伟，杨学军. 基于事务回退的事务存储系统的故障恢复[J]. 软件学报, 2011, 22(9):2248-2262.

[58] LIN J, DUNHAM M H, NASCIMENTO M A. A survey of distributed database checkpointing[J]. DPD, 1997, 5(3):289-319.

[59] CORPORATION O. Oracle database[EB/OL]. 1979. https://www.oracle.com/database/.

[60] MICROSOFT. Microsoft sql server[EB/OL]. 1989. https://www.microsoft.com/en-us/sql-server.

[61] 张铁赢，黄贵，章颖强，王剑英，胡炜，赵殿奎，何登成. X-DB：软硬一体的新型数据库系统[J]. 计算机科学与发展, 2018, 55(2):319-326.

[62] HELLAND P, SAMMER H, LYON J, et al. Group commit timers and high volume transaction systems[C/OL]//HPTS: volume 359. Springer, 1987: 301-329. https://doi.org/10.1007/3-540-51085-0_52.

[63] RAFII A, DUBOIS D. Performance tradeoffs of group commit logging[C/OL]//CMGC. Computer Measurement Group, 1989: 164-176. http://www.cmg.org/?s2member_file_download=/proceedings/1989/89INT017.pdf.

[64] GROUP I P G D. Postgresql asynchronous commit[EB/OL]. 1996. https://www.postgresql.org/docs/8.3/wal-async-commit.html.

[65] KEMPER A, NEUMANN T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots[C/OL]//ICDE. IEEE, 2011: 195-206. https://doi.org/10.1109/ICDE.2011.5767867.

[66] DEWITT D J, KATZ R H, OLKEN F, et al. Implementation techniques for main memory database systems[C/OL]//SIGMOD. 1984: 1-8. https://doi.org/10.1145/602259.602261.

[67] SOISALON-SOININEN E, YLÖNEN T. Partial strictness in two-phase locking[C/OL]//ICDT: volume 893. 1995: 139-147. https://doi.org/10.1007/3-540-58907-4_12.

[68] GAWLICK D, KINKADE D. Varieties of concurrency control in IMS/VS fast path [J]. IEEE Database Eng. Bull., 1985, 8(2):3-10.

[69] CHEN S. Flashlogging: exploiting flash devices for synchronous logging performance[C/OL]//SIGMOD. ACM, 2009: 73-86. https://doi.org/10.1145/1559845.1559855.

[70] LEE S, MOON B, PARK C, et al. A case for flash memory ssd in enterprise database applications[C/OL]//SIGMOD. ACM, 2008: 1075-1086. https://doi.org/10.1145/1376616.1376723.

[71] JOHNSON R, PANDIS I, STOICA R, et al. Scalability of write-ahead logging on multicore and multisocket hardware[J]. VLDB J., 2012, 21(2):239-263.

[72] JOHNSON R, PANDIS I, HARDAVELLAS N, et al. Shore-mt: a scalable storage manager for the multicore era[C/OL]//EDBT: volume 360. 2009: 24-35. https://doi.org/10.1145/1516360.1516365.

[73] HENDLER D, SHAVIT N, YERUSHALMI L. A scalable lock-free stack algorithm [J]. JPDC, 2010, 70(1):1-12.

[74] MOIR M, NUSSBAUM D, SHALEV O, et al. Using elimination to implement scalable and lock-free FIFO queues[C/OL]//SPAA. 2005: 253-262. https://doi.org/10.1145/1073970.1074013.

[75] CORPORATION O. Mysql 8.0:optimizing innodb redo logging[EB/OL]. 2019. https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-logging.html.

[76] MALVIYA N, WEISBERG A, MADDEN S, et al. Rethinking main memory OLTP recovery[C/OL]//ICDE. 2014: 604-615. https://doi.org/10.1109/ICDE.2014.6816685.

[77] THOMAS R H. A majority consensus approach to concurrency control for multiple copy databases[J]. ACM Trans. Database Syst., 1979, 4(2):180-209.

[78] WU Y, GUO W, CHAN C, et al. Fast failure recovery for main-memory dbmss on multicores[C/OL]//SIGMOD. ACM, 2017: 267-281. https://doi.org/10.1145/3035918.3064011.

[79] ARULRAJ J, PERRON M, PAVLO A. Write-behind logging[J]. PVLDB, 2016, 10(4):337-348.

[80] PAVLO A, ANGULO G, ARULRAJ J, et al. Self-driving database management systems[C/OL]//CIDR. www.cidrdb.org, 2017. http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf.

[81] KIMURA H. FOEDUS: OLTP engine for a thousand cores and NVRAM[C/OL]//SIGMOD. ACM, 2015: 691-706. https://doi.org/10.1145/2723372.2746480.

[82] GAO S, XU J, HÄRDER T, et al. Pcmlogging: Optimizing transaction logging and recovery performance with PCM[J]. TKDE, 2015, 27(12):3332-3346.

[83] FANG R, HSIAO H, HE B, et al. High performance database logging using storage class memory[C/OL]//ICDE. IEEE Computer Society, 2011: 1221-1231. https://doi.org/10.1109/ICDE.2011.5767918.

[84] CORPORATION O. Timesten in-memory database replication guide[EB/OL]. 2014. https://docs.oracle.com/cd/E11882_01/timesten.112/e21635/toc.htm.

[85] WIKIPEDIA. Infiniband[EB/OL]. 2019. https://en.wikipedia.org/wiki/InfiniBand.

[86] WIKIPEDIA. Remote direct memory access[EB/OL]. 2019. https://en.wikipedia.org/wiki/Remote_direct_memory_access.

[87] 朱阅案，周烜，张延松，周明，朱嘉，王珊. 多核处理器下事务型数据库系统优化技术综述[J]. 计算机学报, 2015, 38(09):1865-1879.

[88] COOPER B F, SILBERSTEIN A, TAM E, et al. Benchmarking cloud serving systems with YCSB[C/OL]//SoCC. ACM, 2010: 143-154. https://doi.org/10.1145/1807128.1807152.

[89] LAMPORT L. Time, clocks, and the ordering of events in a distributed system[J]. ACM, 1978, 21(7):558-565.

[90] CURINO C, ZHANG Y, JONES E P C, et al. Schism: a workload-driven approach to database replication and partitioning[J]. PVLDB, 2010, 3(1):48-57.

[91] ANDREEV K, RÄCKE H. Balanced graph partitioning[J]. Theory Comput. Syst., 2006, 39(6):929-939.

[92] LAB K. Metis package[EB/OL]. 2006. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[93] LARSON P, BLANAS S, DIACONU C, et al. High-performance concurrency control mechanisms for main-memory databases[J]. PVLDB, 2011, 5(4):298-309.

[94] WU Y, ARULRAJ J, LIN J, et al. An empirical evaluation of in-memory multi-version concurrency control[J]. PVLDB, 2017, 10(7):781-792.

[95] COUNCIL T P P. Tpc-c benchmark[EB/OL]. 1992. http://www.tpc.org/tpcc/.

[96] WU Y, GUO W, CHAN C, et al. Fast failure recovery for main-memory dbmss on multicores[C/OL]//SIGMOD. ACM, 2017: 267-281. https://doi.org/10.1145/3035918.3064011.

[97] YAO C, AGRAWAL D, CHEN G, et al. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases[C/OL]//SIGMOD. ACM, 2016: 1119-1134. https://doi.org/10.1145/2882903.2915208.

[98] YU X, BEZERRA G, PAVLO A, et al. Staring into the abyss: An evaluation of concurrency control with one thousand cores[J]. PVLDB, 2014, 8(3):209-220.

[99] COMPANY A T. Agigaram non-volatile dimms[EB/OL]. 2017. http://agigatech.com/products/agigaram-nvdimms/.

[100] COMMUNITY I. nmon for linux[EB/OL]. 2009. http://nmon.sourceforge.net/pmwiki.php.

[101] 潘巍，李战怀，杜洪涛，周陈超. 新型非易失存储环境下事务型数据管理技术研究[J]. 软件学报, 2017, 28(1):59-83.

[102] DRAGOJEVIC A, NARAYANAN D, CASTRO M, et al. Farm: Fast remote memory[C/OL]//NSDI. USENIX Association, 2014: 401-414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87.

[103] TIAN B, HUANG J, MOZAFARI B, et al. Contention-aware lock scheduling for transactional databases[J]. PVLDB, 2018, 11(5):648-662.

[104] LEVANDOSKI J J, LARSON P, STOICA R. Identifying hot and cold data in main-memory databases[C/OL]//ICDE. IEEE Computer Society, 2013: 26-37. https://doi.org/10.1109/ICDE.2013.6544811.

致　　谢

三月桃良，四月秀蔓。倚窗而立，凝眸间，总有些情愫简约明媚，总有些记忆，悄然而至。六度寒暑春秋，回首间，几许彷徨迷惘，几许失落忧伤，都渐行渐远成笑靥中的逝水沉香。余下的，唯有数不尽的感动与欢笑，道不尽的感激与珍重。

回望屋内，实验桌上的项目计划书、论文草稿，层层叠叠地摆放着。尊师的培育与指导，犹如昨日，历历在目。钱卫宁老师，以其渊博的学识，导我于曲路，示我以通途。传授的行事准则，字字珠玑。每次交流与探讨，都让我不胜欢喜，且信心满满。回想起，深夜发来的论文修改意见，以及为毕业工作的奔波，感激之情，真的无以言表。胡卉芃老师，以其严谨的学术态度，指导我论文写作。无数次的挑灯夜战，无数次的推翻重写，只为了更精益求精，增加中稿的几率。每想起，为我毕业担忧的神情，以及走廊尽头来回的踱步声，心里便不胜感激。毛嘉莉老师，如邻家姐姐一般，关心我的生活和学习。懈怠时，有她严厉的批评；疲惫时，有她温馨的鼓励。其间，遇此良友，夫复何求。周烜老师，以其深厚的学术功底，开阔的思维方式，给予我论文指导与帮助。周傲英老师，以其宽阔的眼界，创建了优秀的科研环境，提供了大量学习和工作机会。此期间，还有幸得到蔡鹏老师、张召老师、宫学庆老师、周敏奇老师、张蓉老师、金澈清老师、高明老师、翁楚良老师、徐辰老师、余海萍老师在日常学习和工作中的帮助与指导。尊师们的悉悉教导之情，吾将感念于心，念兹在兹。

环顾四周，是这些同学们陪伴我走过了两千一百九十个日夜。朱涛、郭进伟同学无私地指导我的科研工作，分担我的压力，包容我的情绪。是他们，在最黑暗、最难熬的日子里给予我莫大的帮助与支持。顾伶、李永峰、翁海星、张晨东、樊秋实、庞天泽和裴欧亚同学陪伴我度过了交通银行的实习阶段。是他们，让初到上海的我不再感到孤独与无助。肖冰、王继欣、赵春扬、黄建伟、张小磊、阳文灿、王伟成和张涛同学让我拥有了欢乐的实验室时光。是他们，给了我无微不至的关怀与照顾。最后，还有朱燕超、段惠超、王嘉豪、王冬慧、黄晨晨、李宇明、魏星、贺小龙、张二宝、梁爽、蒋栋磊、卫孝贤、张融荣、潘雨辰、隆飞等同学陪伴我度过了丰富多彩的校园时光。是他们，在日常工作和生活中给予我很多关心与帮助。六年同窗之情，吾将铭记于心。

仰望天空，西边是我的故里，那里有我最最亲爱的家人们。六年的光阴，他们白了头发，皱了眼角。他们的信任与支持，一直是那最坚强的后盾；他们的怀抱，永远是那最温暖的港湾。感恩！

六年漫长而又短暂的博士生涯即将结束了，虽感叹时光之易逝，惋惜韶华之难追，却唯有且行且珍惜。世间的每一次相遇都是久别重逢。这一次，我们花了二十年，下一次呢？那时的我们是否都已扮成了大人模样。虽不知此后身在何方，惟愿我们都能不忘初心，素心向暖，浅笑安然。

周 欢

二零一九年五月

攻读博士学位期间发表的学术论文

[1] **Huan Zhou**, Jinwei Guo, Ouya Pei, Weining Qian, Xuan Zhou, Aoying Zhou, Plover: Parallel In-Memory Database Logging on Scalable Storage Devices. In Proceedings of *APWeb-WAIM Joint Conference on Web and Big Data* (2) 2018: 35-43 2018.

[2] **Huan Zhou**, Huiqi Hu, Tao Zhu, Weining Qian, Aoying Zhou, Yukun He, Laser: Load-Adaptive Group Commit in Lock-Free Transaction Logging. In Proceedings of *APWeb-WAIM Joint Conference on Web and Big Data* (1) 2017: 320-328.

[3] **Huan Zhou**, Jinwei Guo, Huiqi Hu, Weining Qian, Xuan Zhou, Aoying Zhou, Plover: Parallel Logging for Replication Systems. *Frontiers of Computer Science* (2019). https://doi.org/10.1007/s11704-019-8314-y. (Accepted).

[4] **Huan Zhou**, Jinwei Guo, Huiqi Hu, Weining Qian, Xuan Zhou, Aoying Zhou: Guaranteeing Recoverability via Partially Constrained Transaction Logs. CoRR abs/1901.06491 (2019)

[5] Tao Zhu, Huiqi Hu, Weining Qian, **Huan Zhou**, Aoying Zhou. Fault-Tolerant Precise Data Access on Distributed Log-Structured Merge-Tree. *Frontiers of Computer Science* (2018). https://doi.org/10.1007/s11704-018-7198-6. (Accepted).

[6] 朱涛, 郭进伟, **周欢**, 周烜, 周傲英. 分布式数据库中一致性与可用性的关系 [J]. 软件学报, 2018, 29(1): 131-149.

[7] Peng Cai, Jinwei Guo, **Huan Zhou**, Weining Qian, Aoying Zhou, Fast Raft Replication for Transactional Database Systems over Unreliable Networks. In Proceedings of *24rd International Conference on Database Systems for Advanced Applications*, 2019.

[8] Jixin Wang, Jinwei Guo, **Huan Zhou**, Peng Cai, Weining Qian, Adaptive Transaction Scheduling for Highly Contended Workloads. In Proceedings of *24rd International Conference on Database Systems for Advanced Applications*, 2019.