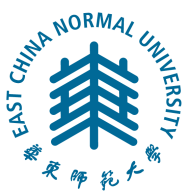
2018 届博士学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：52121500008



East China Normal University

博 士 学 位 论 文

DOCTORAL DISSERTATION

论文题目：分布式存储上的高性能事务处理

院　　系：	计算机科学与软件工程学院
专 业 名 称：	软件工程
研 究 方 向：	数据库系统
指 导 教 师：	王晓玲 教授
学位申请人：	朱涛

2018 年 05 月 18 日

Dissertation for doctor degree in 2018 University Code: 10269

Student ID: 52121500008

EAST CHINA NORMAL UNIVERSITY

High Performance Transaction Processing on Distributed Storage

Department:	School of Computer Science and Software Engineering
Major:	Software Engineering
Research direction:	Database System
Supervisor:	Prof. Xiaoling Wang
Candidate:	Tao Zhu

2018.05.18

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《分布式存储上的高性能事务处理》，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：朱涛　　日期：2018年 5 月 25日

华东师范大学学位论文著作权使用声明

《分布式存储上的高性能事务处理》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：王晓玲　　本人签名：朱涛

2018 年 5 月 25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）。

朱涛 博士学位论文答辩委员会成员名单

姓名	职称	单位	备注
李战怀	教授	西北工业大学	**主席**
薛向阳	教授	复旦大学	
王国胤	教授	重庆邮电大学	
王长波	教授	华东师范大学	
贺樑	教授	华东师范大学	

摘　要

关系型数据库系统出现在上世纪 70 年代。其中，事务处理是数据库系统最为关键的特性之一。事务大大简化了上层应用管理数据的复杂度。因此，事务型数据库系统被广泛地运用在银行、证券、电子商务等领域的关键应用中。近些年，互联网的快速发展进一步促进了应用规模的扩展，同时也导致了后台数据库系统需要支撑大吞吐量的事务处理以及海量数据的存储。然而，传统磁盘数据库系统的性能很大程度上受限于低效磁盘读写能力。这导致了它们很难满足新业务日益增长的性能需求。

得益于软硬件技术的发展，许多新型数据库系统被设计用于满足互联网业务的数据管理需求。从硬件的角度而言，内存和多核 CPU 的研发与制造技术均有了长足的发展。这些硬件的发展促成了内存数据库的研究和使用。相对于磁盘数据库，内存数据库在事务处理性能上有数量级的提升。从软件的角度而言，Google 首先成功研发和部署了大型的分布式数据库系统。随后，市场上出现了大量的 NoSQL 分布式数据库系统。这些系统能够利用大量廉价的普通服务器，提供极佳的数据存储和读写能力。然而，这些系统均存在一些重要的限制。例如，内存数据库要求业务的数据量不能超过单点内存容量，并且上层应用需要采用存储过程的方式执行事务。NoSQL 系统为了实现良好的扩展性，弱化或不支持关系模型、SQL 接口以及事务管理。

为了支撑业务在事务处理和数据存储方面的需求，本文给出了一个新型的分布式数据库系统 Cedar。它融合了内存事务处理和分布式存储两方面的优势。该系统使用一个两层的分布式日志合并树组织数据。它解耦了系统的数据存储、事务管理和查询处理三个模块。其中，一个分布式存储引擎会保存数据库的历史快照；一个内存事务引擎会管理新提交的增量数据；内存中的数据会周期性地合并到分布式存储中；最后，一个可扩展的分布式查询层会负责 SQL 处理、事务逻辑执行等。本文主要设计和优化了该系统上的事务处理模块，主要贡献可以总结如下：

1. **分布式日志合并树上的事务管理和通信优化。**Cedar 的架构近似于一颗分布式日志合并树。首先，本文为 Cedar 设计了一种新型的并发控制策略。它保证了内存事务引擎高效的事务管理性能。基于事务管理模块，我们设计了 Cedar 的数据合并算法。它保证数据合并操作不会阻塞或中断并发的事务请求。最后，针对 Cedar 的数据组织特点，本文设计和实现了数据缓存、事务编译等机制来有效地减少事务执行过程中节点间的网络通信。

2. **内存事务引擎上的交互式事务处理优化。**已有的内存事务引擎通常假设事务是作为存储过程执行的。这限制了应用操作数据库的灵活性。因此，本文考虑了如何优化交互式事务的处理。该问题面临着更高的复杂度和难度。由于事务执行需要在客户端和服务器之间进行多次网络交互，这产生了大量的 CPU 阻塞和上下文切换的代价。本文设计一种基于协程的执行模型来更好地处理网络读写或事务冲突导致的阻塞。此外，网络交互的延迟大大增加了事务的执行时间。这会引起更加频繁的事务冲突。本文设计了一个轻量级、多核可扩展的锁管理器来高效地识别和调度冲突的操作。

3. **分布式日志合并树的只读请求优化。**由于分布式日志合并树的结构特点决定了其在处理只读请求方面的效率不高，本文提出了一种精准数据访问优化方法。该算法主要包括一种可快速更新和同步的布隆过滤器，以及一种分布式租约管理机制。前者能以较低的代价在多个节点上同步数据的存在性，并过滤无效的远程访问。后者确保过滤部分远程数据访问不会弱化读取结果的一致性。通过减少无效的节点间通信，该算法不仅能够降低数据读取的延迟，还能够大大减少处理只读操作产生的资源开销，从而提升系统整体的吞吐率。

综上所述，我们在一种新型的分布式数据库系统架构上研究了如何实现高性能的事务处理。首先，针对分布式日志合并树存在的结构性问题，本文从并发控制、数据访问等角度优化了数据合并以及网络通信对事务吞吐率的影响；之后，由于内存事务引擎在处理交互式负载方面存在较大的不足，本文设计了新型的执行引擎和锁管理器来降低事务执行的 CPU 开销；最后，考虑到应用负载中同样存在大量的只读请求，本文设计了精准数据访问算法来过滤无效的网络通信。未来的研究方向包括：优化数据合并的效率、设计可扩展的事务处理机制以及提升系统在查询分析方面的能力。

关键词: 事务处理，分布式系统，内存数据库，日志合并树，并发控制

ABSTRACT

Relational database system was developed in 1970s. And one of its most important feature is the transaction processing, which greatly simplifies the complexity of the application development. Hence, transactional database system has been widely adopted by many areas, such as banking, stocking and online shopping. In recent years, the fast development of Internet contributes to the expansion of applications, and requires back-end database systems to support high-throughput transaction processing and massive data storage. However, the performance of disk-based DBMSs is largely limited by the slow disk I/O. They are hard to meet the increasing performance requirements of new applications.

Benefited from the software and hardware development, many new database systems are designed to satisfy the data management requirements from Internet businesses. From the hardware perspective, the memory and multi-core CPU have gained great development, which contributes to the implementation of in-memory database system. Compared with the disk-based one, in-memory DBMSs have an order of magnitude increase in transaction throughput. From a software perspective, Google first successfully developed and deployed several large distributed database systems. Subsequently, many NoSQL systems have come into the market. They can utilize a number of inexpensive, commodity servers to provide excellent data storage and read/write capabilities. However, these systems have some non-negligible limitations. For example, in-memory systems require the database to be fit in the main memory of a single server, and applications should invoke transactions as stored procedures. NoSQL systems sacrifice some important features (e.g. relational model, SQL, and transaction) for good scalability.

In order to satisfy the transaction processing and data storage requirements, this work gives Cedar, a new distributed database system combining the advantages of both in-memory transaction processing and distributed storage. It uses a two-layer distributed log-structured merge-tree (LSM-Tree) to organize all data. It decouples three modules of the system: the data storage, the transaction management, and the query processing. Overall, a distributed storage engine manages a consistent database snapshot; an in-memory transaction engine manages the rest newly committed data; in-memory data is periodically merged into the distributed storage; and lastly, an extensible distributed query layer is responsible for SQL processing, transaction logic execution, and so on. This work focuses on improving the transaction performance of Cedar, and main contributions are:

1. **Transaction management and communication optimization on the distributed LSM-Tree.** The architecture of Cedar is analogous to a two-layer distributed LSM-Tree. Firstly, a new concurrency control method is proposed to enable high-throughput transaction processing on Cedar. Secondly, based on the transaction management module, a data

compaction algorithm is designed to merge in-memory data back to the distributed storage engine, without blocking or interrupting any on-going transactions. Finally, based on the characteristics of data storage, we design caching, transaction compilation, and other means to effectively reduce inter-node communications during transaction execution.

2. **Optimizing interactive transaction processing on the in-memory transaction engine.** Existing in-memory systems assume that transactions are executed as stored procedures, limiting the flexibility of applications to operate the database. Hence, we consider how to improve the performance of interactive transaction processing, which faces more complexities and difficulties. Since each transaction invokes multiple network interactions between the client and the server, it generates much CPU blocking and context switching overhead. We propose a coroutine-based execution model to better handle blocking resulted from network I/O or transaction conflicts. In addition, the latency of network interaction also increases the lasting time of each transaction, resulting in more conflicts. We design a lightweight, multi-core scalable lock manager to efficiently identify and schedule conflicting operations.

3. **Optimizing the read-only request processing on the distributed LSM-Tree.** The distributed LSM-Tree is inefficient in servicing read-only requests due to its characteristics. We propose a precise data access method. It mainly includes a Bloom filter that can be efficiently updated and synchronized, and a lease-based synchronization mechanism. The former can efficiently synchronize data existence among multiple nodes, and filter most useless remote access. The latter ensures that such filtering does not weaken the consistency of the result. By reducing the number of useless inter-node communications, the algorithm not only decreases latencies of read requests, but also significantly reduce the resource used by read processing, and increases the overall system throughput.

In summary, we investigate how to achieve high-throughput transaction processing based on a new distributed architecture. Firstly, realizing major drawbacks of distributed log-structured merged-tree, this work designs new concurrency control schema and data access optimizations so that data compaction and network communications do not limit throughput. Secondly, since existing in-memory transaction engine is not well designed for interactive workloads, this work designs a new execution engine and a lock manager to reduce the CPU overhead of transaction execution. Lastly, considering that application workloads also contain lots of read-only requests, we designs a precise data access algorithm to reduce useless network communications. Future directions include: improving the efficiency of the data compaction operation, designing scalable transaction processing schema and promoting the performance of analytic query processing.

Keywords: *Transaction Processing, Distributed System, In-Memory Database System, Log-Structured Merge-Tree, Concurrency Control*

目录

插图

表格

第一章　引言

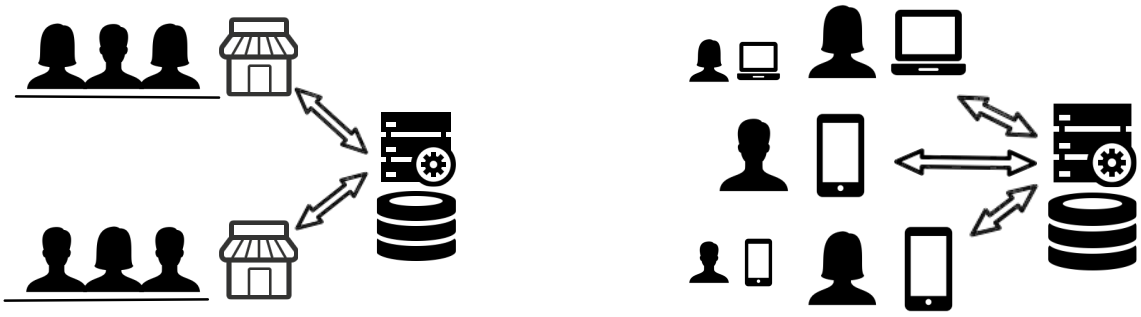
关系型数据库系统出现在上世纪 70 年代。其中，事务处理是数据库系统最为关键的特性之一。当使用事务访问数据库时，应用能够确保以下重要性质：原子性（**A**tomic），一致性（**C**onsistency），隔离性（**I**solation）和持久性（**D**urablity） [1, 2]。这四个性质通常简称为事务的 ACID 特性，它们对上层业务而言是极其重要的。例如，在银行业务中，转账包含了两个操作：扣除某个账户的部分余额，然后将扣除的金钱存入另一个账户中。原子性保证了扣钱和存钱两个操作或者均发生，或者均不发生。一致性保证了扣钱操作扣除的数字不会超过账户余额。隔离性保证了多个并发执行的转账操作不会互相影响。逻辑上，数据库是按某个次序依次完成这些转账请求的。持久性保证成功转账操作对数据库的修改不会因为系统故障或其他原因丢失。这些性质对许多应用（如账务，订单）而言都是必不可少的。在过去的数十年中，事务型数据库系统获得了巨大的成功。它们被广泛应用在银行、证券、电子商务等领域的关键应用中。然而，随着互联网的快速发展，应用对数据库功能和性能的要求不断提高。传统的事务型数据库系统开始显现出诸多不足。

1.1　研究背景

接下来，这一节将分别从应用背景和软硬件发展背景两个方面分析目前事务型数据库系统研究面临的机会与挑战。

1.1.1　应用背景

近些年来，互联网的快速发展对世界带来了巨大的改变。在线上，用户能够通过互联网获得包括在线购物、远程聊天等服务。在线下，用户还能够利用互联网租用共享单车、实现移动支付等。互联网的便捷性带来了用户数量的激增。根据中国互联网络信息中心给出的数据，截止 2017 年底，中国网络用户的数量已经增长至 7.72 亿。该数值在五年内增长了 3.52 亿，接近翻倍 [3]。庞大的用户规模对互联网

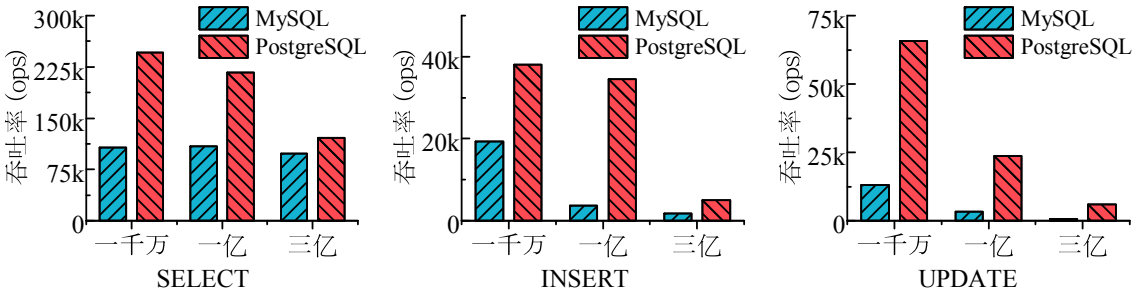


(a) 通过代理间接访问　　(b) 通过互联网直接访问

图 1.1: 不同的数据库访问模式

应用的处理能力提出了更高的要求。同时，这也对数据库系统的事务处理和数据存储能力提出了更大的挑战。

事务处理需求的增长. 首先，互联网应用要求事务型数据库能支持更高的并发度和吞吐率。这主要由两方面的原因导致的。一方面是用户数量的激增；另一方面是用户访问数据库的方式发生了改变。图 1.1 对比了两种不同的数据库使用方式。在过去，终端用户需要通过各类线下的代理访问数据库。例如，在银行业务中，客户是通过柜台或者 ATM 机操作数据库中保存的账户信息。由于线下代理的数量相对而言是有限的，后台数据库并不需要处理很大的并发访问。在互联网时代，终端用户可以直接通过线上应用访问数据库。例如，客户可以使用个人电脑访问银行官网，操作自己的账户。因此，数据库系统需要应对的并发访问量大大的提升了。一个典型的例子是在线火车票销售平台。2010 年 1 月 30 日，12306 网站正式上线运行并推出线上购票服务。用户可以在该网站上直接购买火车票。在此之前，用户需要到各个线下服务站点购买火车票。售票系统的负载强度主要取决于服务站点的数量。即便同一时刻存在大量的购票用户，他们也会在各个站点排队等待购票。然而，线上购票服务的推出改变了这一情形。大量用户可以通过各类终端（手机、电脑等）同时通过 12306 网站查询余票信息并抢购车票。这大大增加了购票系统的访问压力，也对后台数据库的事务处理性能提出了更高的要求。据报道，2018 年 1 月 3 日至 2 月 1 日期间，该网站日均售出车票为 859.2 万张，高峰期间，全天完成的余票查询为 720 亿次，发售车票 1028.7 万张 [4]。传统的集中式数据库很难处理这种级



(a) 单行 SELECT (b) 单行 INSERT (c) 单行 UPDATE

图 1.2: MySQL 和 PostgreSQL 管理不同规模数据库的性能

别的负载。因此，开发人员不得不在业务层进行许多优化。例如，部署缓存层处理只读请求；构建队列使并发请求排队访问数据库；通过分库分表部署多个数据库实例等。

数据存储需求的增长. 此外，互联网的繁荣也催生了大量数据的生成。一些在线购物平台每天需要完成千万笔订单，日均生成数据量超过 50TB [1] [5]。互联网生成数据的速率还在不断增加。根据一家国际公司（IDC）的最新报告，2016 年全球生成的数据量约为 16 ZB，而 2025 年这一数字将增长 10 倍，达到 163 ZB [6]。快速增长的数据规模要求数据库系统能够提供更大的存储容量。然而，传统的集中式数据库已经很难满足数据存储的需求。图 1.2 中展现了 MySQL 5.6 和 PostgreSQL 9.3.0 [2]在管理不同规模数据库时的读写性能。该实验调整了数据库中记录的数量 [3]并测试了两个系统的单行读写吞吐率。可以看到，在记录数量从一千万增加至三亿过程中，两个系统的性能均持续下降。这主要是因为单台服务器仅能提供有限的磁盘读写能力。数据量增多后，完成单次读写产生的平均磁盘 I/O 会增加。所以单机数据库系统越来越难满足大型互联网应用的数据存取需求。此外，随着业务的快速发展，集中式数据库很难快速的扩容来满足上层应用日益增长的存储需求。

业务需求的快速增长要求数据库系统能够提供更高的事务处理和数据存储能力。一些研究工作 [7, 8] 利用分布式技术来设计新型的数据库系统，提升系统的数

[1] 1 TB = 1024 GB, 1 PB = 1024 TB, 1 EB = 1024 PB, 1 ZB = 1024 EB

[2] 部署在配置了 Intel E5 2620 CPU、128GB DDR3 内存以及 3.6TB 的机械硬盘的服务器上

[3] 平均记录长度为 200 字节

据存储能力。另外一些研究工作 [9–12] 致力于利用大容量内存和多核 CPU 来提升数据库系统的事务处理能力。

1.1.2 软硬件的发展

硬件技术在摩尔定律的影响下飞速发展。其中，内存容量的提升以及多核 CPU 的出现对数据库系统的设计产生了很大的影响。此外，许多互联网公司在过去二十年来逐步研发和部署了各类分布式数据库系统。这些系统的成功运用证实了利用大量廉价的服务器来构建一个高性能数据库系统是可行的。

内存技术的发展带来了其价格的快速下降。在过去 30 年内，内存的价格大约每 5 年下降一个数量级。当下，普通商用服务器配备的内存容量不断提升。大多数服务器拥有数十 GB 以上的内存空间。部分中高端的服务器甚至配备了 TB 级别的内存空间。一台拥有 32 物理核心，1TB 内存的服务器价格大约在 5 万美元 [13]。这使得设计以内存为主要存储介质的数据库系统成为了可能。相对于磁盘，内存在访问延迟和带宽方面有巨大的优势。在磁盘数据库中，低速的磁盘读写和高速的 CPU 计算之间的性能差异是系统的主要瓶颈来源。已有研究 [14] 证实：磁盘数据库仅有 10% 的 CPU 时间用于有效的数据处理，其他时间都耗费在了缓存管理，并发控制和写前日志等模块。而使用内存作为数据的主要存储介质可以直接消除这一瓶颈。一些学术原型系统 [15] 验证了：与基于磁盘的数据库系统相比，内存数据库系统的事务吞吐率达到了数量级的提升。

另一个重要的硬件发展趋势是多核 CPU 的出现。CPU 芯片的主频在 2003 年之前基本保持每 18 个月提升一倍的发展速度。但之后主频的提升陷入瓶颈。这是因为，当运行频率超过 4.5G Hz 后，芯片的功耗会急剧增加。而保持芯片正常工作的温度范围是在零下 40 摄氏度到 125 摄氏度之间。随着单位面积晶体管数量的增加，CPU 芯片的热积累问题越来越明显。因此，在 2003 年以后，芯片供应商开始通过横向增加物理核心的数量来提升 CPU 的计算能力。当前，一个中端的 Intel CPU 芯片大约配备了 12 至 32 个物理核心。在可见的未来，CPU 中包含的物理核

心数量会继续增长。若能够充分利用多核 CPU 提供的强大计算力，数据库系统的处理能力能够得到很大幅度地提升。

最后，在过去的二十年中，普通商用服务器和集中式数据库的搭配渐渐无法满足互联网业务的数据管理需求。使用高端服务器的成本通常又比较高，并且存在扩容难的问题。因此，许多互联网公司都逐步研发了分布式存储系统，例如 BigTable [7], Dynamo [16], Cassandra [17] 等。这些数据库系统通常也被称为 NoSQL 系统。它们能够横向地增加服务器来扩大系统的存储能力。它们能够部署在一组通用服务器上并获得较高的数据存取性能。但是，为了获得良好的扩展性，这些系统弱化了一些重要的数据库特性，例如：使用简化的键值存储模型、不支持 SQL、不支持事务等。由于缺少这些特性，使用这类系统会大大增加应用开发的复杂度 [8]。为了解决这些痛点，近些年来学术界和工业界重新开始设计支持事务处理的分布式关系型数据库，也称为 NewSQL [9]。

1.2 研究内容与挑战

尽管内存数据库和分布式存储系统均获得了较大的发展，但这些系统在实际使用中依然存在一些不足之处。例如，内存数据库系统要求存储的数据规模不能超过内存的容量；分布式 NoSQL 系统缺少或弱化了事务功能。针对这些问题，本文的主要目标是设计一个具有横向扩展能力，同时又能保证高性能事务处理的 NewSQL 系统。为了实现以上目标，本文仔细分析了现有的集中式和分布式数据库系统在事务处理方面的优点与缺点；然后，给出了一个新型的分布式数据库系统 Cedar。该系统基于普通的商用服务器和以太网络设备。这保证了系统部署的硬件成本较低。它将所有的数据组织在一个两层的日志合并树 [18] 上，并分别使用分布式存储引擎和内存事务引擎来管理不同层上的数据。这种分层的数据管理机制保证了系统的横向存储扩展能力以及高效的事务处理能力。基于该架构，本文从并发控制、系统恢复以及数据访问三个方面，探索如何提升系统在处理不同类型负载时的性能。系统设计和优化主要面临了以下挑战。

1. **分布式系统中，多节点的数据存储方式增加了事务管理的复杂性。**为了保证事务的 ACID 性质，系统依赖于分布式并发控制技术 [19] 或两阶段提交协议 [20]。但是这些技术通常代价比较高昂。以两阶段提交协议为例，它在事务提交阶段需要执行两次网络通信以及两次写盘操作。这会大大增加事务的提交延迟，进而导致更多的访问冲突，降低系统的并发度。此外，由于两阶段提价协议是一种阻塞协议。在事务提交阶段，如果相关的节点发生了故障，此时事务有可能陷入长时间的阻塞直到故障的节点恢复。因此，数据库系统需要更加高效的事务管理机制。

2. **分布式系统中，大量的节点间通信会严重制约事务处理性能的提升。**由于事务需要访问的数据可能存储在不同的节点上。因此，事务在执行过程中需要频繁的与其他节点进行交互。在两台通过千兆以太网链接的服务器间，完成一次网络通讯需要约 100 us 的时间。如果事务执行产生了较多的节点间通信，那么网络延迟会成为事务延迟的主要来源。一些系统会部署高端的 Infiniband 网络设备来降低网络通信延迟。但这些设备价格还比较高昂，会显著增加硬件成本。因此，分布式系统要尽可能减少节点间通信。

3. **在多核、大内存的硬件环境下，磁盘数据库中广泛使用的技术成为了新的性能瓶颈。**磁盘数据库的核心优化目标是解决低速磁盘访问和高速磁盘计算之间的矛盾。但内存数据库系统的主要优化目标变为了优化多核 CPU 的使用。以两阶段锁协议 [19] 为例，磁盘数据库使用了集中式锁表来实现两阶段锁。但是集中式锁表结构复杂。这导致单次加锁/解锁操作会消耗更多的 CPU 指令。此外，该结构中使用了较多的临界区。多个物理核心访问同一临界区时会产生较多的冲突。随着负载压力的提高，锁表会成为系统多核扩展性的瓶颈。因此，为了提升事务处理的性能，内存事务引擎需要设计和使用轻量级的、易于多核扩展的结构。

4. **现有的内存事务处理技术在调度和处理交互式的事务请求时存在较大的不**

足。 近些年，针对内存数据库的事务处理研究通常会假设事务是作为存储过程执行的。基于这种假设，事务在执行过程中不会因为网络或磁盘读写而阻塞。这大大简化了内存数据库的执行和调度模型。但实际上，大量的应用依然偏向于采用交互式的方式执行事务请求。客户端与服务器间交互产生的网络读写会频繁地阻塞事务的执行。这会对系统的吞吐率和延迟均带来负面的影响。因此，如何让内存事务引擎更加高效地处理交互式事务是一个重要且困难的问题。

5. **日志合并树优化了写入性能，但它在一定程度上牺牲了数据读取的性能。** 日志合并树采用分层的数据组织方式。数据是追加到顶层存储结构中，然后周期性地合并到下层的存储结构中。顶层结构通常位于在内存中，它保证了高效的数据写入。但是读取数据记录需要自顶向下访问多个结构。这带来了额外的开销。由于应用也会频繁的查询数据库中的数据，因此只读请求的处理性能对事务型数据库而言也是非常重要的。设计基于日志合并树的数据库系统需要考虑如何优化该结构上的数据访问。

1.3 主要贡献

本文给出了一种新型的共享一切的分布式数据库系统 Cedar。该架构使用一个两层的分布式日志合并树组织数据。基于该架构，Cedar 解耦了系统的数据存储、事务管理和查询处理三个模块，并采用不同的设计保证这三者的性能。其中，数据库的大部分记录主要持久化在一个分布式存储引擎中，它保证系统存储能力的横向扩展性；一个内存事务引擎会管理最新的数据写入，并提供高效的并发控制和系统恢复；最后，一个可扩展的分布式查询层会负责 SQL 处理、事务逻辑执行以及维持客户端链接等。本文从并发控制、数据访问以及节点间通信多个方面优化了系统在处理不同类型请求（存储过程式事务、交互式事务以及只读事务）时的性能。主要贡献可以概括为以下几点：

1. 设计和优化了分布式日志合并树上的并发控制和数据合并算法。它们共同保

证了系统能够有效地隔离并发事务请求以及数据合并操作，并保证正确的故障恢复。本文为 Cedar 设计了乐观并发控制 [21] 和多版本并发控制 [22] 相结合的策略。它结合了两者的特点，保证了内存事务引擎的高吞吐率的事务管理性能。在事务管理模块上，本文设计了一种无阻塞的数据合并算法。它保证数据合并操作不会阻塞或打断正在并发执行的事务。此外，Cedar 的分布式架构导致了事务执行过程会产生大量的网络通信。为此，针对不同类型的远程访问，本文设计了精细化的数据访问策略。其中，数据缓存机制用于减少对分布式存储引擎的远程访问；事务编译优化用于减少对内存事务引擎的远程访问。

2. 针对内存事务引擎上的交互式事务处理，本文仔细分析了现有技术的不足以及该问题的难点。一方面，在事务执行过程中，应用客户端会向数据库服务器多次发送 SQL 请求。这些网络交互会频繁地阻塞事务的执行、造成线程上下文的切换。本文设计了一种新型的事务执行模型。它能够有效地并发执行大量事务请求，并且尽可能减少事务并发和网络交互产生的 CPU 阻塞和线程上下文切换，保证计算资源的充分利用。另一方面，网络交互的延迟也会大大增加事务的持续时间，从而引起更多的事务冲突。本文设计了一种轻量级的锁管理器。它能以极低的 CPU 代价良好地调度交互式事务。并且，该结构采用了无锁设计（latch-free），具有良好的多核扩展性。

3. 针对分布式日志合并树上的只读请求处理，本文提出了一种精准数据访问算法。它能够有效地减少处理只读请求产生的远程通信。精准数据访问算法主要包括一种可快速更新和同步的布隆过滤器，以及一种分布式租约管理机制。前者能以较低的代价将一个写节点保存的记录集合信息同步到查询节点。查询节点访问该布隆过滤器来避免无效的远程数据访问。后者用于保证在查询节点上使用的布隆过滤器不会错误地过滤必要的远程访问。它确保使用精准数据访问算法优化的只读请求依然能够达到强一致性，即可线性化 [23]。通过减少无效的节点间通信，该算法不仅能够降低数据读取的延迟，还能够大

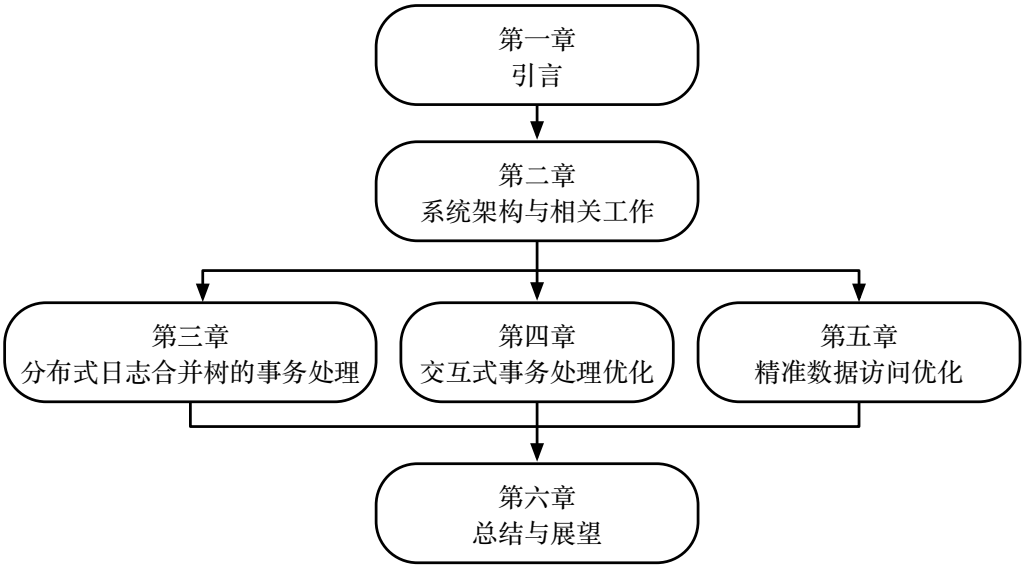


图 1.3: 本文的组织结构

大减少处理只读操作产生的资源开销，从而提升系统整体的吞吐率。

1.4 章节安排

本文结构组织如图 1.3 所示，后续章节内容的安排如下：

- 第二章首先介绍了事务型数据库系统的研究背景。然后，这一章给出了 Cedar 的整体架构以及数据组织方式。该架构实质上对应了一个两层的日志合并树 [18] 。它将数据库的历史快照保存在分布式存储引擎中，将最新写入的增量数据保存于内存事务引擎中。最后，与 Cedar 相关的系统主要包括集中式内存数据库系统，无共享（Shared-Nothing）的分布式数据库系统以及共享一切（Shared-Everything）的分布式数据库系统。最后，基于对比分析说明了 Cedar 的主要特别以及与现有系统的区别。

- 第三章主要探讨了在分布式日志合并树上，如何实现并发控制、系统恢复、数据合并以及优化远程数据访问。这一章讨论了正常情况下的事务管理，然后给出了数据合并算法，合并期间的事务管理机制，以及事务管理的正确性。为了保证事务执行的高效性，针对不同类型的远程数据访问，第三章提出了

多种的优化方法。最后，实验对比 Cedar 和不同类型分布式数据库系统的性能。

- 第四章主要介绍如何在内存事务引擎上高效地处理交互式事务。首先，这一章讨论了交互式事务和存储过程式事务的主要区别；然后，分析了交互式事务处理面临的主要难点。针对这些难点，本文给出了一种基于协程 [24] 的事务执行模型；基于该执行模型，设计了轻粒度、可扩展的两阶段锁管理器。最后，大量实验验证了方案的可行性和优势。

- 第五章分析了分布式日志合并树上数据访问的面临的困难。然后，这一章给出了一种基于布隆过滤器的远程数据访问过滤机制，并讨论了布隆过滤器的更新和同步问题。之后，本文讨论过滤远程数据访问可能引起的分布式一致性问题，并给出了一种基于租约的解决方案。最后，实验验证了精准数据访问算法在处理制只读请求方面的优势。

- 第六章总结全文内容，并说明本文工作的主要贡献和不足之处，最后讨论未来的研究方向。

第二章　系统架构与相关工作

2.1　背景

过去一些年来，NoSQL 系统得到了快速发展，并成功应用于大量互联网应用中。NoSQL 系统的成功体现了分布式系统在水平扩展性方面的巨大优势。然而，由于这些系统采用了分布式的数据库存储方式，使得它们很难良好的支持数据库系统的一大关键特点–事务处理。例如，BigTable [7] 仅支持单行的事务管理。而其他系统（如 Dynamo [16]）完全不支持事务。为了满足应用对事务处理功能的需求，业界提出了 NewSQL 系统的概念 [25]。NewSQL 系统致力于在分布式集群上搭建数据库系统，并提供高性能的事务处理。

然而，设计 NewSQL 系统面临着分布式事务处理的困难 [26] 。为了实现分布式事务，数据库系统需要在多个节点间进行必要的同步来解决潜在的事务冲突并保证 ACID 性质 [27]。这大大增加了系统设计的复杂度。近些年来有许多新的解决方案被提出。一些系统延用了*无共享的 (shared-nothing)* 的架构。例如，HStore [28] 和 VoltDB [29] 会根据业务负载的特征对数据库进行水平分区，并尽可能地保证绝大多数事务仅访问一个数据分区 [30]。在单个分区上，它们将事务排队执行以消除并发执行带来的管理开销，提升 CPU 的使用效率。Calvin [10] 在分布式键值数据库上构建了一个*确定性调度器*。确定性调度器在事务执行之前产生事务之间的调度计划。它有助于解耦系统的事务引擎和底层的存储引擎，同时优化了分布式事务的提交流程。另外一些系统延用了*共享一切 (shared-everything)* 的架构。Tell [31] 和 FaRM [32] 利用远程内存直接访问技术（Remote Direct Memroy Access, RDMA）设计了一个共享的分布式内存存储层 [33, 34]。集群中的节点能够互相共享其他节点上的内存空间。DrTM [35, 36] 同样使用了 RDMA 技术来优化节点间的数据交换，此外它还利用了硬件事务内存 [37]（Hardware Transactional Memory，HTM）来优化并发控制。

尽管这些新的技术在分布式事务处理方面获得了许多进步，它们依然有各种各样的不足和限制。例如，对 HStore/VoltDB 而言，它们希望应用和负载可以良好的分区。当负载中存在需要访问多个分区的分布式事务，系统的整体吞吐率会快速下降。Calvin 的确定性事务调度策略要求按照固定的周期在节点间同步和协商一组事务的调度结果，这导致了每个事务的处理延迟变高 [38]。此外，它要求在事务开始执行前提前知道它们的读/写记录集合。但这些信息在许多实际业务中是无法提前获得的。另外一方面，Tell，FaRM 和 DrTM 这些共享一切的分布式系统依赖于高端的硬件设备支持。但这些硬件设备还没有广泛普及，同时价格也比较昂贵。这意味着，如果不能假定业务负载和事务的一些特性，或者在没有特殊硬件支持的情况下，使用廉价商业服务器设计和实现通用的高性能的事务型数据库系统依然是一个极具挑战的问题。

除了分布式系统的快速发展外，近些年来单机内存数据库系统也得到了快速的发展。这些内存事务处理系统能够充分的利用多核 CPU 架构和大内存容量，在一个中端的服务器上实现高性能的事务处理。Silo [15] 和 Hekaton [13] 已经证实了一个单点的内存事务引擎能够达到每秒几十万（甚至上百万）的事务吞吐率。

然而，像 Silo 和 Hekaton 这样的单点内存数据库系统很难满足大数据时代应用的存储需求。当下，单个应用的数据库可能达到数十（甚至数百）TB 级别。一方面，单个节点的内存容量很难满足如此庞大的数据存储需求，另一方面，将所有的数据保存在内存中也会导致硬件的成本非常高昂。因此，系统仍然需要设计能够横向扩展的分布式数据存储机制。

基于以上背景，本文的目标是利用一组普通的商业服务器和以太网设备，设计一个分布式事务型数据库系统。它能够既拥有分布式系统*在数据存储方面的横向扩展性*，同时拥有内存数据库系统*在事务处理方面的高性能*。接下来，本章将首先介绍该系统的整体架构，然后将它与现有的其他系统进行对比分析。

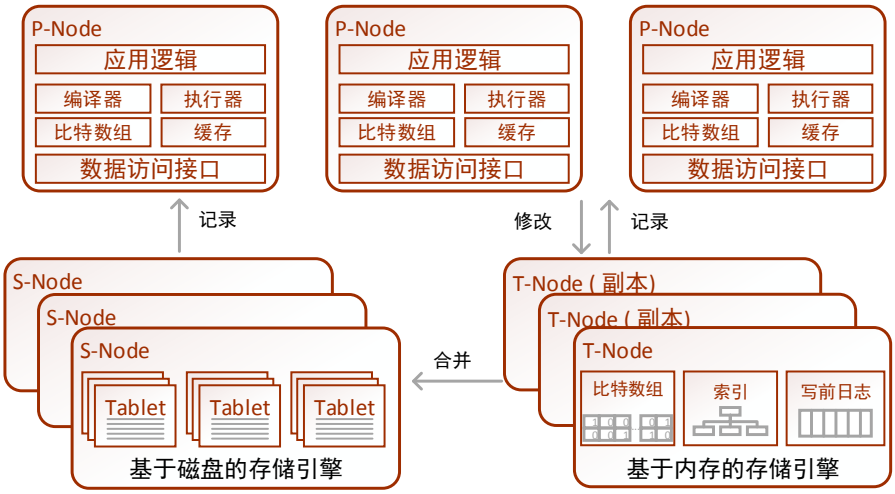


图 2.1: Cedar 的整体架构

2.2 系统架构概览

本节主要介绍 Cedar 的整体架构和数据组织方式，并分析该架构的特点。图 2.1给出了系统的整体架构。这里首先介绍本文在设计 Cedar 时的主要考虑，然后介绍它的整体架构、数据存储方式，以及该架构的主要特点。下一节将 Cedar 与现有的其他系统进行对比分析，并进一步阐述它的特点。

2.2.1 设计考虑

共享一切的架构. 许多分布式事务型数据库系统采用数据分区的方式来获得横向扩展性，例如：HStore [28] 和 VoltDB [29]。然而，数据分区存在一个很大的局限性：它很难处理需要访问多个数据分区的事务。通常，这些系统依赖于昂贵的分布式提交协议来保证多个节点上数据写入的原子性。所以，这种方法主要适合于负载提前确定并且可以分区的业务。

Cedar 选择了不同的方式来管理分布式存储的数据，并支持动态的、ad-hoc 的事务负载。Cedar 将数据库中已提交的数据分为两大部分：1）一个一致的数据库历史快照；以及 2）所有在该快照生成后新提交的增量修改。这里，一致的数据库历史快照是指：该快照包含了数据库在某时刻前所有已提交事务的写入内容，并且不包含任何未提交的数据。

数据库历史快照中包含的数据量通常非常的大，因此，它可以分布式地存储在多个节点（S-nodes ）的磁盘中（部分频繁访问的数据项会缓存在这些节点的内存中）。相对于该数据库快照，新提交的增量修改的数据量会小很多。这部分数据可以存储在一个事务节点（T-node ）的内存中。这种数据存储方式类似于两层的日志合并树 [18]。两者的主要区别在于 Cedar 的实现方式是分布式的。基于这种存储方式，Cedar 总是将事务执行产生的修改数据写入 T-node 的内存中。由于所有事务写入都是在单点上完成的。它避免了分布式事务的挑战，由此减少了调度分布式事务产生的昂贵开销。但是，这种存储方式使得事务访问记录的过程变得更加复杂了。为了读取一条记录的最新版本，一方面，事务需要获得记录在 S-node 上保存的快照版本；另一方面，事务需要要尝试获得记录在 T-node 上可能存在的最新修改；最后，事务将两部分数据合并才能获得最新的记录。

内存事务处理. 许多最新的研究 [13, 28] 已经充分证实：相对于基于磁盘的事务处理技术，内存事务引擎在吞吐率和延迟方面具有巨大的优势。在内存数据库中，索引、并发控制和日志恢复机制都需要很细致的设计，但同时它们也比以往基于磁盘的设计更加的高效。内存数据库的主要不足在于系统的存储容量有限。随着数据量不断地积累，当数据库的规模超过了内存的容量时，系统仅能在内存中保存部分数据，它需要提供相应的机制来管理超出容量的部分。否则，系统性能会因为频繁且昂贵的内存换入/ 换出而快速下降。

Cedar 将事务执行产生的最新写入数据保存在 T-node 的内存中。因此，该节点本质上是一个内存事务引擎。为了应对内存容量的限制，Cedar 会周期性地执行*数据合并*操作。当 T-node 内存中积累的写入数据越来越多时，数据合并操作会将 T-node 上所有已经提交的写入数据与 S-nodes 上存储当前数据库历史快照合并，最终在 S-nodes 上产生一份新的历史快照。在合并完成后，T-node 可以删除所有已经被合并的数据，并释放相应的内存空间。

Cedar 的数据合并操作不会影响或打断正在进行的事务。此外，事务处理和数据合并操作主要使用的硬件资源是不同的。数据合并主要消耗集群的网络带宽，而

事务处理主要消耗的是 T-node 的 CPU 资源。因此，Cedar 在数据合并期间依然能够保持良好的性能。

精细化的数据访问机制. 共享一切的系统架构允许每个计算节点（P-node ）直接访问和操作存储层上管理的数据。然而，计算节点在访问数据时需要与远程的存储节点交互。这会带来额外的网络通信代价。已有的研究显示可以使用支持 RDMA 的 InfiniBand 或 Myrinet 网络设备来改善远程数据访问的性能 [31]。但这些高端的网络设备通常都比较昂贵。一个典型的 InfiniBand 交换机的价格通常在 1 万美元以上。但如果使用普通的以太网络设备，节点间的通讯又会产生不可忽视的代价。

为了解决这个问题，Cedar 在查询处理层和数据存储层之间设计和实现了精细化的数据访问策略。首先，由 S-nodes 构成的分布式磁盘存储上保存的数据是不可变的。这使得计算节点可以很容易的缓存 S-node 上的数据以供重复读取。这不仅可以避免很多无效的网络通信，也可以减少 S-node 上的磁盘读取操作。此外，T-node 的内存存储引擎中保存的数据相对于整个数据库而言是很小的一部分。所以，对于不同事务的多数读取请求而言，它们并不需要访问 T-node 。Cedar 使用了一种基于比特数组的机制来过滤事务执行过程中对 T-node 的无效数据访问。最后，一个事务请求通常需要从 S-nodes 和 T-node 读取多条记录。与其一条条的读取这些记录，Cedar 会尽可能地通过一次网络通信读取多条记录。为了实现该功能，本文提出了一种事务编译机制。它会正确的调整一次事务请求中多个读取操作的执行顺序，并将连续的读取操作合并成为一次通信。下一章将具体介绍这些优化。

2.2.2 整体架构

图 2.1 给出了 Cedar 的整体架构。它主要由三部分组成：

1）T-node 是一个内存事务处理节点。它负责管理所有最新提交的更新数据（相对于 S-nodes 上保存的当前数据库历史快照），以及支持事务管理。T-node 上实现了一种新型的多版本乐观并发控制协议。为了保证数据的持久性，T-node 的事务引擎实现了写前日志机制（WAL），并在本地磁盘上保存事务的 redo 日志。

2）多个 S-nodes 共同管理一致的数据库历史快照，同时服务对该快照的读取请求。它们构成了一个分布式的磁盘存储引擎。T-node 的内存存储和所有 S-nodes 的磁盘存储共同构成了 Cedar 的存储层。T-node 上的最新已提交数据会通过数据合并操作周期性地合并到 S-nodes 上。当存储容量不足时，系统可以弹性的增加 S-node 来扩容。T-node 和 S-node 都使用了多副本技术来保证系统在部分组件故障后依然可以对外提供服务。

3）一组 P-node 负责查询处理以及事务逻辑执行。当接收到客户端发起的事务请求后，P-node 会从 S-nodes 和 T-node 拉取相关的数据；然后执行用户定义的业务逻辑；将修改内容写入 T-node 并提交事务；最后响应客户端。P-node 实例可以在集群中任意服务器上启动（例如，启动了 S-node 进程的机器）。所以，Cedar 可以弹性地增加 P-nodes 来获得更多的查询处理资源。针对 P-nodes 和存储层之间远程数据交互带来的代价，下一章详细给出了具体的通信优化算法。值得注意的是：Cedar 不会在 T-node 上启动 P-node 实例。这是为了节约 T-node 的硬件资源，并用于关键性的事务管理工作（例如，并发控制、写前日志等）。

2.2.3 数据存储

上一节主要介绍了 Cedar 的整体架构。本节将详细地介绍数据存储的机制，主要包括了不同类型的数据在系统中是如何组织和管理的，以及 T-node 上的数据是如何合并到 S-node 上的。

数据分布. 如前文所说的，Cedar 的数据存储分为两部分：最新已提交的数据，以及一致的数据库历史快照。历史快照由 S-node 管理，它被组织基于磁盘的结构 SSTable 中。而最新已提交的数据是由 T-node 管理，它被组织在基于内存的结构 Memtable 中。这里使用的结构 SSTable 和 Memtable 主要借鉴了 BigTable [7]。

SSTable 按照主键顺序组织数据库中的记录。同一个表的记录按照主键顺序排序后动态的划分为多个不相交的范围。每个范围的数据被顺序存储在一个称为 *tablet* 的结构中。每个 tablet 的大小默认为 256MB。

在 T-node 上，Memtable 中的记录均保存了多个版本（每个版本均是由已提交的事务产生）。记录的不同版本按照版本号排序后组织在链表中。当事务 t_x 提交了对记录 r 的修改后，Memtable 会为 r 生成一个新的版本。该版本仅保存记录 r 被事务 t_x 修改的列以及相应的取值。这主要是为了避免保存完整记录产生的额外内存开销。在 Memtable 上，记录（以及它们的链表）被组织在内存 B-tree。这保证了对记录的高效的范围查询。此外，事务类负载中通常包含了大量的点查询。因此，Memtable 还额外构建了哈希索引来支持对单条记录的快速访问。

Memtable 既处理事务的读取操作，也处理写入操作。而 SSTable 仅处理读取操作。当事务 t_x 尝试更新记录 r 时，如果该记录在 Memtable 上不存在，那么 t_x 会从 SSTable 中读取该记录，然后在 Memtable 上写入修改后的版本（仅包括主键和被更新的列）。如果事务执行的写入操作不需要读取记录当前的版本，那么 t_x 可以跳过对 SSTable 的读取，直接在 Memtable 中写入新版本。

事实上，S-node 上维护的 SSTable 和 T-node 上维护的 Memtable 共同构成了一个分布式日志合并树（Log-Structured Merge-Tree, LSM-Tree）[18]。其中，Memtable 代表了 LSM-Tree 的内存部分，而 SSTable 代表了它的磁盘部分。下文介绍的数据合并操作对应了 LSM-Tree 的合并过程。

数据合并. Memtable 的大小会随着事务写入数据的积累不断增长。T-node 会周期性地执行数据合并来减少内存的使用率。在数据合并过程中，T-node 会创建一个新的 Memtable 来接收后续事务的写入。而当前的 Memtable 会被隔离冻结，然后与 S-nodes 上的 SSTable 合并产生一个新的 SSTable。1. 如果某个记录在 Memtable 中存在，且在当前的 SSTable 中不存在，那么该记录会根据它的主键取值插入到 SSTable 对应的 tablet 中。2. 如果某个记录 r 在 Memtable 和 SSTable 中均存在，对 r 在 Memtable 中所有被更新的列，合并操作会在 SSTable 中保存这些列的最新版本。3. 如果记录 r 在 Memtable 中被标记为删除，同时它在 SSTable 中存在，那么合并操作会在对应的 tablet 中删除该记录。在完成 Memtable 和 SSTable 的数据合并后，部分 tablet 可能在插入新记录后实际大小超过 256MB，此时，它会分裂为

两个大小接近的新 tablet。部分 tablet 可能在删除已有记录后实际大小太低（低于 128MB ），此时，系统会尝试将它与范围相邻的 tablet 合并。

为了保证数据合并不会阻塞或者中断当前正在执行的事务，Cedar 在将 Memtable 中的数据合并到对应的 tablet 时使用写时复制（copy-on-write）技术来更新 tablet 的内容。不同于直接在当前的 tablet 结构上写入修改内容，写时复制会生成 tablet 的拷贝，然后在拷贝版本上写入数据。使用写时复制技术保证了数据合并对 tablet 的修改不会阻塞正常事务对该结构的读取。

使用写时复制会导致实际写入磁盘的数据增加。当 SSTable 不断变大后，更多的 tablets 会被创建。而每次合并时，Memtable 的大小是相对固定的。这会导致，多次合并后，每个 tablet 上的修改会不断减少。在最坏情况下，即便某个 tablet 上仅有一条记录被更新了，写时复制依然需要拷贝整个 tablet，并向磁盘写入 256MB 的数据。为了解决这个问题，每个 tablet 物理上会组织为多个数据块（block）。每个数据块大小默认为 64KB。写时复制会在数据块的级别进行。数据合并操作不会拷贝整个 tablet 结构，而仅拷贝并重写需要修改的数据块。最后，为了高效地定位数据块的物理存储位置，每个 tablet 保存了一个索引文件。该文件记录了它所有数据块的物理地址。在完成对数据块的合并后，tablet 会将它所有数据块的物理位置写入一个新的索引文件中。

Cedar 会周期性地执行数据合并。通常该操作发生在业务系统每天运行的低谷时刻。例如，每天上午 0：00-4：00。除了定期的合并外，Cedar 也可以在 T-node 内存不足时启动一次数据合并。例如，当服务器的内存使用率达到 80% 的时候。

多副本. Cedar 使用多副本技术来保证系统的高可用。对 SSTable 而言，每个 tablet 会保存至少三个副本。这些副本分布在不同的 S-nodes 上。多副本同时也可以改善多个 S-nodes 间的负载均衡。一个读取请求在访问某个 tablet 时，可以选择它的任意一个副本。Memtable 会在两个备 T-nodes 上保存副本。当任意事务在主 T-node 上提交并产生 redo 日志后，Cedar 会将 redo 日志同步到其他备 T-node 上。备节点通过回放 redo 日志来更新和同步本地的 Memtable 。日志同步是通过分布式

一致性协议 Raft [39] 实现的。

基于对 Cedar 架构的介绍，下一节将对比分析现有的事务型数据库系统的架构，并分析 Cedar 在事务处理，数据存储方面的特点。

2.3 相关工作

当前，事务型数据库系统主要有以下三种：集中式数据库、分布式无共享型数据库、分布式共享一切数据库。下文将重点介绍近些年来在这三个方向上的最新工作。其中，集中式数据库主要包含的是基于内存的数据库系统。此外，Cedar 采用的架构类似于一个两层的日志合并树。因此，本节也会介绍若干同样采用日志合并树组织数据的系统。

2.3.1 集中式内存数据库

内存数据库的快速发展主要得益于内存容量的快速提升，以及内存价格的下降。Hekaton [13, 40] 是微软设计的内存事务引擎，并已经集成到了 SQL Server 中。为了充分利用内存存储带来的优势，它在内部使用了大量无锁数据结构，多版本乐观的并发控制协议，并且将 SQL 和存储过程代码编译成了高效的机器码，以降低 CPU 在各个部分的固有开销。Silo [15] 是一个针对 NUMA 架构设计的单点内存数据库系统原型。它在 Hekaton 的基础上，通过减少系统中全局的冲突点，进一步优化了内存数据库在多核 CPU 上的扩展性。HyPer [12] 是针对混合负载（OLTP+OLAP）设计的内存数据库系统。它能够在虚拟内存中快速地生成数据库的只读快照。OLAP 类的查询请求在只读快照上执行，不会影响正常执行的 OLTP 事务请求。虽然，HyPer 存在一个新的版本 [41] 能够通过高性能网络设备将系统扩展到多个节点上来处理 OLAP 负载，但是它依然将所有的 OLTP 负载运行在一个节点上。这些单点的内存数据库系统的实际使用主要受限于内存的容量。对于数据量非常庞大的应用而言，单点的内存容量是不足以支撑整个数据库的存储需求的。针对该问题，现有研究提出将数据库中的记录根据它们被访问的频率划分为“冷数据”和“热数据”两类，并

将“冷数据”转移到磁盘中 [42, 43] 来减少内存资源的消耗。但这些机制并不能根本上的解决集中式内存数据库系统在存储容量方面的局限性。

除了内存数据库系统外，工业界也设计实现了一些新的磁盘数据库系统。MyRocks [44] 为 MySQL 设计了一种基于日志合并树的存储引擎。相对于 InnoDB（MySQL 上通常的存储引擎），它能够在数据库大小不断增加后提供更好的读写性能。Amazon Aurora [45] 在 MySQL 基础上设计了分布式的存储引擎。它也是提升了数据库系统在存储大量数据后的读写性能。这些系统均主要优化了传统数据库系统的存储能力。事务处理主要还是延用了磁盘数据库采用的设计。

集中式数据库的一大问题在于有限的存储能力。诸如 MyRocks 和 Amazon Aurora 主要优化了磁盘存储的读写性能。但是随着存储数据量的不断增加，这些系统依然面临着性能衰退的问题。内存数据库采用内存作为数据的主要存储介质，这极大地改变了数据库的并发控制，但内存数据库同样有存储容量方面的限制。

针对集中式数据库在数据存储容量方面的不足，Cedar 通过将整个数据库划分存储在两个结构中来解决这一问题。一个分布式存储引擎会负责存储某个一致的数据库历史快照。内存事务引擎仅保存快照生成后提交的增量写入。对于整个数据库而言，增量部分的数据量是很小的，它完全可以存储在单点的内存中。此外，周期性执行的数据合并会进一步释放数据存储消耗的内存空间。而保存快照的分布式存储是可扩展的。它可以通过横向增加服务器来获得更大的存储空间并保证数据读取的性能。

2.3.2 分布式无共享型数据库

无共享（Shared-Nothing）架构的分布式数据库通常采用数据分区（分库分表）的方式横向扩展系统的存储和处理能力。在这类系统中，数据库中的记录是按照表的某个列进行分区（哈希分区或者范围分区）后存储在不同的节点上的。良好的数据分区会使尽可能多的事务独立地运行在不同的数据分区上。如果上层业务和负载能够良好划分，那么这种架构能够实现极佳的性能和扩展性。但在处理需要访问

多个分区的分布式事务时，这种架构的性能会很差 [38] 。这主要是因为，分布式事务依赖于昂贵的分布式提交协议（两阶段提交 [46]）来保证原子性。除此之外，基于数据库分区进行水平扩展的方案存在另外一个缺点。它必须提前获得负载在数据访问方面的特征，以便于能够计算一个好的分区方式，减少潜在的分布式事务数量。HStore [9, 28] 以及它后续的商业版本 VoltDB [29] 通过将数据库分区后绑定到不同节点的不同 CPU 核心来实现横向扩展，同时在每个物理核心上使用单线程串行执行事务请求来尽可能的消除并发管理带来的 CPU 代价。Calvin [10] 在事务执行前确定它们的调度顺序。这种策略称之为确定性调度 [47]。由于所有事务的调度顺序是提前确定的，Calvin 可以利用这一点来优化两阶段提交协议，加速分布式事务的提交。Accordion [48] 和 E-Store [49] 为采用无共享架构的分布式系统设计了在线重分区算法。它们能够在线的改变分区布局来控制和减少分布式事务的数量。SAP HANA [11] 将时常被事务请求访问的“热数据”全部存储在一个配置较高的服务器中，以避免分布式事务的产生。其它的数据会分布式地保存在若干配置较低的服务器中。这些服务器会并行的处理 OLAP 业务来提升查询性能。Spanner [8, 50] 是 Google 设计的跨地域分布式数据库，来全局的管理分布在不同地域的数据。它的数据是经过分区后在不同地域存储了多个副本。它保证了数据库服务极高的可用性和强一致性。它依赖于全局有效的时钟服务来实现事务的并发控制，以及两阶段提交协议来保证分布式事务的原子性。为了处理分布式事务提交和时钟服务中存在的误差，Spanner 的事务提交延迟通常非常的高。这是它在事务处理方面比较大的一个缺陷。可以看到，无共享型架构的扩展性主要取决于负载中分布式事务的比例。现有的研究主要致力于解决昂贵的分布式事务对这类系统性能的影响。当前，这类架构的系统主要还是运用于能够较容易地进行分库分表的应用中。

Cedar 没有采用无共享的架构来实现横向扩展。一个集中式的内存事务引擎会保证 Cedar 高效的并发控制与系统恢复。这种架构的好处在于彻底避免了分布式事务管理对性能造成的巨大影响。但潜在的风险是内存事务引擎可能成为系统性能的瓶颈。事实上，已有的研究证明了：单点内存事务管理依然可以达到非常高的

吞吐率 [15]。并且，Cedar 还将事务的执行与它的并发控制和日志恢复解耦了。若干分布式查询计算节点 P-nodes 会负责事务的执行。这进一步减少了事务管理节点的压力。后续实验发现 Cedar 的事务吞吐率主要受到 P-nodes 数量的影响。

2.3.3 分布式共享一切型数据库

共享一切（Shared-Everything）是另外一种能够保证高性能和良好扩展性的分布式架构 [51, 52]。在该架构中，集群中的节点共享对数据库中、不同节点缓存中数据的访问权限。事务请求可以在任意节点上执行，同时可以访问任意其它节点上存储的数据。它不假设数据库以及业务负载是可以良好切分的。传统的共享一切型数据库系统有 IBM DB2 Data Sharing [53] 以及 Oracle RAC [51]。前者是为*并行系统综合体*大型机设计的。后者是为通过高端网络和存储阵列连接的普通商用服务器设计的。两者都依赖于特殊的硬件来支持节点间高效通信。此外，这两个系统都依赖于共享的全局锁管理器来同步不同节点对数据的并发访问，并保证事务的正确并发。Tell [31] 提出了一种基于普通商用服务器和远程内存直接访问技术（RDMA）的新型的共享一切架构。它依赖于 InfiniBand 的网络设备来实现节点间的内存快速访问 [33]，以及高效的数据交换。类似的，DrTM [35] 也实现了基于 RDMA 的共享一切型分布式数据库。RDMA 网络保证了节点间快速的交换数据和锁。它额外利用了硬件事务内存（HTM）来提升基于两阶段锁的并发控制效率。HANA SOE [54] 将数据库系统解耦成若干模块：一个用于查询处理的可扩展计算集群，一个用于提供容错性和持久性的分布式事务日志模块，以及一个用于保证事务并发的集中式节点。这些模块同样依赖于 RDMA 技术来保证高速的远程通信。可以看到，现有的共享一切型分布式系统均依赖于专用的或者高端的硬件设备来保证节点间高效的通信能力。但目前，提供 RDMA 功能的网络设备价格仍然很昂贵，供应商也比较少。例如，一台支持 43Tb/s 带宽以及 216 个端口的 InfiniBand 交换机价格大约是 6 万美元。另一方面，市面上支持 HTM 的服务器级 CPU 同样比较少。支持 HTM 的 Intel CPU 由于存在漏洞，供应商已经暂时把该功能关闭了，并尝试在后续

的版本中解决漏洞 [1] 。为了降低系统的部署成本，本文致力于在普通商用服务器和以太网络设备上设计高性能和可扩展的共享一切型数据库系统。

Cedar 采用的也是共享一切的分布式架构。查询计算层中的任意 P-nodes 均允许访问存储层中任意节点上的数据。但 Cedar 并不依赖于高端的网络设备来保证节点间高效的数据交换。Cedar 的数据分散存储在分布式存储引擎和内存事务引擎中。针对这两类数据，Cedar 使用了不同的远程数据访问优化策略来减少网络通信的开销。下一章将详细介绍这一部分的内容。

2.3.4 基于日志合并树的存储系统

日志合并树 [18] 根据数据写入系统的时序将整个数据库划分为若干个部分，并由多个独立的结构来维护每个部分的数据。其他研究工作 [55, 56] 也提出了类似的理念。在日志合并树中，每个结构都针对存储数据的访问特征以及存储介质的硬件特征进行了优化。许多 NoSQL 类的分布式存储系统都采用了日志合并树的设计理念。例如，BigTable [7] 和 Cassandra [17]。但现有的这些系统都没有事务处理的功能。不同于这些存储系统，Cedar 是面向联机事务处理的关系型数据库系统。

[1] https://www.realworldtech.com/haswell-tm/

第三章　分布式日志合并树的事务处理

上一节主要介绍了 Cedar 的设计背景、设计考虑、整体架构以及存储机制。本质上，Cedar 的架构是一个两层的分布式日志合并树。它使用一个节点 T-node 来提供内存事务处理，若干存储节点 S-nodes 来负载海量数据的存储和读取。在 Cedar 上，数据是首先写入 T-node 的内存上的。内存中的数据会定期的合并到由 S-nodes 组成的分布式磁盘存储中。系统可以弹性的增减 S-nodes 来横向调整存储能力。因此，这种架构保证了 Cedar 具有较高的写入性能，且能够良好地横向扩展系统的存储容量。最后，Cedar 使用一组 P-nodes 来负责与客户端通信，执行 SQL 和事务请求。它们为系统提供了可横向扩展的计算能力。尽管这种架构有以上的优势，但这不足以支撑 Cedar 成为一个高性能的联机事务处理系统。实现该目标依然面临着以下难点。

首先，系统采用了分布式日志合并树的结构来组织数据。这种方式有效的解耦了数据存储和事务管理模块，并降低了分布式环境下事务管理的复杂性。但是系统需要周期性地执行数据合并操作。该操作会大范围的改变系统中数据的组织方式。这意味着合并操作会与系统中运行的事务请求会产生读写冲突，从而影响事务的并发。然而，Cedar 的目标是作为一个联机事务处理系统使用。这需要为 Cedar 设计合适的并发控制，系统恢复和数据合并算法，并尽可能减少数据合并对事务处理产生的阻塞。

此外，Cedar 解耦了计算模块和数据存储模块。事务的执行是由一组 P-nodes 完成的，而存储是由 S-nodes 和 T-node 负责的。解耦存储和计算保证了系统能够横向扩展计算能力。但是，这会导致事务执行过程中产生大量的网络交互。低速网络交互和高速 CPU 计算及内存访问之间的矛盾会成为事务处理性能的主要瓶颈。因此，系统需要有效的优化机制来减少事务执行过程中产生的远程数据访问。

针对以上挑战，本章的主要贡献如下：1）提出了针对分布式日志合并树的并

发控制算法，它结合了多版本并发控制（MVCC）和乐观并发控制（OCC）的特点，保证了 T-node 高吞吐率的事务处理性能；2）在 Cedar 事务管理模块的基础上设计了相应的数据合并算法，它能够高效的将 T-node 上已提交的数据合并到 S-nodes 的磁盘中，同时不会阻塞或者打断正在进行的事务；3）针对分布式日志合并树上事务处理的特点，设计和实现了有效的通信优化策略（例如，数据缓存、事务编译等），进一步提升系统的事务处理性能；4）利用若干基准测试（TPC-C [57]，Smallbank 以及一个真实的电商负载）验证了 Cedar 相对于已有分布式系统在事务处理方面的优势。

本章的内容组织如下：第 3.1 节主要介绍了 Cedar 在 S-nodes，P-nodes，T-node 上实现事务处理、并发控制和系统恢复的细节；第 3.2 节重点介绍数据合并的执行流程，并给出在数据合并过程中事务处理的实现细节；第 3.3 节介绍若干重要的通信优化机制，保证 Cedar 高性能的事务处理；第 3.4 节给出了 Cedar 与一些其他系统的性能对比结果。对比的系统包括了 VoltDB [29]（代表了最新的 Shared-Nothing 系统）、MySQL Cluster [58]（代表了传统基于数据分区横向扩展的系统）、Tell [31]（代表了基于高端网络设备设计的 Shared-Everything 系统）。实验使用了若干不同的基准测试来验证各个系统的性能，包括常用的 TPC-C [57]、Smallbank 以及一个电商应用的真实负载。第 3.5 节总结本章的内容。

3.1 并发控制与系统恢复

Cedar 设计了一种新型的多版本乐观并发控制策略来实现事务的的快照隔离 [22]。尽管，该隔离级别下可能出现 写倾斜的异常，它并没有达到可串行化的要求，但它仍然为大量实际应用所接受。同时，它也是许多数据库系统支持的最高隔离级别，例如，PostgreSQL（9.1 版本前）、Tell [31] 以及 Oracle 11g [59]。本章主要考虑如何在 Cedar 上实现快照隔离。实现可串行化将作为一项未来工作。另一方面，为了保证数据持久性，以及在系统故障后正确的恢复数据库，T-node 在事务提交前会在持久化存储设备上写入 redo 日志 [60]（即，写前日志）。

3.1.1 并发控制

Cedar 结合 OCC 和 MVCC [21, 22] 实现了快照隔离的并发控制机制。在 Cedar 的数据存储机制下，每个记录在 Memtable 上维护了多个版本。给定一个事务 t_x 以及它的开始时刻的时间戳 rt_x。该时间戳可以是 t_x 执行第一次读取前的任意时刻。在 t_x 的执行过程中，它可以访问任意记录在 rt_x 时刻之前（包括 rt_x 时刻）创建的最新版本。因此，rt_x 也被视为事务 t_x 的**读取时间戳**（*read-timestamp*）。在 t_x 完成执行进入提交阶段后，它会获得一个**提交时间戳**（*commit-timestamp*），记为 ct_x。该时间戳应**大于**任意其他事务已经获得的读取时间戳或提交时间戳。然后，事务 t_x 需要验证：没有其他事务在 rt_x 和 ct_x 时刻之间修改了 t_x 需要修改的数据。否则，t_x 需要被回滚以避免出现**丢失更新**的异常 [22]。在验证通过后，事务将允许提交。它会为写入集合中的每个记录创建一个新版本。它们的版本号为 t_x 的提交时间戳 ct_x。

基于以上并发控制机制，Cedar 的数据存储可以按以下方式解释。假定最近的一次数据合并启动的时间戳为 t_{dc}。那么，所有已提交的事务可以分为两个部分：（第一部分）提交时间戳小于 t_{dc} 的事务；（第二部分）提交时间戳大于 t_{dc} 的事务。基于这样的划分，SSTable 中保存了由第一部分的所有事务提交的记录最新版本。而 Memtable 中保存了由第二部分的所有事务提交产生的所有新版本。

实现方式. T-node 使用一个全局单调递增的计数器来为事务分配时间戳。Cedar 将事务处理的过程分为以下几个阶段：处理阶段、验证阶段和写入/提交阶段。

处理阶段. 在处理阶段中，P-node 的一个工作线程会负责执行用户定义的事务执行逻辑。它会从 T-node 和 S-nodes 上读取事务 t_x 需要访问的数据记录。事务 t_x 在第一次与 T-node 通信时从该节点获得它的读取时间戳 rt_x。当 t_x 需要读取某条记录时，P-node 会读取该记录版本号不大于 rt_x 的最新版本。具体而言，P-node 首先尝试从 Memtable 读取最新的版本。如果没有获得合适的版本（即，时间戳不大于 rt_x 的版本），那么 P-node 会继续尝试从 SSTable 的相应 tablet 上读取该记录。当 t_x 需要写入某条记录的修改时，它会暂时将写入内容缓存在本地 P-node 的内存中。

当 t_x 完成业务逻辑的执行后，它将进入下一个阶段。此时，P-node 会将 t_x 的提交请求发送给 T-node 。提交请求中包含了 t_x 在执行阶段缓存的写入集合。T-node 会负责验证和提交该事务。

验证阶段. T-node 会完成事务验证阶段的工作。该阶段主要的任务是判断 t_x 与其他并发事务间是否存在写-写冲突。在验证阶段中，对写入集合 w_x 中的每条记录，t_x 会首先尝试获得它们的写锁。然后对任意 $r \in w_x$，t_x 需要检查记录 r 的提交链上是否存在版本号大于 rt_x 的新版本。如果 t_x 获得了写集合中所有记录的写锁，同时这些记录上也没有增加新的版本，那么 T-node 可以保证 t_x 与并发事务间不存在写-写冲突，即 t_x 可以提交。否则，为了避免丢失更新的异常，T-node 会回滚事务 t_x。综上，在完成验证阶段后，T-node 会决定是否提交事务 t_x。如果它决定回滚 t_x，那么 T-node 会向 P-node 返回事务回滚的决定。该 P-node 会终止 t_x 的本次执行。否则，事务将继续进入下一个阶段。

写入/提交阶段. 在该阶段中，事务 t_x 会为写集合中的所有记录在 Memtable 中创建一个新的版本。该事务临时将自己的事务 ID 填入这些新版本的版本号字段中。接下来，T-node 将全局时间戳计数器的取值原子性的加一，并获得修改后的取值作为事务 t_x 的提交时间戳。然后，事务 t_x 会将自己的提交时间戳填入所有它创建的记录版本中。最后，T-node 会释放所有 t_x 持有的数据锁。

正确性证明. 给定事务 t_x 及它的读取时间戳 rt_x 以及提交时间戳 ct_x，Cedar 保证 t_x 会读取一个一致的数据库快照，并且不会发生丢失更新的异常。证明如下：

一致性的快照读. 一方面，t_x 能够读取所有在 rt_x 时刻前提交的事务所写入的数据版本。这是因为，这些事务在 t_x 获得自己的读取时间戳 rt_x 之前，它们已经根据各自的写集合，创建了新的记录版本，并且获得了提交时间戳。另一方面，t_x 不会读取系统中其他事务产生的更新数据。这是因为所有在 rt_x 时刻后提交的事务势必会获得更大的提交时间戳。它们写入 Memtable 的记录版本的版本号必然大于 rt_x。所以，它们产生的更新不会被 t_x 读取。综上，t_x 总是读取一个一致的数据库快照，即由提交时间戳不大于 rt_x 的事务产生的数据库版本。

丢失更新的预防. 当出现以下情况时系统中存在丢失更新的异常：事务 t_x 的写入集合 w_x 中存在一条记录 $r \in w_x$，另外一个事务为记录 r 创建了一个新版本，该版本的版本号在 (rt_x, ct_x) 之间。假定创建该版本的事务为 t_y。那么，存在以下两种情况：

1）t_y 先于 t_x 获得记录 r 的数据锁。那么，t_x 仅可能在 t_y 提交后才能获得 r 的锁。此时，记录 r 的提交链上势必已经包含了 t_y 创建的新版本。所以，t_x 在验证阶段会发现 r 上存在一个新的版本，然后主动回滚自身。

2）t_y 在 t_x 获得记录 r 的数据锁后尝试加锁。在这种情况下，在 t_x 释放 r 上的数据锁前，t_y 无法获得这把锁，同样无法进一步获得提交时间戳。另一方面，t_x 在获得提交时间戳后才会释放锁。这意味着，t_x 先于 t_y 获得提交时间戳，即 $ct_x < ct_y$。这与假设相悖，即要求 t_y 创建的新版本的版本号在 (rt_x, ct_x) 之间。而记录 $r \in w_y$ 的新版本的版本号为 t_y 的提交时间戳 ct_y。该时间戳不在假设的范围内。

3.1.2 系统恢复

本节讨论当不同节点发生故障下线后，Cedar 是如何恰当的恢复数据库的状态的。

P-node 故障处理. 当一个 P-node 因为故障下线后，如果事务还没有发出提交请求，那么它依然在处理阶段中。这些事务可以被视为回滚了。系统不需要恢复任何数据。如果事务处于验证或者提交阶段，那么它们会由 T-node 正确的终止。这两个阶段的执行不需要和故障的 P-node 进行必不可少的网络通信。对于这类事务，T-node 可以恰当的验证并决定它们是否提交或回滚。综上，在 P-node 故障后，所有受影响的事务都可以被正确的结束。数据库的快照隔离和持久性依然能够被保证。

T-node 故障恢复. T-node 将 Memtable 保存在内存中。为了避免数据丢失，它实现了写前日志，并为所有的已提交事务在本地磁盘上保存 redo 日志记录。当 T-node 发生故障后，它能够通过回放本地的 redo 日志来恢复内存中的数据。进一步

的，为了避免单点故障，Cedar 会将所有的 redo 日志记录使用 Raft [39, 61] 协议同步到两个备 T-nodes。为了副本一致性，Raft [39] 协议要求任意事务提交前需要将它的 redo 日志记录同步到半数以上的 T-nodes 中。每个备 T-node 上的 Memtable 副本通过回放日志来与主副本保持同步。当主 T-node 故障下线后，所有活跃的事务都会直接终止。Cedar 会在剩余的两个备 T-node 中选出一个新的主副本。系统中所有后续的提交请求会转发到新的主 T-node 上。因此，Cedar 能够快速的从 T-node 的故障中恢复，并在若干秒内恢复系统的服务。

S-node 故障恢复. S-node 的故障不会导致数据的丢失。这是因为它将所有的数据都保存在本地磁盘中。另一方面，单个 S-node 的故障也不会影响系统的整体可用性。这是因为 SSTable 为每个 tablet 都保存了至少 3 个副本。这些副本分布在不同的 S-nodes 上。当一个 S-node 故障下线后，P-node 依然可以从其他的 S-nodes 读取到它需要访问的 tablet。

本节主要介绍了 Cedar 在正常情况下的并发控制和系统恢复策略。然而，Cedar 的一个特殊性在于需要进行周期性的数据合并。下一节将介绍如何在数据合并期间保证事务语义。

3.2 数据合并期间的事务管理

数据合并将 Memtable 中的数据转移到 SSTable 中。它使得 T-node 能够彻底的回收 Memtable 占用的内存空间。数据合并会通过合并已有的 Memtable 和 SSTable 产生一个新的 SSTable。下文将讨论如何在数据合并期间保证事务处理。

3.2.1 数据合并

假定 m_0 和 s_0 分别是当前系统中存在的 Memtable 和 SSTable。数据合并会通过合并 m_0 以及 s_0 创建一个新的 SSTables_1。此外，它会在 T-node 上创建一个新的 Memtable m_1 来替换 m_0 接收后续的事务写入。逻辑上，s_1 保存了每条记录在 m_0 或者 s_0 上存储的最新版本。它是一个一致的数据库历史快照。这意味着对当前合

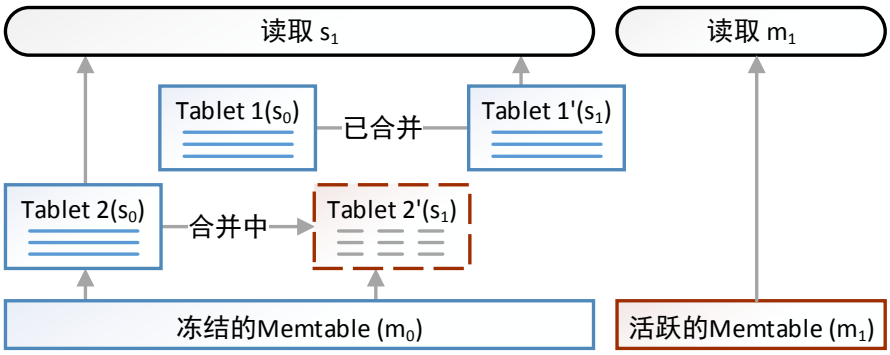


图 3.1: 数据合并期间的数据访问机制

并操作而言，存在一个时间戳 t_{dc} ，它满足所有在 t_{dc} 前提交的事务将更新数据保存在 s_1 中，而在 t_{dc} 之后提交的事务将更新数据保存在 m_1 中。

当数据合并启动时，T-node 会创建 m_1 来接收后续的写入数据。这里，当且仅当一个事务的验证阶段开始于合并操作启动之前，那么它会将它的更新数据写入 m_0 中。T-node 会等待这类事务全部完成提交。之后，没有任何事务会再次更新 m_0。此时，S-nodes 开始将它们本地的 tablets 与 m_0 合并。如上一章提及的，S-node 不会直接修改已有的 tablet 结构。它会使用写时复制技术更新 tablet 的数据块。针对需要被更新的数据块，写时复制会生成数据块的拷贝，然后在拷贝版本上写入合并后的记录。该机制保证了：当前正在执行的事务依然可以读取 s_0 中的数据。当一个 tablet 完成了对 m_0 上相应范围数据的合并后，它所在的 S-node 会通知 T-node 对应范围的数据已经完成合并了。在所有新的 tablets 都合并完成后，数据合并操作进入完成阶段。此时，T-node 允许删除内存中的 m_0 并回收内存资源。此外，它可以删除磁盘上保存的属于 m_0 的 redo 日志。

图 3.1 说明在合并期间事务是如何读取记录的。如果读取请求访问的记录是在 t_{dc} 时刻后提交的，那么它可以直接读取 m_1。反之，它会读取 s_1。对 s_1 的访问可能存在以下两种情形：1）如果将要读取的记录所在的 tablet 已经完成了合并（例如，图 3.1中的 Tablet 1’），那么仅需从 s_1 对应的 tablet 上读取所需的记录；2）如果将要读取的记录所在的 tablet 尚未完成合并（例如，图 3.1中的 Tablet 2），那么处理该读取请求的 S-node 会从 m_0 读取所需记录的增量修改，并与该记录在 s_0 中

的部分合并，最后将合并的结果返回给 P-node 。

3.2.2 合并期间的并发控制

在数据合并过程中，Cedar 依然要保证快照隔离。在第 3.1.1节的基础上，本文使用以下并发控制策略保证合并期间的快照隔离。

1）如果一个事务的验证阶段开始在数据合并操作初始化之前，这些事务被归为*类型一*。这些事务按照节 3.1.1给出的步骤，在 m_0 上验证写-写冲突，并写入更新数据。

2）仅当所有类型一的事务完成验证阶段并回滚或者获取提交时间戳后，数据合并操作会从全局计数器获得一个合并时间戳 t_{dc}。因此，所有类型一事务获得的提交时间戳势必都小于 t_{dc}。数据合并会在类型一的事务都结束后真正开始。此时，m_0 的内容不会再改变。

3）如果事务的验证阶段开始在数据合并操作初始化之后，该事务被归为 *类型二*。这种类型的事务会在合并操作获得它的时间戳 t_{dc} 之后真正开始进行验证。这主要是为了保证它们获得的提交时间戳必然大于 t_{dc}。类型二的事务会同时在 m_0 和 m_1 上验证是否发生写-写冲突，提交阶段它们会将数据写入 m_1 中。给定一个类型二的事务 t_x，对于写集合中的每个记录 $r \in w_x$，t_x 会获得 r 在 m_0 和 m_1 上的锁，然后分别验证该记录是否存在更新的版本（版本号大于 rt_x）。在验证成功后，t_x 会将更新内容写入 m_1 中，并获得它的提交时间戳。

4）最后，如果事务获得的读取时间戳大于 t_{dc}，该事务被归为 *类型三*。此类事务仅需要在 m_1 上验证，并在验证成功后将数据写入 m_1 中。

图 3.2 解释了在合并期间针对不同类型事务的验证和写入方式。

正确性. 事务的一致性快照读取是基于读取时间戳实现的。这部分的证明和第 3.1.1节中普通情况下保证快照隔离的证明相同。

这里提出的事务并发机制以及数据合并的过程同样能够防止丢失更新的异常出现。考虑一个事务 t_x，它的读取时间戳和提交时间戳分别为 rt_x 和 ct_x。假设存

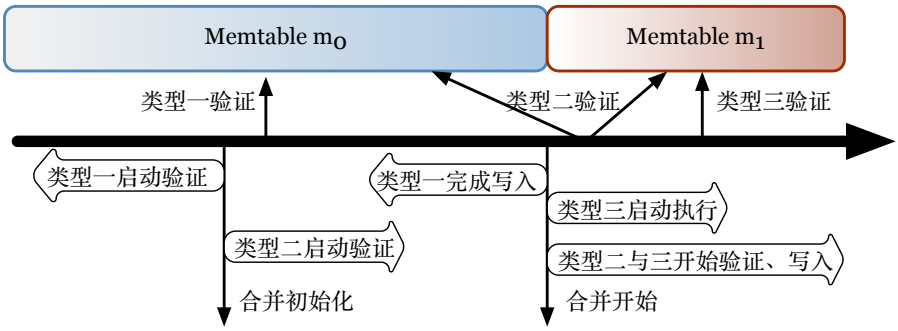


图 3.2: 合并期间的事务管理

在一个并发的事务 t_y，它在 rt_x 时刻和 ct_x 时刻之间提交，即满足 $rt_x < ct_y < ct_x$，此外 t_y 修改了部分 t_x 同样也会修改的记录。这里只需要考虑 $ct_y < t_{dc} < ct_x$ 的情况。这是因为对于其他情况而言，t_x 和 t_y 是在相同的 Memtable 集合上验证并提交写入数据的。根据第 3.1.1节给出的证明，这些情况下必然不会出现丢失更新。当满足 $ct_y < t_{dc} < ct_x$ 时，必然有 $rt_x < ct_y < t_{dc} < ct_x$。所以，$t_x$ 属于**类型二**的事务。它会同时在 m_0 和 m_1 上验证是否发生了丢失更新的异常。因此，t_x 在验证阶段必然会发现由 t_y 提交产生的写入版本。此时，t_x 会发现某个记录存在一个版本号大于 rt_x 的新版本，t_x 将会主动回滚。所以，t_x 和 t_y 不会同时成功提交，因此不会导致丢失更新。综上，即便在数据合并期间系统中存在并发执行的事务，本章提出的并发控制技术也不会导致丢失更新的异常。

3.2.3 合并期间的系统恢复.

若数据合并期间发生了节点故障，恢复机制需要能够正确的恢复 m_0 和 m_1 的数据。如上文所述，数据合并操作将事务分成了不同的类型。类型一的事务在数据合并操作开始前完成验证和提交（获得提交时间戳）。在恢复阶段，它们提交的修改内容应该恢复到 m_0 中。类型二和类型三的事务在数据合并操作开始后真正进入验证阶段。在恢复阶段，T-node 应将它们产生的修改恢复到 m_1 中。以上分析是针对系统故障时正在进行的数据合并操作而言的。除此之外，系统需要考虑如何处理最近一次**已完成**的数据合并操作。如果一个事务在该合并操作开始前成功提交，那么在恢复阶段不需要恢复这些事务产生的更新。这是因为，它们提交的数据已

经成功持久化到了 SSTable 中。

为了实现以上目标，当数据合并算法获得时间戳 t_{dc} 并正式启动时，T-node 会向写前日志文件中写入一条*合并启动日志*（Compaction Start Log Entry, CSLE）。当数据合并结束后，T-node 会在本地磁盘的特定文件中写入*合并结束日志*（Compaction End Log Entry, CELE）。CELE 会保存本次合并操作的 CSLE 日志项在写前日志中的实际位置。值得说明的是，在写前日志中，事务的 redo 日志的顺序是与它们提交时间戳的大小一致的。这保证了类型一的事务产生的 redo 日志均出现在 CSLE 之前（因为这些事务的提交时间戳小于 t_{dc}），而类型二和类型三的事务产生的 redo 日志均出现在 CSLE 之后。

当 T-node 故障下线后，恢复算法会读取当前 CELE 的内容，获得最近已经完成的数据合并操作的 CSLE 日志的位置，并从该位置开始读取和回放写前日志。一开始，恢复算法会将数据写入到 m_0 中。当它读取到下一个 CSLE 时，它会在 T-node 上创建一个新的 Memtable m_1。之后，它会将数据写入到 m_1 中。在 T-node 恢复后，S-nodes 可以继续读取 m_0 的数据，并与本地的 tablet 合并。

如果在数据合并期间 S-node 故障下线了，S-node 上的数据不会丢失，因为它采用了磁盘存储。但是某个 S-node β 在下线前可能正在进行数据合并。因此，当 β 重新接入集群后，它可能保存了老 SSTable 的 tablet，以及由合并产生的部分不完整的新 tablet。由于 β 上保存的数据在其他 S-nodes 上保存了副本。当 β 重新接入系统时，可能出现其他存活的副本已经完成了对 m_0 数据的合并，本次数据合并操作已经结束了。此时，刚恢复的节点 β 可以直接从其他副本上拷贝最新的 tablet。如果数据合并操作尚未结束，那么 β 可以继续从 T-node 读取 m_0 并完成合并。

最后，P-nodes 上不保存任何数据。因此，在合并期间，P-node 的故障不会导致数据丢失，同样也不需要进行恢复。

3.2.4 存储管理

在合并期间，m_0 和 s_0 处于只读状态，而 s_1 和 m_1 正在被持续更新。当数据合并结束后，s_1 保存了 m_0 和 s_0 中记录的最新版本，此时系统可以将 m_0 和 s_0 删除了。删除 m_0 时不仅可以删除内存中的 Memtable ，同时可以删除磁盘上用于恢复 m_0 的写前日志。但删除前必须保证没有任何事务正在使用这些结构，即没有任何活跃事务正在使用小于 t_{dc} 的读取时间戳。

3.3 网络通信优化

目前为止，本章主要介绍了如何保证 Cedar 事务处理的正确性。Cedar 采用了分布式的架构。数据分布存储在 S-nodes 和 T-node 上。系统解耦了查询处理层和存储层，由 P-nodes 负责事务逻辑，SQL 请求等的执行。这导致了事务处理过程中会引入大量的网络通信。因此，Cedar 基于自身的架构特点，设计了多种有效的通信优化方式来减少节点间的通信。

3.3.1 数据访问优化

在事务执行过程中，它将要访问的记录版本是由事务的读取时间戳指定的。这些记录版本可能存储在 S-nodes 的 SSTable 上，也可能存储在 T-node 的 Memtable 上。因此，Cedar 并不知道应该从哪个节点读取合适的版本。负责事务执行的 P-node 需要同时访问 SSTable 和 Memtable 来保证读取内容的一致性。但实际上，其中一次远程访问是无效的。

本节提出了两种机制来减少 P-nodes 对 S-nodes 以及 T-node 产生的远程数据访问。首先，P-nodes 建立了 SSTable 的数据缓存池来减少 P-node 对 S-nodes 的远程数据访问。然后，本节提出和设计了一种异步比特数组结构来识别可能的无效 T-node 访问。

3.3.1.1 SSTable 缓存

事务执行过程中，P-node 需要访问 SSTable 上存储的记录。这些远程数据访问请求可以有效的使用数据缓存来优化。而且 SSTable 的不变性使得 P-nodes 能够很容易地设计缓存机制。数据缓存池会保存以往从 SSTable 上读取的数据记录。它可以优化未来对相同记录的重复访问。

数据缓存池是一个基于哈希表的键-值存储结构。键部分存储了记录的主键，值部分存储了实际的记录内容。在处理对 SSTable 的数据读取请求时，P-node 会首先查询该记录在数据缓存池中是否存在。仅当缓存未命中时，P-node 会发起对 S-node 的远程数据访问。在收到 S-node 回复的内容后，P-node 会将结果添加到数据缓存池中。当缓存池预留的内存空间耗尽时，它会使用通用的缓存淘汰算法（Least Recently Used）来删除特定的缓存内容。每个 P-node 各自维护一个数据缓存池。由于单个 P-node 上会同时运行多个事务，它们可能会并发地访问数据缓存池。为了保证并发访问的正确性，缓存池使用的哈希表是由临界区保护的。为了避免并发读取请求争夺临界区的访问权限产生较多的冲突，P-node 进一步使用哈希分区的方式将一个大的哈希表划分为若干较小的哈希表。每个小的哈希表使用由不同的临界区保护。此时，如果两个读取请求访问的记录落在不同的哈希表上，那么它们会进入不同的临界区，避免产生冲突。

由于每个 SSTable 在创建后始终维持不变，因此，P-node 缓存 SSTable 的内容后不需要担心数据的分布式一致性问题。此外，SSTable 的数据是保存在磁盘上的，所以 Cedar 也不需要持久化缓存池的内容。当一个 P-node 下线重启后，它可以在事务处理过程中访问 SSTable 并重新构建本地缓存池。最后，P-node 上缓存的内容同样会过期。当发生数据合并操作后，现有的 SSTable 会被一个新的版本替代。此时，P-node 会重新创建一个新的数据缓存池。

3.3.1.2 异步比特数组

SSTable 是整个数据库的一个一致的历史快照。它保存了数据库运行至最近一

次合并产生的所有数据。相对而言，Memtable 仅保存自上一次数据合并后上层应用提交的修改。Memtable 中的数据对整个数据库而言是很小的一部分。这隐含了以下事实：*一个读取请求访问 T-node 极有可能获得不到任何数据*。这种现象被称为*空读*。对整个系统而言，空读不仅是无效的，同时存在一些负面作用。它会增加事务处理的延迟，而且会耗费 T-node 的处理能力。

为了避免事务执行过程中产生大量的空读，T-node 使用了一种称之为 *记事本* 的结构来编码 Memtable 上数据的存在情况。该结构会周期性的同步到所有的 P-nodes 上。然后，每个 P-node 可以查看本地缓存的 *记事本* 来高效地判断某个 T-node 访问是否有可能是空读。

结构. *记事本* 本质上是一个比特数组。在该数组中，每个比特用于代表某个 tablet 上的某列是否在 Memtable 上保存了修改内容。换言之，如果一个 tablet T 中任意记录的列 C 被修改了，那么对应 (T, C) 的比特会被设置为 1；否则，该比特位为 0。除了这种编码方案，其他的方式也是可行的（例如，每个比特对应一条记录），但这样会大大的增加比特数组的大小。

维护. Cedar 在集群中保存了两种类型的比特数组。第一类是 T-node 上保存的原版比特数组，记为 b。第二类是每个 P-node 上缓存的异步比特数组，记为 b'。它是 b 在某个时刻 t 的版本，即 $b' = b_t$（b_t 是 b 在 t 时刻的版本）。在事务执行过程中，P-node 仅通过查询本地缓存的 b' 来判断将要执行的读取请求是否为空读，而不必与 T-node 进行额外的通信交互。

当 Memtable 上的某个记录的某列被首次更新时，T-node 会尝试更新 b。值得注意的是，如果某个数据项（某行的某列）在 Memtable 上已经存在了一个版本，那么此时 b 将不需要被再次更新。因为 b 必然已经保存了该数据项存在的信息。每个 P-node 会周期性的从 T-node 拉取最新的 *记事本* 版本，然后更新本地缓存的 b'。

使用. 在 P-node p 处理事务 t_x 的过程中，p 会检查本地的 b' 来判断 T-node 上是否保存了事务 t_x 所需记录的新版本。若 t_x 将要读取记录 r 的第 C 列，并且 r 属于的 tablet 为 T，如果 b' 中对应 (T, C) 的比特位为 0，那么 p 会认为当前的读取请

求为空读，继而跳过本次对 T-node 的访问；否则，p 会发起对 T-node 的远程数据访问。

由于比特数组编码的粒度较粗，因此查询 b' 可能会导致假阳。换言之，b' 认为某个数据项在 T-node 上存在，但实际上它不存在。假阳会导致对 T-node 的空读。考虑一个 tablet T，以及 T 保存的记录 r_1 和 r_2。若 r_1 的第 C 列在 Memtable 更新产生了新版本，而 r_2 的第 C 列没有被更新过。此时在读取 r_2 的第 C 列时，P-node 会发现 (T, C) 比特位的取值为 1，继而尝试访问 T-node 并产生一次空读。事实上，记事本 主要是用来优化对只读列（或读多写少）的访问。对于这些列而言，假阳的问题并不会很显著。因为，它们在 记事本 中的比特位通常不会被设为 1 。

查询 b' 得到的结果同样存在假阴。这是因为 b' 是 记事本 在某个历史时刻的异步版本。它和 T-node 上保存的最新版本的 b 并不是完全同步的。当出现假阴时，P-node 可能会忽略某条记录的最新版本，进而读取到落后的版本。这会破坏事务的快照隔离语义。为了避免该问题，事务会在验证阶段再次检查执行阶段遇到的所有可能的空读请求。在执行阶段，如果事务查到 b' 的第 (T, C) 个比特取值为 0，那么它在验证阶段会再次查看 b 中该比特位是否也为 0。如果任何由 b' 判定的空读没有得到 b 的确认，那么当前事务会读取 Memtable 保存的记录最新版本并重新执行。由于 b 的更新仅发生在一个 tablet 中任意行的一列首次被修改时，因此它实际被修改的频率很低。b 不会频繁的被修改。因此，假阴很少发生。

3.3.2 事务编译

Cedar 除了支持 JDBC/ODBC 连接外，同样支持存储过程。事务的执行逻辑可以被表示为存储过程代码。使用存储过程时，事务服从 One-Shot 执行模型 [9] 。换言之，事务执行过程中不会有任何的客户端-服务器交互。通常，使用存储过程会增加数据库系统的计算负担。但它能够有效的降低每个事务请求的处理延迟。同时，它还使许多服务端的优化策略成为了可能 [36, 62–64]。基于以上条件，Cedar 设计了事务编译模块。该模块会为每个事务实例生成更优的执行计划，从而大大

减少执行过程中产生的节点间网络通信。

3.3.2.1 存储过程的表示

在给定不同的输入参数和数据库快照时，执行同一个存储过程会产生不同的事务实例。在存储过程创建后，P-node 会编译并生成它的物理计划。一个存储过程的物理理计划可以被表示为多个操作构成的执行序列。表 3.1列出了存储过程物理计划中可能包含的操作。这里，诸如分支、循环等嵌套结构可以被视为单个复合操作。在表 3.1中，读操作需要访问存储层的数据，它们被实现为远程过程调用；而其他操作均只包含 P-node 本地计算。

表 3.1: 存储过程物理计划中包含的操作类型

读操作	读取 Memtable
	读取 SSTable
写操作	将更新内容写入 P-nodes 的本地缓冲区
计算操作	算术表达式，关系表达式，过程语言操作

给定一个存储过程的操作序列，编译阶段可以通过调整两个操作的执行次序来产生一个更优的执行计划。调整的关键是需要保证存储过程的原始语义，即在给定不同的输入情况下，执行原始的操作序列和调整后的序列应该始终生成相同的事务实例。

这里，可以定义操作之间的两种约束关系，并证明如果操作之间不存在这些关系，那么调整它们的执行次序对语义不会产生影响。

- 程序约束：如果两个操作之间存在数据依赖或控制依赖 [65]，那么它们之间存在程序约束。这里，当两个操作访问了相同的变量，并且其中一个是写操作，那么它们直接存在数据依赖。当一个操作是否执行取决于另外一个操作的执行结果，那么它们之间存在控制依赖。

- 访问约束：如果两个操作访问了相同的数据库记录，并且其中一个操作为写操作，那么它们之间存在访问约束。

程序约束保证了多个操作是按照正确的顺序读取和修改变量，同时程序的分支和循环结构被正确的执行。访问约束可以被理解为一种面向数据库记录的特殊数据依赖。可以证明如果调整操作次序没有破坏操作之间的约束关系，那么调整后的物理计划和原始物理计划语义相同。

证明. 首先，一个事务实例可以采用以下定义形式化的描述 [66]。在该模型下，两个事务实例被认为具有相同的语义，如果它们之间满足：两者的读/写操作集合完全相同，并且操作之间的偏序关系集合也完全相同。

定义 3.3.1 (事务语义)**.** *给定数据库 D，事务实例 t 是由具有偏序关系的若干读操作 read(x) 和写操作 write(x) 构成的，其中 $x \in D$；偏序关系存在访问相同数据项的读操作和写操作（以及多个写操作）之间。*

另一方面，如果两个物理计划满足以下定义的关系，那么它们可以被认为是相同的。

定义 3.3.2 (等价关系)**.** *当给定任意相同的参数和数据库状态时，如果两份物理计划产生的事务实例具有相同的语义，那么这两个物理计划可以被认为是等价的。*

考虑一个物理计划：$p = o_1, o_2, \ldots, o_n$ 以及根据上文给出的约束合法变化后产生的计划：$p' = o'_1, o'_2, \ldots, o'_n$，可以证明：当给定任意相同的参数和数据库快照时，它们会始终生成具有相同语义的事务实例。

如果 p' 中所有的操作的输入和 p 中对应操作相同，那么 p 和 p' 必然生成相同的读/写操作集合。考虑任意两个操作 o'_u 和 o'_v，它们访问相同的记录并且其中一个是写操作。它们在 p' 中的相对顺序必然与在 p 中顺序一致。否则，改变它们的顺序会违反访问约束。

如果 p' 中某个操作的输入与 p 中对应操作不相同，假设 o'_i 是首个获得不同输入的操作。导致输入不同的原因有以下几种：

（1）o'_j 修改了 o'_i 读取的变量，而 o'_j 本应该在 o'_i 之前（或之后）执行，但实际上没有。在这种情况下，o'_i 和 o'_j 之间存在数据依赖关系。所以，程序约束会限制

它们的顺序被调整。该情况不会出现。

（2）o'_u 和 o'_v 都在 o'_i 之前修改了同一个被 o'_i 读取的变量，而 o'_u 和 o'_v 的实际执行顺序与原始物理计划 p 中顺序不同。这会被修改的变量最终取值发生改变。在这种情况下，o'_u 和 o'_v 之间存在数据依赖关系。程序约束会限制它们的顺序被调整。该情况也不会出现。

（3）o'_i 读取数据库记录得到了不同的值。如果 o'_i 读取的是 p' 中某个写操作产生的结果，采用类似于情况（1）中对读写相同变量的分析，可以推断这种情形同样不会出现。如果 o'_i 读取的是另外一个事务产生的写入，那么由于 p 和 p' 操作的是相同的数据库快照，o'_i 在这两个物理计划中读取到的记录取值必然是相同的。所以，这种情况也不会出现。

综上，p 和 p' 产生的事务实例的语义始终相同。 □

根据以上证明，可以明确：给定 p，在不改变已有约束关系的前提下，合法的调整操作执行顺序产生的物理计划 p' 与原始的物理计划 p 是等价的。

事务编译算法可以通过比较两个操作的读/写变量集合和记录集合来识别它们之间的约束关系。操作使用到的变量在编译期间就可以确定。另一方面，操作可能的数据库记录在编译期间是无法确定的。但两个操作仅当访问相同的数据表时，才有可能访问到相同的记录。因此，若两个操作访问相同的表且其中一个是写操作时，那么这两个操作之间可能存在访问约束。

在得到操作间的约束关系后，存储过程的操作序列可以被表示为一个 **执行图**。在图上，每个节点代表了一个操作，每条有向边代表了操作间相对执行顺序的约束。图 3.3 给出了一个例子。如果图上存在一条从操作 o_y 到操作 o_x 的路径，那么这种关系称之为 o_x 的执行受到 o_y 的限制。

在操作序列中，嵌套结构（如分支、循环）被表示为单个复合操作。为了实现更好的优化效果，事务编译需要尝试进一步分解复合操作。循环结构可以使用 loop distribution [67, 68] 技术将一个大的循环切分为若干更小的循环。每个小循环对应一个复合操作。针对分支结构，它包含的控制依赖仅能通过真实的执行解决。但分

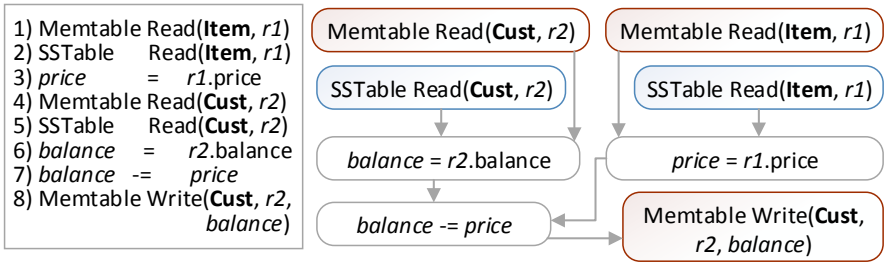


图 3.3: 存储过程操作序列和执行图的例子

支结构中的读操作可以被提前执行，并缓存返回的结果用于后续的操作。尽管实际执行可能不会进入某个分支，但额外执行这些分支中的读操作不会改变存储过程的语义。这是因为读取 SSTable 或者 Memtable 的快照不会改变任何变量或者记录。所以，如果一个读操作不受分支结构内任何其他操作的限制，那么将该操作拿到分支外执行是安全的。

如果一个复合结构仅包含若干计算操作，那么它将被标记为计算操作。如果一个复合结构仅包含一次 SSTable 读取和若干计算操作，那么它被视为 SSTable 读取操作。如果复合结构包含一次 Memtable 读取和若干计算操作，那么它被视为 Memtable 读取操作。如果复合结构包含多次 Memtable 读取或者 SSTable 读取，那么它会被标记为复杂操作。

3.3.2.2 合并优化.

基于以上对存储过程物理计划的表示方式，本节首先提出一种针对 Memtable 读取的合并优化算法。在乐观并发控制协议中，事务的持续时间是从它获得读取时间戳，到获得提交时间戳的这一段时间。在这期间，任何冲突的写入操作均会导致当前事务回滚。合并优化会尝试打包多个 Memtable 远程读取操作，从而降低事务延迟和持续时间。

为了减少远程数据访问，一种优化方式是将多次 Memtable 读取操作合并为一次通信。如果两个 Memtable 读取操作不会互相限制对方的执行，那么这两个操作可以合并；否则，其中一个 Memtable 读取需要等待另外一个执行完成后才能执行。

算法 1用于合并多个 Memtable 读取操作。算法输入是某个存储过程的物理计

划 seq。在该算法中，第一个循环会标记出所有能够被合并的 Memtable 读取操作。在该循环中，如果某个 Memtable 读取操作直接或者间接地受另一个 Memtable 读取限制，那它将被标记为不可合并。复杂操作（包含多次 Memtable 或 SSTable 读取的复合操作）同样不可以被合并。这是因为它内部包含的多个 Memtable 读取操作可能互相间存在约束关系。第二个循环会将所有可以被合并的 Memtable 读取添加一个操作序列 $group\_op$ 中。此外，如果某个本地计算操作必须在任意被合并的 Memtable 读取操作之前执行（两者间存在约束关系），那么该计算操作同样会被添加到 $group\_op$ 中。算法最后返回 $group\_op$。算法 1 的整体复杂度为 $\mathcal{O}(|seq|+|E|)$，其中 $|seq|$ 是输入的物理计划操作序列 seq 的长度（包含的操作数量），而 $|E|$ 是操作之间约束关系的数量。

算法 1: 合并 Memtable 读取操作

输入: 操作序列: $seq = (o_1, o_2, ..., o_n)$
输出: 合并执行的操作: $group\_op$

```
1  Initialize label[1 : n] = normal, group_op = ();
2  for i = 1 to n do
       /* 标记执行受限于其他 Memtable 读取的操作 */
3      if o_i is complex operation then
4          label[i] = block;
5          continue;
6      forall edge e ends with o_i do
7          let e starts with o_j;
8          if label[j] == (group or block) then
9              label[i] = block;
10             break;
       /* 标记可以被合并的 Memtable 读取操作 */
11     if o_i is Memtable read and label[i] ≠ block then
12         label[i] = group;
13 for i = n to 1 do
14     if label[i] == group then
           /* 处理约束关系 */
15         forall edge e ends with o_i do
16             let e starts with o_j;
17             label[j] = group;
18         add o_i to the front of group_op;
```

算法 1生成了一个操作序列 $group\_op$。P-node 会依次执行 $group\_op$ 中包含的操作。其中，本地计算操作会被直接执行；Memtable 读取操作产生的数据请求会添加到一个固定的成组通信请求中。当 $group\_op$ 中所有的操作都执行完成后，P-node 会将成组通信请求发送给 T-node ，并将返回的结果缓存在存储过程执行上下文中。至此，$group\_op$ 执行完成，之后 P-node 将开始实际执行存储过程的物理计划 seq。此时，当遇到了被合并的 Memtable 读取操作时，P-node 会直接从本地缓存中获取所需的数据。

3.3.2.3 预执行优化.

算法 2: 预执行 SSTable 读取操作

输入: 操作序列: $seq = (o_1, o_2, ..., o_n)$
输出: 预执行的操作: pre_op

```
1  初始化 label[1 : n] = normal, pre_op = () ;
2  for i = 1 to n do
       /* 标记执行受限于其他 Memtable 读取的操作 */
3      if o_i is complex operation or Memtable-read then
4          label[i] = block;
5          continue;
6      forall edge e ends with o_i do
7          let edge e starts with o_j;
8          if label[j] == block then
9              label[i] = block ;
10             break;
       /* 标记可以被预执行的 SSTable 读取操作 */
11     if o_i is SSTable-read and label[i] ≠ block then
12         label[i] = preexec ;
13 for i = n to 1 do
14     if label[i] == preexec then
           /* 处理约束关系 */
15         forall edge e ends with o_i do
16             let edge e starts with o_j ;
17             label[j] = preexec ;
18         add o_i to the front of pre_op ;
```

在 Cedar 中，SSTable 保存的数据是不变的。即便数据合并操作也只会生成一

个新的 SSTable，而不会修改现有的结构。因此，在事务开始（首次访问 T-node 获得读取时间戳）之前与之后执行 SSTable 读取操作产生的效果是完全相同的。给定一个事务实例 t_x 的存储过程，P-node 可以在 t_x 真正开始前尽可能地执行物理计划中的 SSTable 读取操作。这样可以有效地减少 t_x 开始后执行的远程数据访问，从而降低事务的持续时间。由于 t_x 是在执行第一个 Memtable 读取操作时从 T-node 获得读取时间戳的，因此如果一个 SSTable 不受任意 Memtable 读取操作限制，那么它可以被预执行。

实际上，在处理一条记录 r 的读取请求时，P-node 会先读取 Memtable ，仅当这次远程访问没有返回 r 的任何版本时，它会继续读取 SSTable。而在预执行优化中，P-node 在读取 r 时会先访问 SSTable，而后访问 Memtable 。事实上，改变读取顺序并不会影响结果的正确性。这种改变的主要缺点在于可能会引入一些不必要的 SSTable 读取（如果对应的 Memtable 读取能够返回记录的最新版本）。但使用第 3.3.1.1节提出的 SSTable 数据缓存可以减少这些额外的开销。并且对大多数记录而言，它们的最新版本是保存在 SSTable 中的。

算法 2 用于找出可以预执行的 SSTable 读取操作。它的输入是存储过程的物理计划 seq。第一个循环会标记出所有可以预执行的 SSTable 读取操作。下一个循环会构造一个预执行操作序列 pre_op。该序列包含了可以预执行的 SSTable 读取操作，同时也包含了需要在任意 $op \in pre\_op$ 前执行的本地计算操作（两者间存在约束关系）。该算法的复杂度为 $\mathcal{O}(|seq| + |E|)$。

给定一个存储过程调用请求，P-node 会首先执行算法 1 生成的 pre_op，然后执行算法 2 生成的 $group\_op$，最后执行实际的物理计划 seq。执行 pre_op 返回的 SSTable 数据会被缓存在存储过程执行上下文中。在实际执行 seq 时，如果遇到了被预执行的 SSTable 读取操作时，P-node 会直接从缓存中获取所需的数据。

3.4 实验与分析

本节重点对比了 Cedar 和其他分布式系统在处理不同类型负载时的性能。下文首先介绍实验的环境和系统的部署，然后通过不同类型的测试验证 Cedar 的性能。

3.4.1 实验环境

硬件环境. Cedar 是在开源数据库系统 Oceanbase [69] 的代码基础上实现的，在原始的代码库上添加或修改了 51,281 行 C++ 代码。因此，Cedar 是一个建立在 457,206 行代码基础上的完整数据库系统。为了与其他基于高端网络设备的数据库系统进行比较，所有的实验都运行在 Emulab 平台 [70] 提供的 11 台服务器上。该平台能够允许客户方便地配置不同的网络设备和拓扑结构。每个服务器配备了两个 2.4GHz、8 核心的 E5-2630 处理器。当启用超线程技术后，它能够同时运行 32 个线程。服务器之间默认是采用千兆以太网络链接的。各个数据库系统默认部署在 10 台服务器上。另外一台服务器用于模拟客户端。

对比的系统. 实验对比 Cedar 和 MySQL-Cluster 5.6，Tell（共享一切的分布式系统代表）[31]，VoltDB 6.6 企业版（无共享的分布式系统代表）[29] 在事务处理方面的吞吐率和延迟。由于 Tell 依赖于支持 RDMA 的 InfiniBand 高端网络设备，为了保证各个系统之间性能对比是公平的，我们在以太网上使用 RoCE（RDMA over Converged Ethernet）构建了 RDMA 环境，并使用 Tell-1G 和 Tell-10G 分别代表在千兆网（1-Gigabits）和万兆网（10-Gigabits）上部署的 Tell 系统。

Cedar 没有和轻量级的原型数据库系统对比，如 Silo [15] 。因为这些系统主要是为验证新型并发控制协议而设计的。虽然它们给出令人印象深刻的吞吐率数字，但是它们无法在真实应用中使用。这些系统通常缺少一些重要的特性，例如写前日志、SQL 引擎以及客户端网络链接功能等。在真实系统中，这些特性通常会产生不可忽视的性能代价，但被轻量级的原型系统所忽略。

部署方式. Cedar 使用一台服务器部署 T-node ，在每个剩余服务上部署一个 S-node 实例以及一个 P-node 实例。Tell 使用一个服务器部署提交管理器，使用两

个服务器部署存储节点，并使用剩余的服务器部署处理节点。针对 Tell 的部署，实验测试了不同的处理节点和存储节点数量组合，并选择了最优的配置方式。Tell 需要的处理节点数量要多于存储节点数量。MySQL-Cluster 在每个服务器上部署了一个 mysqld 和 ndbmtd 实例，其中，前者是 SQL 计算引擎而后者是内存存储引擎。根据官方推荐的配置策略 [1]，VoltDB 在每个服务器上创建了 27 个数据分区。这是通过调整单个服务器上的数据分区数量以达到最高的事务吞吐率，最终确定了最优的分区数量。

测试方式. 实验使用了三种不同的基准测试来验证各个系统的性能：TPC-C，Smallbank 以及 E-commerce。各个系统的性能是通过事务吞吐率评估的（每个事务处理数量，TPS）。每个测试样例调整并发客户端的数量来达到每个系统最优的吞吐率。TPC-C 以及 Smallbank 的负载模拟使用了开源软件 oltpbench [71]。

3.4.2 实验结果与分析

在下文中，实验首先使用三种不同的测试基准来对比不同系统的性能。然后，若干不同的实验对 Cedar 在数据合并，节点故障等方面的表现进行独立的验证。

3.4.2.1 TPC-C 测试

该实验采用了一个标准的 TPC-C 负载。表 3.2 列出了负载中包含的各类事务比例。事务请求调用参数是根据 TPC-C 说明文档生成的。默认情况下，数据库中存储了 200 个仓库的数据。其中，Shared-Nothing 系统根据仓库表的主键进行数据分区。

表 3.2: TPC-C 负载中各类事务的比例

事务类型	NewOrder	Payment	OrderStatus	Delivery	StockLevel
事务比例	45%	43%	4%	4%	4%

初始化状态下，Cedar 在 Memtable 中存储了约 160 万条不同的记录（共使用 2.5 GB 的内存空间）；在 SSTable 中存储了约 1 亿条不同的记录（共使用 42GB 的

[1] https://www.voltdb.com/

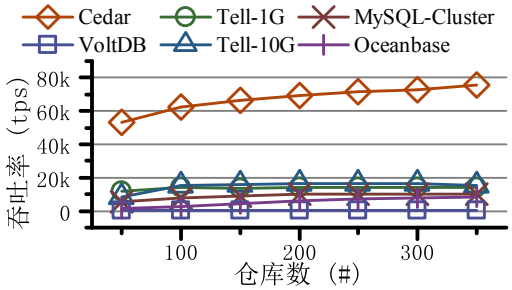


图 3.4: TPC-C：调整仓库数量

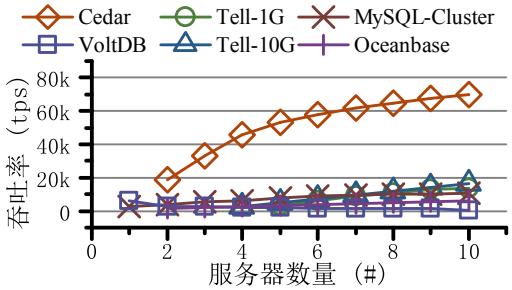


图 3.5: TPC-C：调整节点数量

磁盘空间）。由于 SSTable 为每条记录维护了 3 个副本。因此，SSTable 实际存储了 14GB 的数据。在所有的测试完成后，Memtable 中总共包含 11GB 的数据，而 SSTable 的大小约为 655GB （若不考虑副本，实际数据的大小为 215GB）。

图 3.4 给出了通过调整 TPC-C 仓库数量得到的各个系统的性能。当数据库中包含 50 个仓库时，Cedar 的性能约为 53k TPS。当仓库数量增长到 350 时，Cedar 的性能随着增长到了约 75k TPS。随着仓库数量的增加，负载中的访问冲突也随之下降。这导致了更少的访问冲突，从而事务处理获得了更高的并发度。整体上，Cedar 的性能明显的优于其他的系统。在 350 个仓库下，它的性能是 Tell-10G（约 15.6k TPS）的 4.8 倍。值得注意的是，与 Cedar 使用相同网络设备的 Tell-1G 表现出的性能还要差于 Tell-10G。此外，由于负载中存在较多（超过 10%）的分布式事务，VoltDB 的性能表现始终是最差的。最后，开源版本的 Oceanbase 主要是针对短事务优化的（每个事务中仅包含极少量的 SQL 请求）。因此，它并不擅长处理一般化的事务负载。在所有的测试案例下，Cedar 达到的性能约为 Oceanbase 的 10 倍。由于 Cedar 是在 Oceanbase 上实现的，它的性能整体上是优于后者的。因此，在后续的实验不再将 Oceanbase 列入对比系统中。

图 3.5 通过调整用于部署系统的服务器数量，来验证不同系统的横向扩展性。其中，Cedar，Tell 以及 MySQL-Cluster 的性能随着节点数量的增加而不断上升。相对而言，VoltDB 的性能是呈下降趋势的。这是因为在 VoltDB 中，分布式事务是在单个线程上排队执行的。它们的执行会阻塞系统中所有的工作线程。随着使用的服务器数量增加，处理这类事务的代价也会随着增加。最终的结果是 VoltDB 在使

表 3.3: 90 分位延迟，TPC-C 负载

延迟（ms）	Cedar	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB	Oceanbase
Payment	6	17	7	17	15619	38
NewOrder	15	28	12	103	30	60
OrderStatus	6	20	8	23	14	30
Delivery	40	160	53	427	14	174
StockLevel	9	14	7	17	14	60
Overall	12	30	12	95	2751	54

用单个服务器时达到了最优的性能。对于 Cedar 而言，它的性能增长在使用 7 台服务器后变慢了。这里的主要原因是：随着服务器数量的增加，Cedar 能够并发处理更多的事务请求以达到更高的吞吐率，但一个副作用是并发度的提升导致负载中的访问冲突也大大增加了。最终，在验证节点验证失败的事务数量也随着增加了。另外一个原因是：当与更多的 P-nodes 协同工作时，T-node 的处理压力也增加了。而在本实验配置中，T-node 和 P-nodes 使用的是相同的服务器。因此，Cedar 的整体性能随着 P-nodes 的增加呈次线性增长。在实际部署 Cedar 时，T-node 一般被推荐部署在配置相对高端的服务器上；而 P-nodes （以及 S-nodes ）可以使用性能相对较弱的服务器。这一方面可以降低整体的硬件成本，另一方面也保证系统更好的横向延伸能力。

由于 VoltDB 采用水平分区的方式进行横向扩展，它的性能在较大程度上会受到分布式事务数量的影响。图 3.6 给出了通过调整（负载中）跨仓库事务比例时各个系统表现出来的性能。由于所有记录（除商品表外）是按照各自主键中的仓库编号列分区的，如果一个事务会同时访问多个仓库的记录，那么它有可能需要访问多个数据分区，进而成为分布式事务。当负载中没有任何的分布式事务时，VoltDB 达到了最优的性能，约为 141k TPS。该性能约为 Cedar 最优性能的两倍。当随着分布式事务比率的增加，VoltDB 的性能快速的下降。相对而言，其他系统的性能受该参数的影响很小。

表 3.3 列举出了各个类型事务的 90 分位延迟。Cedar 的处理不同类型事务的延迟均较低。Tell 的延迟受到网络设备的影响。在使用万兆网时，它的事务处理延迟

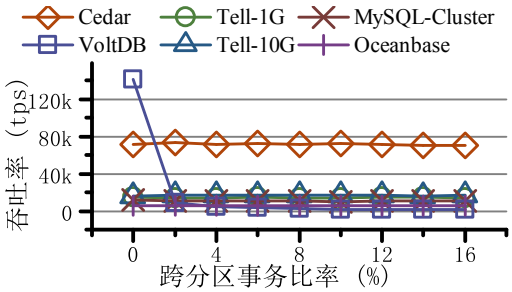


图 3.6: TPC-C：调整跨仓库事务数量

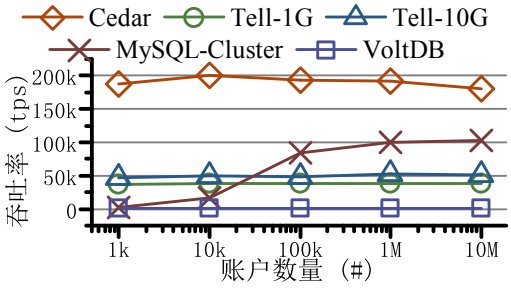


图 3.7: Smallbank：调整账户数量

要优于使用千兆网时的表现。MySQL-Cluster 的长延迟主要是由于客户端与服务器之间的交互导致的。这因为它使用 JDBC 接口交互式地发送事务的 SQL 请求，而没有采用存储过程。VoltDB 在处理分布式事务时延迟非常的高。在标准的 TPC-C 负载下，约有 15.0% 的 Payment 以及 9.5% 的 NewOrder 请求是分布式事务。因此，Payment 事务的 90 分位延迟非常的高。而 NewOrder 事务虽然在 90 分位上的延迟较低，但它 95 分位的延迟达到了 15819 ms。

3.4.2.2 Smallbank 测试

Smallbank 模拟了一个银行应用。该基准测试包含了三张表和六种事务。一张 *user* 表用于保存用户的个人信息；一张 *savings* 表用于保存定期存款账户余额；一张 *checking* 表用于保存活期存款账户余额。每张表都以 *account-id*（账户编号）为主键。负载中包含了六种事务。表 3.4 列举了每种类型事务的比例。其中，Amalgamate 和 SendPayment 类型的事务会同时操作两个不同的用户账户。而其他类型的事务仅访问一个用户账户。数据库中总共保存了 1 千万的用户账户，即每张表中保存了 1 千万条记录，总共 3 千万条记录。初始化状态下，Cedar 在 Memtable 中存储了 8 百万条记录（共占据 3 GB 的内存空间），同时在 SSTable 中存储了 3 千万条记录（共占据 1.1 TB 的磁盘空间）。在所有的实验完成后，Memtable 中保存了 5.2 GB 的数据，而 SSTable 的大小约为 1.1 TB 。

图 3.7 通过调整数据库中的账户数量来影响事务的访问冲突，从而测试各个系统的性能。图中横轴采用的是对数坐标。整体上，Cedar 保持了最好的性能。在账

表 3.4: Smallbank 测试中各类事务的比例

事务类型	Amalgamate	Balance	Deposit Checking	Send Payment	Transact Savings	Write Check
事务比例	15%	15%	15%	25%	15%	15%

表 3.5: 90 分位延迟，Smallbank 负载

延迟 (ms)	Cedar	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB
Amalgamate	5	5	4	8	100
Balance	3	3	3	4	5
Deposit Checking	4	4	4	3	5
SendPayment	7	4	4	12	102
Transact Savings	3	4	4	6	5
WriteCheck	5	4	4	5	6
Overall	5	4	4	8	92

户数量较少时，Cedar 的吞吐率随着账户数增多而有一定的提升。这是因为账户数量增加后负载中的并发访问冲突下降了。当账户数量从 100k 增加至 10 M 的过程中，Cedar 的吞吐率存在一定程度的下降。这是因为当一个 P-node 需要访问越来越多的账户时，它维护的 SSTable 缓存命中率会持续下降。这导致了 P-node 需要频繁地通过远程过程调用拉取 S-nodes 上的数据。因此，事务吞吐率存在一定的下降。

Tell 表现出了相对稳定的性能。整体上，Cedar 的性能约为 Tell-1G 的 4 倍。当使用更好的网络设备（万兆网）时，Tell-10G 的性能仅有略微的提升。MySQL-Cluster 的性能随着账户数量的增长呈明显的上升趋势。这里，由于 MySQL-Cluster 采用的是 JDBC 的方式发送事务请求。因此，事务执行过程中会引入更多的客户端-服务器网络交互。这增加了事务的持续时间，导致更多的并发事务发生访问冲突。此时，账户数量的增加能够使并发事务访问不同的账户，从而降低负载中的冲突。随着账户数量的不断增长，MySQL-Cluster 的性能最终随着各个服务器 CPU 处理资源耗尽而收敛。VoltDB 的性能始终受限于负载中分布式事务的数量。通过表 3.5 中列出的 90 分位延迟，可以看到 VoltDB 在处理 *Amalgamate* 和 SendPayment 时耗费的时间非常的长。这两类事务会访问多个账号的数据。而不同的账户可能存储在不同的分区上。因此，它们可能形成分布式事务。另一方面，这两类事务总共构成

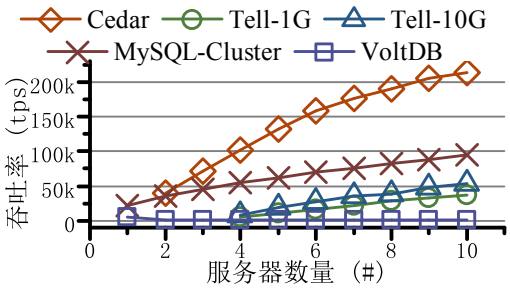


图 3.8: Smallbank：调整节点数量

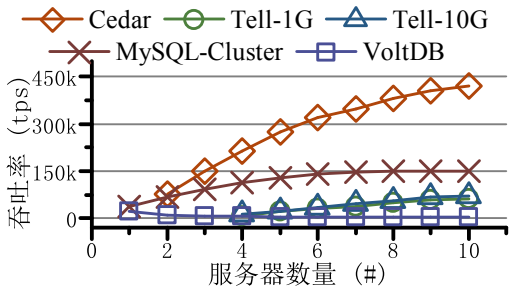


图 3.9: E-commerce：调整节点数量

了负载中 40% 的请求。因此，它们对 VoltDB 的性能产生了很大的影响。

图 3.8 测试了在处理 Smallbank 负载时，各个系统系统的横向扩展性。在该实验中，数据库中保存了 1M 的账户信息。在使用不同数量的服务器时，Cedar 均保持了最优的性能。其中，Cedar, Tell 和 MySQL-Cluster 的性能随着服务器数量的增加而成线性增长。另一方面，VoltDB 的吞吐率依然受到分布式事务处理的限制。相对于服务器数量的增加，它的性能没有显著地变化。

3.4.2.3 E-commerce 测试

第三个测试基准对应的是一个电商类的应用。它的负载是根据国内某个电商网站六个月用户请求历史生成的。该测试包含七张表和四种事务。应用中包含了两种角色的用户：买家和卖家。其中有四张表是与买家相关的：User，Cart，Favorite 以及 Order。这些表分别保存了买家的个人信息、购物车列表、收藏商品列表以及购物记录。另外三张表示与卖家相关的：Seller，Item 和 Stock。它们分别包含了卖家的信息、贩卖的商品以及商品库存信息。在无共享的分布式数据库中，买家相关的表可以采用 *user-id*（买家编号）进行水平分区；卖家相关的表可以采用 *seller-id*（卖家编号）进行水平分区。初始化状态下，Cedar 在 Memtable 中存储了 11M 条记录（共使用了 5GB 内存）；同时它在 SSTable 存储了 25M 条记录（共占用了 815GB 的磁盘空间）。在完成所有的实验后，Memtable 保存了 8.6GB 的数据，而 SSTable 保存了 881GB 的数据。

表 3.6 列出了 E-commerce 负载中包含的事务类型和事务比例。其中，OnClick

表 3.6: E-commerce 测试中各类事务的比例

事务类型	OnClick	AddCart	Purchase	AddFavorite
事务比例	88%	1%	6%	5%

表 3.7: 90 分位延迟，E-commerce 负载

延迟 (ms)	Cedar	Tell-1G	Tell-10G	MySQL-Cluster	VoltDB
OnClick	1	8	4	4	4
AddFavorite	2	12	5	6	47
AddCart	2	2	14	4	4
Purchase	4	12	4	6	49
Overall	1	8	4	4	19

请求产生的是只读事务，而其他的均为读-写事务。OnClick 会读取一个商品的相关信息，主要访问 Item 和 Stock 表。AddCart 会将一个商品添加到买家的购物车中，主要访问 User 和 Cart 表。AddFavorite 会将一个商品添加到买家的收藏夹，同时更新商品的收藏次数，主要访问 User ，Favorite ，Item 三张表。Purchase 创建一个买家订单，同时减少商品的库存数量，主要访问 User ，Order ，Item 和 Stock 表。

图 3.9 通过调整服务器的数量，验证了各个系统在处理 E-commerce 负载时的横向扩展性。Cedar 的吞吐率随着服务器数量的增加而线性提升。当使用 10 台服务器时，它的性能达到了 438k TPS。此时，它的吞吐率是其他系统最优性能的 3 倍以上。表 3.6 给出了各个类型事务的 90 分位延迟。可以看到，Cedar 能够在 1ms 内完成绝大多数事务的处理。这主要是因为 Cedar 对节点之间的远程数据交互进行了大量的优化。MySQL-Cluster 和 Tell 的性能同样随着服务器数量的增多而增加。但它们的事务延迟相对较高。VoltDB 在处理 AddFavorite 和 Purchase 类型事务时的延迟相对较高。这是因为这两类事务访问的表是根据不同的字段分区的。这些事务在执行过程中可能会访问多个分区的数据。此外，VoltDB 在处理分布式事务时会造成对单分区事务的阻塞。这导致了 OnClick 和 AddCart 的延迟也相对较高。

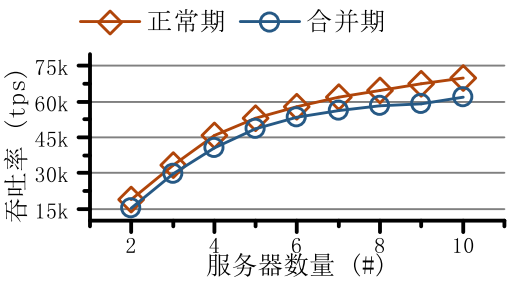


图 3.10: TPC-C：数据合并

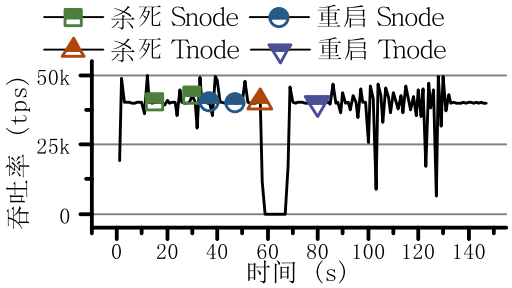


图 3.11: Cedar：节点故障期间吞吐率

3.4.2.4 数据合并和节点故障

在事务处理过程中，Cedar 可能会在后台启动数据合并操作，以减少 T-node 上的数据存储压力。图 3.10 给出了数据合并对事务吞吐率的影响。该实验采用了标准的 TPC-C 负载，并在数据库中生成了 200 个仓库的数据。如 3.10 所示，当 Cedar 使用的服务器数量少于 5 台时，数据合并操作对系统性能的影响很小。这是因为这些测试案例中，系统性能主要受 P-nodes 数量限制。而数据合并操作并不会影响 P-nodes 的运行。当 Cedar 的部署规模变大后，相对于非合并期间，系统吞吐率大约有 10% 的损失。这是因为当 P-nodes 数量增加后，T-node 处理能力对系统整体性能的影响变大了。而数据合并操作会消耗 T-node 的网络带宽和 CPU 资源。这使得 T-node 能够用于事务处理的硬件资源减少了，从而导致了吞吐率的减少。

图 3.11 的实验分析节点故障对 Cedar 性能的影响。本实验引入了 T-node 的多副本机制。这里，Cedar 使用三台服务器用于部署 T-nodes ，剩余的 7 台服务器用于部署 S-nodes 和 P-nodes 。其中，一个主 T-node 上保存了 Memtable 的主副本，另外两个保存了备副本。该实验使用的负载是 TPC-C ，并在数据库中生成了 200 个仓库的数据。在实验过程中，一些进程会被临时性的关闭。图 3.11 给出了 Cedar 在一段时间内的性能。

如图 3.11 所示，关闭两个 S-nodes 对性能没有产生影响。这是因为 SSTable 为每个 tablet 都保存了三个副本，P-nodes 可以读取任意一个副本获得所需的数据。此外，P-nodes 同时也在本地缓存了 SSTable 的数据。因此，失去两个 S-nodes 对性能没有产生大的影响。然后，当关闭了主 T-node 后，系统的吞吐率立刻下跌为 0。此

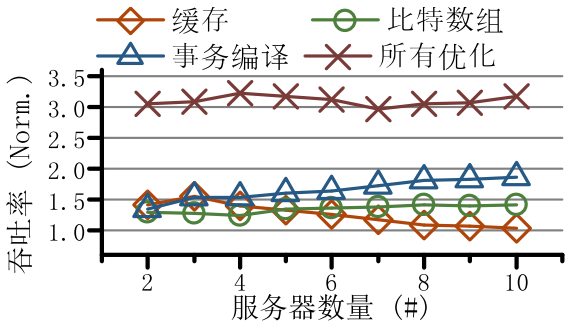


图 3.12: 不同访问优化下的性能提升

时，系统中没有任何 T-node 能够服务写入请求。在大约经过 7 秒后，一个备 T-node 成为了新的主节点，此时系统开始恢复服务。当先前故障的 T-node（记为 tn_1）重新加入集群后，新的主 T-node（记为 tn_2）会读取本地最新的 redo 日志，并将这些日志发送给正在恢复数据的 tn_1。由于这部分的代价，系统的性能会存在一定的抖动和下降。在经过大约 40 秒后，tn_1 完成与 tn_2 的数据同步。之后，系统的吞吐率恢复到正常水平。

3.4.2.5 访问优化

图 3.12 分析了不同数据访问优化策略对性能的影响。该实验首先测试了一个基线系统。该系统不启用任何的优化策略。在此基础上，本实验逐步引入不同的优化机制，并记录相对于基线系统的性能提升。图中，纵轴给出了使用不同优化策略（组合）后，Cedar 相对于基线系统性能的提升幅度。该实验采用了 TPC-C 基准测试。随着 Cedar 改变部署的 P-nodes 和 S-nodes 数量，不同优化策略在性能提升幅度方面表现出了不同的变化趋势。当 Cedar 使用的服务器数量增多后，SSTable 缓存的优化效果下降了。一方面，这是因为 S-nodes 增加后，系统的整体数据访问量上升了，降低了缓存的命中率。另一方面，当更多的 P-nodes 与 T-node 进行交互，T-node 访问对性能的影响变得更加重要。当启动事务编译后，单个事务中对 T-node 的多次访问会被合并。因此，在 P-nodes 数量增多后，该优化策略对吞吐率带来了更大的提升。异步比特数组对性能的提升作用相对稳定。它主要用于过滤一些不必要的 T-node 的访问。在开启所有的优化策略后，这些策略共带来了约 3 倍的吞

吐率提升。在不同的系统规模下，优化的幅度始终比较稳定。

3.5 本章小结

本章主要介绍了如何在分布式的日志合并树上实现高性能的事务处理。本章的工作主要构成了 Cedar 的事务处理模块。针对 Cedar 的架构特点，本章设计了新型的并发控制、系统恢复和数据合并算法；此外，提出了多种网络通信优化机制来进一步提升 Cedar 的事务处理性能；最后，基于多种基准测试的实验验证了 Cedar 相对于已有的分布式数据库系统具有更好的事务处理性能。

第四章　交互式事务处理的优化

Cedar 采用了内存事务引擎来保证高效的事务管理。内存事务处理的发展主要得益于近些年来内存容量的快速增长和价格的下降 [72, 73]。内存数据库系统将存储的所有数据组织在内存中 [11–13, 29] 。这使得事务处理的性能不再被缓慢的磁盘读写所限制。因此，数据库系统设计与实现的主要考虑因素发生了重大的变化。大量研究表明 [14, 73, 74]：在内存数据库中如何有效的提高多核 CPU 的利用率成为了最重要的考量。

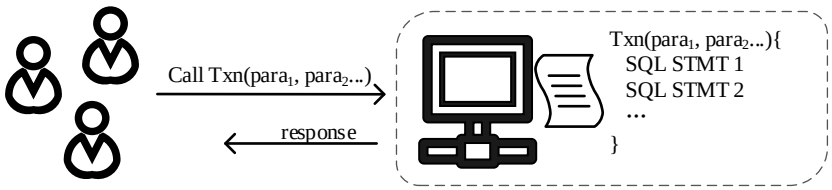


图 4.1: 存储过程式事务处理模型

为了改善 CPU 的使用，内存数据库系统通常假设存储过程式的事务执行模型（也称为 one-shot 模型 [1]，见图 4.1）。在该模型中，每笔的事务都被描述成一次存储过程调用。事务的执行逻辑被保存为数据库端的存储过程代码。客户端发送调用参数到数据库系统，而后者向前者返回事务成功或失败的结果。单笔事务执行过程中不会引入任何客户端和数据库系统之间的网络交互。在 one-shot 模型中，事务处理不需要考虑任何 I/O 阻塞问题（网络、磁盘）。所以，在没有任何访问冲突发生的情况下，这些事务在开始后可以一直在 CPU 上运行直到结束。基于这些特点，现有的研究 [15, 40, 75] 提出了很多轻量级的并发控制算法。它们均能非常高效地识别事务之间的访问冲突，产生的维护代价非常的小。另一方面，这些算法都采用回滚重试的方式来调度识别到的访问冲突。事实上，基于回滚重试的方式调用冲突会消耗掉较多的计算资源 [36, 62]。因此，这些方法的有效性主要建立在以下基础上：多数的存储过程式的事务执行时间非常短，相应地，它们之间极少发生冲突，

[1]One-shot 事务既可以采用存储过程执行，也可以描述为一段程序代码运行在嵌入式数据库上

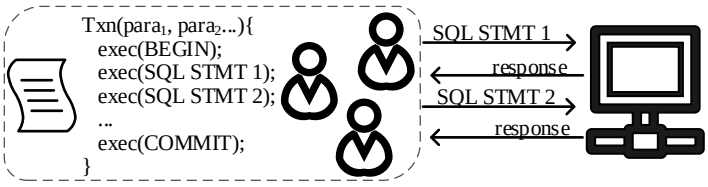


图 4.2: 交互式事务处理模型

即多数事务不会发生回滚重试。

在内存数据库中，尽管针对存储过程式事务的优化取得了极其良好的吞吐率提升，然而另外一种类型的事务处理目前还没有得到很多的关注，即交互式事务处理（见图 4.2）。在交互式事务处理中，上层应用是通过 SQLs 和 JDBC/ODBC 接口访问数据库的。执行一笔事务是将若干条 SQL 请求依次发送到数据库服务器。近期的一份调查 [76] 显示在多数应用中交互式的事务处理比存储过程式的事务处理运用的更加普遍。据报道：约有 54% 的受访者很少或者从不在数据库系统中使用存储过程。仅有 16% 的受访者表示有超过半数的事务是作为存储过程执行。导致这些情况的原因有以下方面：（1）存储过程很难维护和调试；（2）存储过程缺乏可移植性。不同数据库系统支持的存储过程语法并不相同，这使得开发人员很难在不同的平台和数据库上迁移或复用应用程序。

基于以上情况，本章主要考虑如何在内存数据库系统上优化交互式事务处理。相较于存储过程式的事务处理，交互式事务处理的主要不同点可以总结如下：1）事务会因为网络 I/O 而阻塞，这需要执行引擎能够有效地处理网络 I/O 导致的阻塞；2）由于事务的执行包含了多次网络交互，网络延迟会大大增长事务的持续时间，从而导致更多事务间的冲突。所以，针对交互式事务处理的并发控制策略不仅需要能够有效的识别冲突，还要有效的处理冲突。本章主要研究和设计内存数据库上的交互式事务处理，主要贡献可以总结如下：（1）分析了交互事务处理的主要难点，并给出若干重要的设计考虑；（2）设计了一种新型的事务执行模型，它能够有效的交替执行并发的事务请求，充分的利用多核 CPU 资源，并有效的处理各种类型导致的阻塞；（3）设计了一种轻量级的无锁（latch-free 的数据结构 [2] [77, 78]

[2]此处，锁（latch）是程序运行过程中用来保护被多线程并发操作的数据结构

来实现两阶段锁（two phase locking）[3]管理器 [79]；（4）在通用基准测试上的实验证实了：相对于已有方案，本章提出的技术在处理交互式事务时能够达到更好的事务吞吐率。

本章的内容组织如下：第一节主要分析交互式事务的性质并总结主要的设计考虑点；第二节给出了一种新的执行引擎，它能够有效地在多核硬件上并发执行事务请求；第三节设计了一种新的两阶段锁管理器并分析了它的正确性；第四节通过实验验证和分析本章提出的技术。

4.1 问题分析

本节首先分析了关于交互式事务的两个重要特点，然后给出在优化交互式事务处理方面的主要考虑。

频繁的网络 I/O 阻塞. 由于事务的执行是通过交互式的向数据库服务器发送 SQL 请求进行的，单笔事务处理会频繁的因为网络 I/O 而阻塞。这导致了，当一个 SQL 请求处理完成时，当前事务需要将自己挂起并在下一个请求达到后恢复。数据库系统需要能够有效的处理事务的挂起和恢复。

较长的事务持续时间. 事务的持续时间是指从它开始到最后完成提交的这段时间。考虑一个访问了数十条记录的存储过程式事务，它的处理阶段仅包含内存访问和 CPU 计算（这里不考虑提交阶段，提交需要向磁盘写入日志）。所以它的持续时间会非常的短。通常，这样的事务能够在 100us 以内完成处理阶段 [15]。相对而言，一个交互式事务的处理阶段包含了多次客户端和数据库系统之间的网络通信。同一个机房中，在两个使用以太网连接的服务器之间完成一次通讯来回大约需要 100us 左右。如果事务执行过程包含了 10 次交互，那么它的持续时间将在 1000us 以上。

鉴于这些特点，设计内存事务处理引擎应该遵循以下几项原则。

高效利用 CPU 的执行模型. 考虑事务 t_x 在 CPU 物理核心 c_i 上运行。如果 t_x 因为网络 I/O 阻塞了，c_i 将进入空闲状态。为了能够充分利用 CPU 的计算时间，在 t_x

[3]此处，锁（lock）是用于隔离并发事务的读写操作

下一个 SQL 请求进入系统前，c_i 应该继续处理另外一个事务 t_y。为了保证 CPU 资源的充分利用，数据库系统需要能够在相对少量的 CPU 物理核心上并发的处理相对大量的事务请求。已有的系统通常使用 Transaction-To-Thread 的执行模型来支持大量事务的并发处理。在该模型中，每个事务都会绑定一个线程，并由该线程来负责它的执行。当一个线程被阻塞了，相应的 CPU 核心将进入空闲状态，此时，另外一个线程会切换进来并在空闲的 CPU 核心上运行。这可以保证对 CPU 核心的充分利用。然而，该机制的主要缺点在于：每当线程因为网络 I/O 而阻塞时，操作系统会触发一次线程的上下文切换。而每次切换都会耗费一定的 CPU 计算资源。例如，在一个 *Intel E5-2620 CPU* 上，一次上下文切换通常需要耗费 8-13 us 的时间。而在交互式事务执行中会产生大量的网络通信。更多的网络通信会导致更多的 CPU 时间耗费在上下文切换上。这些额外的代价使得该执行模型不是很适用。所以，内存事务引擎需要一种新的执行模型。它能够非常有效的处理网络导致的阻塞。

基于锁的轻量级并发控制. 由于交互式事务的持续时间很长，这导致两个并发事务访问相同数据项的风险大大增加。所以，在这种类型的负载中，事务冲突也会发生得更加频繁。（i）许多内存数据库系统 [15, 40, 80] 偏向于采用乐观并发控制协议（OCC） [21]。但乐观的协议并不适用于调度交互式事务。OCC 在事务的处理阶段结束后增加了一个验证阶段来判定是否存在任何访问冲突。如果验证阶段失败了，客户端不得不从头开始重试整个事务。如上文所述，在交互式的负载中，访问冲突会发生得更加频繁。最终，OCC 会耗费大量的 CPU 计算资源和网络资源于重试失败的事务请求。（ii）基于以上分析，可以发现基于锁的并发控制协议（即：两阶段锁 2PL [79, 81]）在调度交互式事务方面是更好的选择。在基于锁的调度协议中，为了识别冲突，所有的事务在访问任意数据项前需要获取相应的锁。为了解决冲突，加锁失败的事务会被暂时的阻塞，直到它所需的锁被释放后再唤醒。该协议在调度冲突较多或者较少的负载时均能保持平稳的性能。此外，为了使其能高效地工作在内存数据库系统中，基于锁的并发控制协议需要一种轻量级的锁管理器来降低维护该协议产生的 CPU 代价。事实上，已有的实现方案不能满足以上需

求。（a）基于磁盘的数据库系统采用集中式锁表来实现两阶段锁管理器。集中式锁表将所有的数据的锁信息维护在一个哈希表中。但大量的研究发现集中式锁表存在很大维护代价 [82, 83]，并且会成为系统多核扩展性的主要瓶颈 [84, 85] 。（b）很多内存数据库系统实现了 行头锁 [40, 83] 来减少锁管理器产生的 CPU 代价 [86]。行头锁将记录的锁信息分散的保存在记录头部的若干字节中。但该机制只能有效的识别事务冲突。它没有涉及事务的阻塞和恢复机制。当一个事务加锁失败时，它只能（部分）回滚并重试，同时产生较多的 CPU 代价。所以行头锁并不适用于调度包含较多冲突的负载。

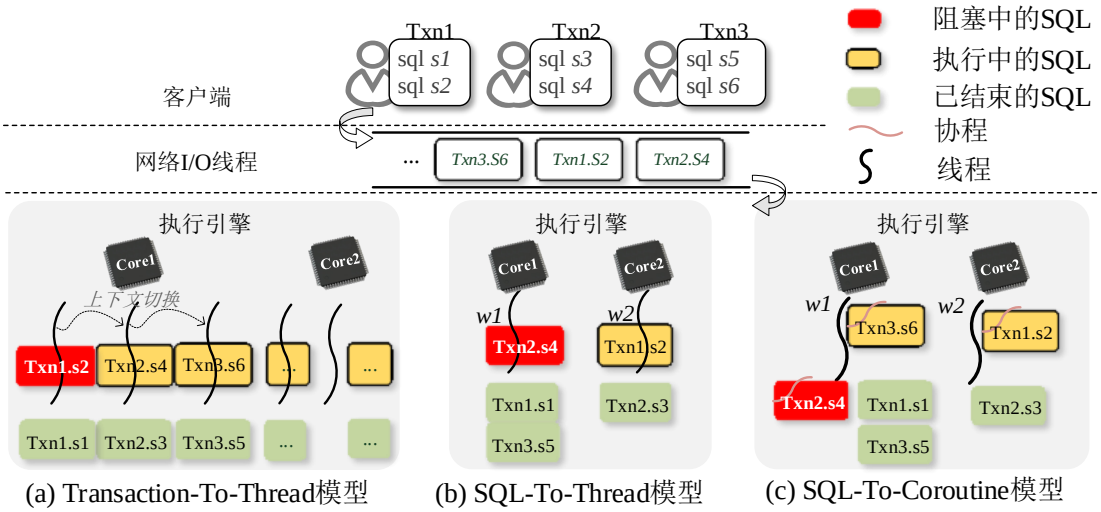
综上，交互式事务处理的优化需要（1）一个能够很好处理大量网络 I/O 阻塞的事务执行引擎；（2）一个能够高效地识别和解决冲突的轻量级锁管理器。

4.2 事务执行模型

上一节阐述了交互式事务处理会产生大量的 I/O 阻塞。这要求事务执行引擎能够以较低的 CPU 代价来调度这些阻塞。本节提出一种新的执行模型。它能够有效的处理多种因素（网络交互、加锁冲突）导致的执行阻塞。整体的优化目标是尽量减少事务执行阻塞产生的上下文切换。下文将首先介绍一种能够有效处理网络 I/O 阻塞的执行模型。基于该模型，本节将进一步讨论如何有效地处理冲突导致的阻塞，并提出一种基于协程的执行模型。

4.2.1 SQL-To-Thread 模型

为了更清晰地讨论，这里首先假设在事务执行过程中不存在冲突导致的阻塞。正如上一节论述的，Transaction-To-Thread 在处理交互式事务时不是一个很好的执行模型。这是因为每次线程在等待下一个 SQL 到达前，该线程会被阻塞并导致一次上下文切换。为了减少上下文切换，一种可行的设计是将不同的 SQL 请求绑定到不同线程上处理。由于客户端是按 SQL 请求的粒度与数据库服务器交互的，一个线程可以完整地执行一个 SQL 请求而不会因为网络 I/O 而阻塞。当一个 SQL 请



(a) Transaction-To-Thread模型 (b) SQL-To-Thread模型 (c) SQL-To-Coroutine模型

图 4.3: 不同种类的执行模型

求处理完成后，当前线程可以继续处理来自另外一个事务的 SQL 请求。这种模型被称之为 SQL-To-Thread（如图 4.3(b) 所示）。

在 SQL-To-Thread 模型中，一组 I/O 线程会专门负责与客户端的数据包交互。同时，数据库系统会创建一组工作线程来负责事务的执行。这里，工作线程的数量等同于可用的 CPU 核心数量。每当一个 SQL 请求达到时，一个 I/O 线程会将它添加到一个任务池中。每个工作线程会持续的尝试从任务池中获取 SQL 请求并执行。在一个请求完成处理后，对应的工作线程会通知一个 I/O 线程将处理的结果返回给客户端，然后，它会继续处理任务池中缓存的 SQL 请求。图 4.3(b) 给出了一个运行实例。两个工作线程 w_1，w_2 运行在两个独立的 CPU 核心上。这里，w_1 首先处理了事务 Txn1 的 SQL s1，然后处理 Txn2.s4。当 Txn1 的下一个 SQL s2 进入系统时，w_1 正在处理 Txn2.s4，而 w_2 处于可用状态。所以，w_2 会开始处理 Txn1.s2。在该模型中，每个线程通过持续处理不同事务的 SQL 请求来保持始终繁忙。它能够消除网络 I/O 导致上下文切换。

4.2.2 SQL-To-Coroutinue 模型

冲突导致的阻塞. 除了网络通信导致的阻塞外，事务的执行也可能因为加锁冲突而阻塞。上一节中介绍的 SQL-To-Thread 模型不能高效地处理这种类型的阻塞。考虑

表 4.1: 操作协程的接口函数

co-init	分配协程实例所需的内存资源，并初始化
co-resume	恢复给定协程实例的运行
co-yield	临时退出当前协程实例的运行

一个 SQL 请求 s_1 尝试获得数据锁 l_i，同时 l_i 已经被另外一个事务 t_y 持有了。此时，在 t_y 释放 l_i 之前，s_1 是无法继续执行的。面临这种情况，s_1 有两种选择：（1）部分回滚并稍后重试（仅回滚当前 SQL 请求 s_1 而不是整个事务）；（2）等待直到需要的锁被释放。在这两种方案中，第一种方案是不可接受的。这是因为交互式事务通常持续时间很长。在 t_y 结束并释放 l_i 前，s_1 极有可能会重试很多次。这些失败的重试将会造成 CPU 时间的大量浪费。因此，第二种方式更加适用于调度加锁导致的阻塞，即让 s_1 暂时挂起当前执行并等待 t_y 释放 l_i，以避免 s_1 的反复重试。

SQL 等待机制. 主要问题在于如何让一个 SQL 请求在线程上等待。当前主要存在两种通用的解决方式 [87]: 1. 忙等 (busy-waiting); 2. 条件等待 (condition-waiting)。在忙等机制中，SQL 请求所在的线程会不断的查看锁是否被释放了。显然，这种方式比回滚-重试的策略更加糟糕。因为它不仅浪费大量的 CPU 时间用于检查锁的状态，而且它会持续占据线程资源并阻塞其他可以正常执行的 SQL 请求。在条件等待的策略中，线程会暂时的挂起并放弃占有的 CPU 核心，它会等待 t_y 释放 l_i 之后再次被操作系统唤醒。这种机制的问题在于会导致 CPU 核心不能被充分利用。然而，上文说明了：为了避免线程切换的代价，每个 CPU 核心上只会启动一个线程。这导致了当线程挂起并放弃占有的 CPU 核心时，该核心将会闲置。

理想情况下，事务引擎能够按以下方式处理 SQL 请求。如果 s_1 因为加锁冲突阻塞了，那么线程会暂时地停止对 s_1 的执行，并开始处理下一个请求。在锁 l_i 被释放后，线程能够再次恢复 s_1 的执行。为了实现这种执行方式，本节提出了一种 SQL-To-Coroutine 的执行模型。表 4.1和图 4.4简要的说明了协程 [24] 使用方式和执行效果。如图 4.4（b）所示，一个调用者 A 使用 *co-init* 来分配协程实例 B 的内存资源，并对其进行初始化。然后调用者通过执行 *co-resume* 来启动 B 的执行。

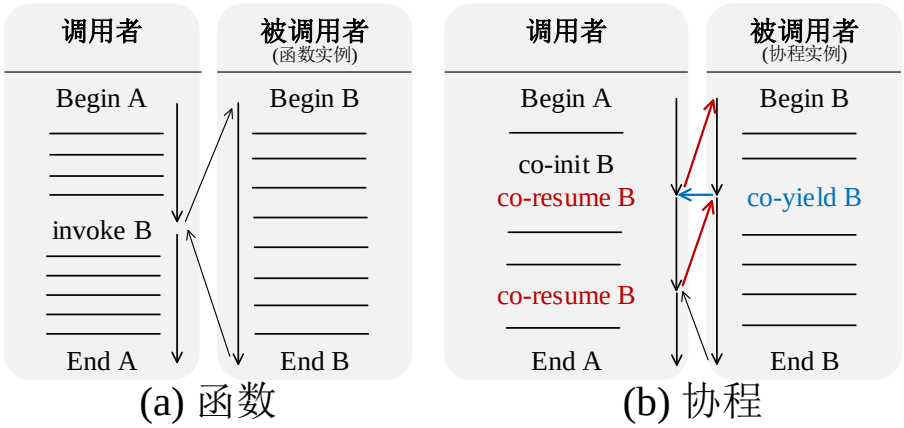


图 4.4: 协程调用的样例

在协程执行的任意时刻，B 能够调用 *co-yield* 来临时的退出当前的执行。在后台，*co-yield* 会将协程实例 B 的栈数据拷贝到由 *co-init* 分配的内存区域中。在处理一段其他程序逻辑后，A 可以再次调用 *co-resume*。此时，B 的栈数据将会被拷贝回线程的执行上下文中，然后 B 将从上一次 *co-yield* 被调用的位置继续执行。本质上，如果 *co-yield* 从不调用，那么协程实例和函数实例的行为是非常接近的。它们最主要的区别在于：协程实例在执行过程中可以从任意中间位置退出，然后从上次退出的位置恢复；在执行彻底结束前它会保存所有的临时变量和状态。关于协程更多的信息可以参见 [24]。

在 SQL-To-Coroutinue 中，每个 SQL 请求都是作为一个协程实例运行的。在执行过程中，如果没有发生加锁冲突，那么协程实例的执行方式和普通的函数调用是相同的。反之，如果发生了加锁冲突，那么线程可以调用 *co-yield* 来临时退出当前事务（SQL 请求）的执行。然后，线程启动一个新的协程实例来处理下一个 SQL 请求。在一段时间后，如果线程被通知冲突事务等待的锁被释放了，它可以调用 *co-yield* 来恢复被临时退出的协程实例。下一节将介绍如何向某个线程通知某个数据锁被释放了。

图 4.3 给出了将 SQL 请求作为协程执行的例子。在这个例子中，Txn2.s4 由于加锁冲突阻塞了。此时，线程 w_1 调用 *co-yield* 来临时退出 Txn2.s4 的执行。然后，w_1 开始继续处理任务池中的下一个请求 Txn3.s6。当 Txn2.s4 需要的锁被另外

一个事务释放之后，w_1 会调用 *co-resume* 来恢复 Txn2.s4 的执行。可以看到，在 SQL-To-Coroutinue 模型中，线程不会因为事务的加锁冲突而阻塞。

4.2.3 若干优化

将 SQL 请求作为协程执行同样会引入一些额外的代价。这些代价主要来自两个方面：协程的初始化和切换。

初始化. *co-init* 会分配 1MB 的空间用来保存协程实例执行的上下文。如果对每个 SQL 请求都需要调用一次 *co-init*，那么大量的 CPU 时间将会浪费在内存的分配和回收上。针对该问题，一个简单的优化方式是在每个线程上构建一个协程资源池。在开始处理一个 SQL 请求时，线程会尝试重用资源池中的协程实例。如果此时资源池是空的，那么线程会调用 *co-init* 来创建一个新的协程实例。在结束对一个 SQL 请求的处理后，线程会将该请求的协程实例放回到资源池中。另外一个问题是，使用协程是否会消耗大量的内存资源。显然，在每个工作线程上，当前正在被处理的 SQL 请求都需要分配一个协程实例。此外，每一个因为加锁冲突阻塞的 SQL 请求也都需要一个协程实例。所以，分配的协程数量主要取决于工作线程数量和被阻塞的 SQL 请求数量。事实上，这两部分数值都不会非常的大。例如，若给定一个数据库系统，它拥有 30 个 CPU 核心和 70 个加锁冲突阻塞的事务，那么在该系统中，所有的协程实例总共使用 100MB 的内存空间。而内存数据库系统通常至少配备了数百 GB 的内存。相对而言，所有协程总共耗费的内存资源很少。

协程切换. 一方面，线程调用 *co-yield* 来临时退出当前正在执行的协程实例。该函数会将协程的栈数据保存到一段预先分配的内存区域中。另一方面，线程调用 *co-resume* 来恢复一个之前被退出的协程实例。该函数会将协程的栈数据拷贝回线程的执行上下文中。那么问题在于这些内存拷贝是否会耗费大量的 CPU 时间。在一个 *Intel E5-2620 CPU* 上测试发现：完成一次协程的退出和恢复通常会耗费少于 50ns 的时间。相对而言，完成一次线程的上下文切换需要大约 10us 的时间。协程切换的代价是远小于线程切换的代价的。另一方面，在内存数据库中完成一次点

查询通常需要数十个微秒的时间。所以，协程切换的代价相对于 SQL 正常执行耗费的 CPU 时间而言是可以忽略不计的。最后，协程的切换仅发生在事务发生加锁冲突时。如果所有的 SQL 执行均没有发生访问冲突，那么协程切换完全不会发生。

本节重点介绍了 SQL-To-Coroutinue 执行模型是如何高效的处理网络交互和加锁冲突导致的事务阻塞。基于该执行模型，下文将进一步讨论如何高效且正确的调度并发事务。

4.3 锁管理器

第 4.1 节分析了交互式事务的并发执行更加容易产生冲突访问。这要求事务引擎需要能够提供高效的机制来识别和调度事务冲突。上文也分析了已有的集中式锁表和行头锁机制均存在较大的缺陷。因此，本节提出一种轻量级，无锁（latch-free）的两阶段锁（two phase locking）管理器 iLock。该结构在识别和解决事务冲突方面产生的 CPU 代价都极小。此外，无锁的设计保证了该结构的多核扩展性 [88]。本节将首先介绍设计 iLock 所需的数据结构，然后设计加锁和释放锁的算法，以及讨论如何避免死锁，最后论证 iLock 的正确性。

首先，为了实现 iLock，数据库中每条记录的头部保存以下信息：

- *lock-state*，锁状态变量，一个 64-bit 的变量，代表该记录被加锁的状态。
- *write-waiter*，写者等待信息，一个 64-bit 的变量，保存了哪些线程正在等待写该记录。
- *read-waiter*，读者等待信息，一个 64-bit 的变量，保存了哪些线程正在等待读该记录。

lock-state 的第一个比特位用于代表该记录所处的加锁模式，用 1 代表写锁模式；用 0 代表读锁模式。在写锁模式下，*lock-state* 的后 63 个比特位用于保存持有该记录写锁的事务的（唯一）标识符。在读锁模式下，*lock-state* 的后 63 个比特位用于保存持有该记录读锁的事务的数量。显然，如果 $lock\text{-}state = 0$（即，*lock-state* 的所有比特位均为 0），那么该记录没有被加锁。

write-waiter 的第 i 个比特位代表了在第 i 个工作线程上是否有任何事务正在尝试获得该记录的写锁。类似的，*read-waiter* 不同的比特位代表了是否有事务在等待获得读锁。事实上，如果系统中存在 N 个工作线程，那么 *write-waiter* 和 *read-waiter* 需要使用 N 个比特位。在 SQL-To-Coroutinue 模型中，变量 N 的取值取决于用于事务处理的 CPU 核心数量。由于当前的主流的商业服务器通常都配备了不超过 64 个 CPU 核心，所以 *write-waiter* 和 *read-waiter* 使用 64 个比特位对大多数情况而言都是足够了。此外，iLock 对 *write-waiter* 和 *read-waiter* 使用的比特位数量没有硬性的限制。在拥有更多物理核心的硬件平台上，它们完全可以使用更多的比特位来一一对应更多的工作线程。

在 iLock 中，每个事务拥有以下字段：

- *tid*，事务的唯一标识符
- *co-pointer*，当前使用的协程实例的对象指针

tid 是一个 63-bit 的正整数。注意，0 不会被用作事务的标识符。因为，标识符 0 将会被用于虚拟写锁机制中。后文将对该机制作进一步的讨论。*co-pointer* 的一个指针，指向了当前正在使用的协程实例地址。它只在事务的某个 SQL 请求因为加锁冲突而阻塞时使用。

每个工作线程维护了以下线程本地结构：

- *thd*，线程的唯一标识符，从 0 开始计数。
- *wait-map*，一个哈希表，用于组织当前线程上所有被阻塞的事务。
- *lock-queue*，缓存了已结束事务给予当前线程的锁。

wait-map 是一个哈希表。它允许多个键值对拥有相同的主键。在这个哈希表中，主键字段保存的是一个加锁请求，而数值字段保存了指向被阻塞事务的指针。当一个事务 t_x 的加锁请求 L_i 因为冲突而阻塞时，它会在 *wait-map* 中添加一个键值对 $< l_i, t_x >$。

lock-queue 缓存了已结束事务给予当前线程的锁。在事务 t_y 释放锁 l_i 之前，它会查看对应记录的 *write-waiter* 和 *read-waiter* 字段，判断是否有线程正在等待该记录的写锁或者读锁。若线程 w_j 上的事务 t_x 正在等待该记录的锁，事务 t_y 会将锁 l_i 添加到线程 w_j 的队列 *lock-queue* 中。之后，线程 w_j 会唤醒本地事务 t_x，让该事务获得这把锁，并继续执行。

4.3.1 锁的获得

这里将主要讨论如何识别冲突的加锁请求，并通过挂起冲突事务来解决冲突。下文将考虑如何处理以下情景：一个事务 t_x 尝试获得记录 r 的读锁或写锁。

识别冲突. 在获得某个记录 r 的锁时，（1）t_x 首先检查它的加锁请求是否与记录当前锁的状态 r.*lock-state* 相容。如果相容，那么（2）t_x 用一个新的状态更新 r.*lock-state*（针对不同的加锁模式，新状态的计算方式不同，下文将具体介绍计算的方式）。以上两个步骤需要**原子性**的完成。这主要是利用 compare-and-swap 指令实现的（算法中的 atomic-cas 函数）。该指令是硬件层面支持的同步原语 [4]。

算法 3 给出了获得写锁的伪代码。在该算法中，行 1 计算了一个新的锁状态。它的首个比特位为 1，后续的比特位保存的是 t_x.*tid*，即 t_x 的标识符。该状态代表了 t_x 持有记录的写锁。然后 行 2 检查当前的锁状态是否为 0（即记录 r 是否没有被任何事务加锁），并尝试原子性的将 r.*lock-state* 修改为 行 1 计算得到的新状态。

算法 4 给出了获得读锁的伪代码。在该算法中，（1）行 2-3 获得 r.*lock-state* 当前的快照，并检查该记录是否处于写锁模式下（即，锁状态的首个比特位为 0）。（2）如果该记录处于写锁模式下，此时事务 t_x 发生了加锁冲突；如果不是，那么 行 15 会对当前的锁状态快照增一，产生新的状态。（3）行 16 检查 r.*lock-state* 最新的状态是否与 行 2 获得的快照相同，并尝试原子性地将 r.*lock-state* 修改为 行 15 计算得到的新状态。（4）行 16 可能会发现 r.*lock-state* 的最新取值与之前获得的快照不同。这是因为 r.*lock-state* 可能被其他线程并发修改了。当这种情况发生时，加锁算法需要重试以上所有步骤。

[4] https://en.wikipedia.org/wiki/Compare-and-swap

算法 3: 获取写锁

```
输入: 记录 r, 事务 t_x
输出: 加锁成功或者失败
1  nstate ← t_x.tid| mask;
2  if 0 ≠ atomic-cas(r.lock-state, 0, nstate) then
3      changed ← atomic-set(r.write-waiter, thd) ;
4      atomic-synchronize();
5      if 0 = atomic-cas(r.lock-state, 0, t_x.tid) then
6          if changed then
7              atomic-unset(r.write-waiter, thd) ;
8          return locked ;
9      t_x.co-pointer← co-self() ;
10     wait-map.put(r.rid, write-mode, t_x) ;
11     co-yield(t_x.co-pointer) ;
12     if timed-out(t_x) then
13         return aborted ;
14     atomic-cas(r.lock-state, mask, nstate) ;
15 return locked ;
```

解决冲突. 如果 r.*lock-state* 与 t_x 的加锁请求处于冲突状态，t_x 通过以下步骤暂时挂起自身的执行。（1）*设置等待位.* 首先，需要将 r.*write-waiter*（或 r.*read-waiter*）的第 *thd*（当前线程的编号）个比特位设置为 1。它用于向持有锁的线程通知：第 *thd* 个线程上有事务正在等待该写锁（或读锁）。然后，需要增加一个内存栅栏（memory fence，即算法中的 atomic-synchronize 函数）来保证在后续步骤执行前，对 *write-waiter*（或 *read-waiter*）的修改已经写入内存了。（2）*重试加锁请求.* t_x 必须再次检查当前锁的状态，以防在首次加锁失败后该记录恰好又被解锁了。此时，如果 t_x 能够加锁成功，那么它继续正常的执行；否则，它将被真正的挂起。（3）*挂起事务执行.* t_x 获得它所处的协程实例地址，并将其保存在 *co-pointer* 中。然后，它会在线程的 *wait-map* 中添加一个键值对。其中，主键为冲突的加锁请求，值为 t_x 结构的内存地址。最后，通过调用 *co-yield* 来临时退出当前事务的执行。

在算法 3中，行 3-11 给出了挂起一个写锁请求的具体步骤。行 3-4 将 r.*write-waiter* 的第 *thd* 个比特位设为 1。行 5-6 尝试再次加锁。i）在加锁重试成功的情况下，如果在 行 3 确实将 r.*write-waiter* 的第 *thd* 个比特位从 0 改成了 1 ，行 7 会将该比特

算法 4: 获取读锁

```
   输入: 记录 r, 事务 t_x
   输出: 加锁成功或失败
 1 do
 2     ostate ← r.lock-state;
 3     if write-locked(ostate) then
 4         changed ← atomic-set(r.read-waiter, thd) ;
 5         atomic-synchronize();
 6         if write-locked(r.lock-state) then
 7             t_x.co-pointer← co-self() ;
 8             wait-map.put(r.rid, read-mode, t_x) ;
 9             co-yield(t_x.co-pointer) ;
10             if timed-out(t_x) then
11                 return aborted ;
12         else if changed then
13             atomic-unset(r.read-waiter, thd) ;
14         ostate ← r.lock-state;
15     nstate ← ostate + 1 ;
16 while ostate ≠ atomic-cas(r.lock-state, ostate, nstate);
17 return locked ;
```

位归零（从 1 改为 0）。这是因为该比特位初始状态为 0，在 t_x 成功获得锁后，当前线程上没有其他事务正在等待该记录的写锁，所以可以将它清零。ii）在加锁重试失败的情况下，行 9-11 将会暂时退出 t_x 的执行。

在算法 4中，行 4-14 给出了挂起一个读锁请求的具体步骤。行 4-5 将 r.*read-waiter* 的第 *thd* 个比特位设为 1。行 6 再次检查当前的锁状态，判断是否可以尝试获取读锁。如果依然无法获取读锁，行 7-9 将会暂时退出 t_x 的执行。否则，行 12-13 会重置 r.*read-waiter* 中第 *thd* 个比特位的取值。行 14-17 会重试加锁操作。

假设记录 r 处于读锁定状态，由于读锁和读锁之间是相容的，如果持续有事务在获得 r 的读锁，那么这可能导致写锁请求迟迟无法被满足，出现写锁饥饿。针对该问题，可以对算法 4的 行 3,6 中的 write-locked 函数进行了一定的调整。如果记录 r 处于写锁定状态，或存在被阻塞的写锁请求（即：r.*write-waiter* $\neq 0$），那么当前的读锁请求会被阻塞。

4.3.2 锁的释放

接下来将重点介绍如何释放事务持有的锁。事务释放锁的过程被分为两个步骤。第一步会根据具体的情况回退记录的锁状态 *lock-state*。第二步中，如果没有任何其它事务正持有记录的锁，那么当前事务会向一些特定的线程发送被该记录的读锁或者写锁，以使得它们能够唤醒本地被阻塞的事务。下文将考虑以下情景：事务 t_x 开始释放它在记录 r 上的锁。

算法 5: 释放写锁

```
   输入: 记录 r, 事务 t_x
1  ostate ← r.lock-state;
2  nstate ← 0 ;
3  if r.write-waiter ≠ 0 ∨ r.read-waiter ≠ 0 then
4      nstate ← mask;
5  atomic-cas(r.lock-state, ostate, nstate) ;
6  atomic-synchronize();
7  if nstate = 0 then
8      if r.write-waiter ≠ 0 ∨ r.read-waiter ≠ 0 then
9          if 0 = atomic-cas(r.lock-state, 0, mask) then
10             nstate ← mask;
11             atomic-synchronize();
12 if nstate = mask then
13     if r.read-waiter ≠ 0 then
14         send-read-lock(r) ;
15     else if r.write-waiter ≠ 0 then
16         send-write-lock(r) ;
```

回退锁状态. 如果多个事务正持有 r 的读锁，那么 t_x 会简单的对 r.*lock-state* 原子性减一。否则，t_x 是最后一个持有 r 上锁的事务。在这种情况下，解锁前需要检查是否有其他事务正处于阻塞等待的状态。（A.）存在阻塞事务。如果当前存在一个被阻塞的事务，那么 r.*lock-state* 会被回退到一个特殊状态 $mask = 0x80000000$。该状态的首个比特位为 1，其他比特位为 0。这是前文提及的*虚拟写锁*机制，即一个标识符为 0 的虚拟事务持有了该记录的写锁。注意：数字 0 不会用作正常事务的标识符。虚拟写锁是用来防止正在并行执行的其他事务*"偷窃"*刚被释放的锁，同

算法 6: 释放读锁

```
   输入: 记录 r, 事务 t_x
 1 do
 2     ostate ← r.lock-state;
 3     nstate ← ostate − 1 ;
 4     if nstate = 0 then
 5         if r.write-waiter ≠ 0 ∨ r.read-waiter ≠ 0 then
 6             nstate = mask;
 7 while ostate ≠ atomic-cas(r.lock-state, ostate, nstate);
 8 atomic-synchronize();
 9 if  nstate = 0 then
10     if r.write-waiter ≠ 0 ∨ r.read-waiter ≠ 0 then
11         if 0 = atomic-cas(r.lock-state, 0, mask) then
12             nstate ← mask;
13             atomic-synchronize();
14 if nstate = mask then
15     if r.write-waiter ≠ 0 then
16         send-write-lock(r) ;
17     else if r.read-waiter ≠ 0 then
18         send-read-lock(r) ;
```

时，保证被阻塞的事务能够首先获得被释放的锁。（B1.）不存在阻塞事务。如果没有任何事务正在等待访问记录 r，那么 t_x 会设置 r.*lock-state* $\leftarrow 0$ 真正的释放 r 上的锁。（B2.）释放锁后出现了新的阻塞事务。在真正释放 r 上的锁后，t_x 需要再一次检查 r.*write-waiter* 和 r.*read-waiter*。这是因为另一个事务 t_y 可能会在 t_x 第一次检查是否存在阻塞事务之后挂起。如果此时发现了新的阻塞事务，那么 t_x 将再次尝试获得虚拟写锁，并恰当的唤醒 t_y。

算法 5给出了释放写锁的具体步骤。首先，根据当下在记录 r 上是否存在阻塞事务， 行 1-6 会将 r.*lock-state* 的取值回退为 0 或者 *mask*。若记录 r 处于未加锁状态，行 7-11 再次检查 r 上的 *write-waiter* 和 *lock-state* 字段，判断是否出现了新的阻塞事务。如果存在阻塞事务，t_x 会尝试获得虚拟写锁。类似的，算法 6给出了释放读锁的具体步骤。首先，行 1-8 会根据 r 上是否有阻塞的加锁请求恰当地回退锁状态。行 9-13 再次检查 r.*write-waiter* 和 r.*read-waiter*，并在发现新的阻塞事务后尝试

获得虚拟写锁。在以上两个算法中，在修改记录的锁状态 *lock-state* 后，它们都需要添加内存栅栏来确保代码的实际执行顺序，确保顺序不会被 CPU 或者编译器打乱。

唤醒阻塞事务. 如果 r 处于虚拟写锁的保护中，那么解锁算法会尝试唤醒被阻塞的事务。算法 5的行 12-16 和算法6的 行 14-18 会用虚拟写锁来唤醒被阻塞的读锁请求或者写锁请求。

算法 7: 传递写锁

```
   输入: 记录 r
1  state ← r.write-waiter;
2  for i ← 1 ; i ≤ thd-num; ++i do
3      j ← (i + thd) mod thd-num;
4      if 0 ≠ (state & (1 ≪j)) then
5          lock-queue[j].push(rid, write-mode) ;
6          break ;
```

算法 7唤醒被阻塞的写锁请求。（1）转送锁。若某个线程上有事务正在等待获得 r 的写锁，那么 t_x 会将虚拟写锁转送给该线程。 行 2-4 从第 $(thd+1)$ 个比特位开始检查，尝试在 *write-waiter* 中找到第一个非 0 的比特位。假定第 j 个比特位是首个非 0 位。行 5 会将虚拟写锁添加到线程 j 的 *lock-queue* 中。（2）恢复事务执行。在线程 j 从 *lock-queue* 中获得记录 r 的虚拟写锁后，它会查看 *wait-map* 并找到事务 t_y，该事务正在等待获得 r 上的写锁。线程 j 在 t_x 的协程实例地址 t_y.*co-pointer* 上调用 *co-resume*，然后，t_y 会从算法 3的 行 12 继续执行。t_y 的这次加锁必然成功。它会直接将虚拟写锁替换为 t_y 的写锁。（3）归零等待位。若线程 j 上没有其他事务等待 r 的写锁，那么它会将 *write-waiter* 的第 j 个比特位清零。

算法 8唤醒被阻塞的读锁请求。它与唤醒写锁请求的步骤略有不同。因为它可以同时唤醒多个处于等待状态的读锁请求。在向其他线程转送读锁请求前，行 2 会计算当前有多少线程正在等待 r 的读锁 [5]。行 3-4 会为每个需要读锁的线程获得一个**虚拟读锁**。它主要用于保证：在每个线程将恢复所有本地阻塞的读锁请求前，记

[5]popcount 是一种高效的算法，能够快速的计算比特数组中的非 0 位数量

算法 8: 传递读锁

输入: 记录 r

```
1 state ← r.read-waiter;
2 reader-number ← popcount(state) ;
3 atomic-cas(r.lock-state, mask, read-number) ;
4 atomic-synchronize();
5 for i ← 0 ; i < thd-num; ++i do
6     if 0 ≠ (state & (1 ≪i)) then
7         lock-queue[i].push(rid, read-mode) ;
```

录 r 始终处于读锁定状态。当一个线程完成了对本地读锁请求的唤醒后，它会释放一次 *虚拟读锁*。行 5-7 会将虚拟读锁分发给到相应线程的 *lock-queue* 中。然后每个线程会恢复被阻塞的读锁请求。被恢复的事务会从算法 4的 行 10 继续运行。在完成对阻塞事务的恢复后，每个线程会将 *read-waiter* 中各自的比特位归零，并释放虚拟读锁。

4.3.3 死锁处理

当多个事务在互相等待对方结束并释放数据锁时，这些事务会陷入死锁状态。这里采用了超时回滚策略来防止死锁发生。该策略仅允许每个事务在限定的时间内等待锁释放。在超时后，事务会主动回滚。这种策略产生的维护代价很小，并且不会造成太多的不必要的回滚。它的主要缺陷在于不能很快的发现和处理死锁。事实上，死锁可以尽可能在应用层面解决。一个设计良好的应用不会产生大量的死锁。所以，iLock 选择超时回滚策略来处理死锁。在算法实现中，每个线程会周期性地检查是否有本地阻塞的事务超时了。当事务超时后，线程会调用 *co-resume* 恢复事务的执行。在算法 3和算法 4中，事务在恢复执行后立刻判定自身是否超时后。如果超时，该事务会立刻回滚。

除超时回滚外，还有一些其他的策略被用于解决死锁问题。例如，维护*等待图*或者使用*等待-死锁策略* [1]。等待图维护了事务之间的等待关系。如果事务 t_x 在等待 t_y 释放锁，那么图上会构建一条从 t_x 执行 t_y 的有向边。每次在等待图上添加有向边后需要判定是否会形成环路。出现环路意味着发生了死锁。此时需要杀死

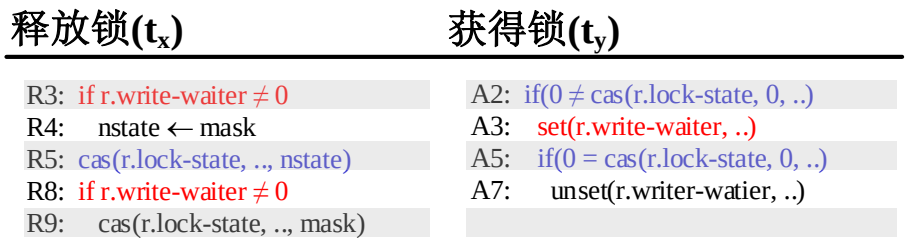


图 4.5: 加锁和解锁的关键代码段

环路上的某个事务来解决死锁。维护等待图的主要问题在于会产生大量的 CPU 维护开销。因此，iLock 没有采用这种策略。另一方面，等待-死锁策略对不同类型的加锁冲突采用不同的处理方式。假定事务 t_x 先于 t_y 开始，如果 t_x 需要 t_y 持有的锁，那么 t_x 可以等待 t_y 结束；反之，如果 t_y 需要 t_x 的锁，那么 t_y 会主动杀死自己。该策略仅允许先发事务等待后发事务持有的锁。由此避免了多个事务之间互相等待对方持有的锁，防止了死锁的发生。该策略的主要问题在于会造成大量不必要的事务回滚。即便 t_y 等待 t_x 结束不会造成死锁，该策略也会直接杀死 t_y。因此，iLock 同样没有采用等待-死亡策略。

4.3.4 正确性

显然，任意时刻两个冲突的加锁请求不会同时被满足。给定记录 r，如果 r 处于读锁定状态，那么写锁请求必然会被阻塞。因为在算法 3中，仅当锁状态满足 $r.\textit{lock-state} = 0$ 时，事务才可以尝试获得写锁。而在读锁定状态下，必然有 $r.\textit{lock-state} \neq 0$。如果 r 处于写锁定状态，那么读锁请求必然被阻塞。因为在算法 4中，仅当锁状态的首个比特位为 0 时，事务才可以尝试获得读锁。而在写锁定状态下，必然 *lock-state* 的首个比特位必然为 1。

另一方面，加锁和解锁算法需要保证并发的加锁和解锁操作能够准确执行。图 4.5假定了以下情景：一个事务 t_x 正在释放记录 r 上的写锁；并发地，另一个事务 t_y 正在尝试获得写锁。此时，算法必须保证在 t_x 结束后，t_y 必然能够获得写锁。表 4.2列举了交替执行加锁和解锁操作所有可能产生的调度。

- 情形 1，t_x 先释放了锁（代码 R5），然后 t_y 进行第一次加锁尝试（代码 A2）。

表 4.2: 所有可能的调度

调度	**结果**
(1) $R5 < A2$	(i) A2 获得记录 r 的锁
(2) $A2 < R5 < A5$	(i) 如果 $A3 < R3$，R4-5 会获得虚拟写锁 (ii) 如果 $R3 < A3 < R8$，R8-9 会获得虚拟写锁 或者 A5 获得记录 r 的锁 (iii) 如果 $R8 < A3$，A5 获得记录 r 的锁
(3) $A5 < R5$ （隐含了 $A3 < R8$）	(i) 如果 $A3 < R3$, R4-5 获得虚拟写锁 (ii) 如果 $R3 < A3$, R8-9 获得虚拟写锁 或者 A5 获得记录 r 的锁

由于写锁已经释放了，必然有 *lock-state* $= 0$，t_y 能够成功获得写锁。

- 情形 2，t_x 释放锁（代码 R5）发生在 t_y 的两次加锁尝试之间（代码 A2 和 A5）。此时，或者 t_y 在第二次加锁尝试中之间获得写锁；或者 t_x 发现存在被阻塞的事务，于是获得虚拟写锁并用于唤醒 t_y（代码 R8-9）。

- 情形 3，t_x 释放锁（代码 R5）发生在 t_y 第二次加锁尝试之后（代码 A5）。此时，t_x 必然会发现存在被阻塞的事务。它会获得虚拟写锁，并唤醒事务 t_y。

在所有的情况下，在 t_x 释放写锁之后，t_y 都能获得写锁。采用类似的分析方式同样可以证明读锁的获取/释放机制的正确性。

4.4 实验与分析

本节重点对比了 *iLock* 和基于协程的执行模型相对于已有解决方案的优势。下文首先介绍实验的环境和方法，然后通过不同的测试分析说明不同方法在处理交互式事务时的性能。

4.4.1 实验环境

本实验用 12,850 行 C++ 代码实现了一个内存数据库系统原型来验证本章提出方法的有效性。该原型包含了一个内存 B^+ 树，一个查询模型和一个并发控制模块。所有的实验都是在一个包含 5 个服务器的集群上完成的。每个服务器配备了两个 2.00 GHz 6 核心的 E5-2620 的 CPU 处理器，192GB 内存，服务器之间通过万兆网络连接。其中，四台服务器用于模拟客户端，一台服务器用于部署数据库系统。实验对比了以下三种不同的方法。

- *iLock* 使用 SQL-To-Coroutinue 执行模型，以及本章提出的新的锁管理器。
- *Row-Locking* 使用 SQL-To-Thread 执行模型，以及现有内存数据库常用的行头锁机制（row locking）。当发生加锁冲突时，事务会回滚重试加锁失败的 SQL 请求。*Row-Locking* 在 Hekaton [13] 的悲观事务调度机制中使用。
- *Lock-Table* 使用 Transaction-To-Thread 执行模型，以及一个集中式的锁表 [82]。这种机制在传统的磁盘数据库中广泛使用。

所有的方法都采用超时机制处理死锁。实验主要采用 TPC-C 和一个微型基准测试来验证不同方法的性能。实验中使用吞吐率（即每秒事务处理量，tps）来评估性能。TPC-C 使用标准的负载。微型基准测试使用一个表。该表包含两列：Key（64 位整型）和 Value（64 位整型）。表中默认存储了 1,000,000 条记录。测试中的每个事务请求会读取 N 条记录并且更新 M 条记录，总共 $N + M$ 个 SQL 请求。访问记录的主键是根据 Zipfian 分布选择的。

4.4.2 实验结果与分析

4.4.2.1 并发度

图 4.6的实验调整了并发的客户端数量来测试不同方法在 TPC-C 负载下的性能。在这组实验中，TPC-C 的数据库包含了 200 个仓库的数据。整体上，*iLock* 具有最好的性能。随着客户端数量的增量，它的吞吐率不断增长。*iLock* 的性能在并发

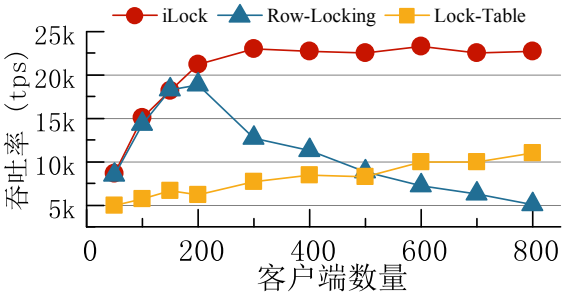


图 4.6: TPC-C: 调整并发度

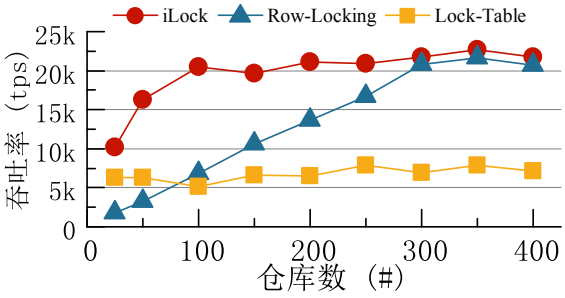


图 4.7: TPC-C: 调整仓库数量

300 个客户端的时候达到了峰值，约为 23k tps。然后，它的性能因为 CPU 达到了最大使用率而收敛。当并发少量客户端时，*Row-Locking* 的性能随着客户端数量的增加而增加。此时，它达到的吞吐率与 *iLock* 几乎相同。这主要是因为，当负载中产生的冲突比较少的时候，两种方法导致的 CPU 开销都非常的低。所以，它们都取得了很好的性能。随着客户端数量的不断增多，*Row-Locking* 的性能出现快速的下降。这是因为当并发的事务增多后，负载中产生了大量的访问冲突。而 *Row-Locking* 会浪费大量的 CPU 时间于重试失败的 SQL 请求。相对而言，*iLock* 表现出了更加高效的性能。当发生加锁冲突时，它会挂起事务的执行，直到所需的锁被释放为止。最后，*Lock-Table* 的性能随着客户端数量的增加而缓慢增长。它的峰值性能约为 10k tps。该方法性能比较差的原因主要是它花费了大量的时间在线程上下文的切换和维护代价高昂的集中式锁表。这导致它很容易耗尽 CPU 的计算能力。

4.4.2.2 冲突量

图 4.7的实验通过调整 TPC-C 数据库中仓库的数量来改变负载中的访问冲突。仓库数量少时，更多的客户端会访问同一个仓库，并产生更多的访问冲突。反之，仓库数量增多时，负载中的访问冲突会降低。在这组实验中，测试默认并发了 300 个客户端。图 4.7给出了实验结果。整体上，*iLock* 依然保持了最优的性能。在数据库中仓库数量较低时，*iLock* 的性能随着仓库数量的增多而增多。这是因为随着仓库数量的增多，负载中的冲突在减少，由于加锁冲突而阻塞的事务数量同样会减少。当仓库数量达到 100 后，*iLock* 的性能进入收敛阶段。此时，CPU 达到了非常

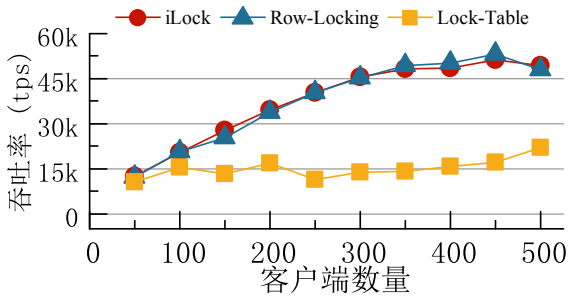


图 4.8: Micro: 10 次读取

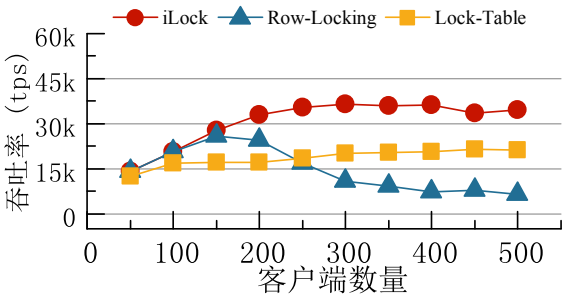


图 4.9: Micro: 5 次读取 +5 次写入

高的使用率，它限制了性能的进一步提升。*Row-Locking* 的性能随着仓库数量的增长呈线性增长。当数据库中的仓库数量较少时，负载中存在了大量的冲突访问。由于 *Row-Locking* 在调度冲突方面的能力较差并会产生较多的 CPU 开销，这导致了它的性能要远差于 *iLock* 的性能。当数据库中仓库数量增加时，负载中发生冲突访问的概率会下降，相应的，这会减少 *Row-Locking* 重试的 SQL 请求数量。此时，更多的 CPU 时间会用于有效的事务处理。因此，*Row-Locking* 的吞吐率提升了。它的吞吐率在仓库数量为 300 时达到了峰值。*Row-Locking* 的峰值性能与 *iLock* 非常接近。此时，它们的吞吐率均主要受限于 CPU 的计算能力。最后，*Lock-Table* 在不同的测试案例下始终保持了较低的，同时也较稳定的性能。通过增加仓库数量来降低负载中的冲突量对 *Lock-Table* 性能的影响很小。在不同的测试案例下，*Lock-Table* 的性能都主要受限于 CPU 的计算能力。*Lock-Table* 产生的维护代价更高，因此它能够达到的吞吐率较低。

4.4.2.3 读写比例

图 4.8和图 4.9使用微型测试验证各个方法在不同的读写比例下的性能。该实验的目的是为了分析各个方法在处理读锁请求和写锁请求时的能力。在测试中，Zipfian 分布的参数被设置为 $\theta = 0.6$。图 4.8的实验使用的是只读负载。其中，每个事务包含 10 次读取请求。图 4.9的实验使用的是写密集负载，其中，每个事务包含了 5 次读取和 5 次写入请求。（1）在只读负载下，*iLock* 和 *Row-Locking* 的性能几乎完全相同。这是因为只读负载中不包含任何访问冲突。这两个方法都能非常

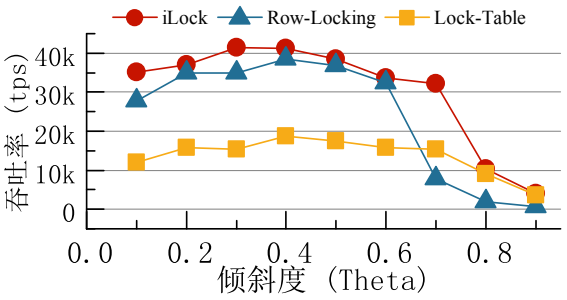


图 4.10: Micro: 调整访问倾斜度

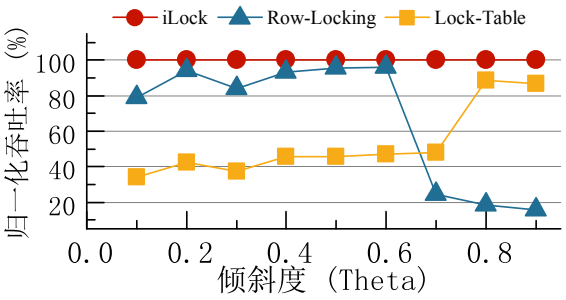
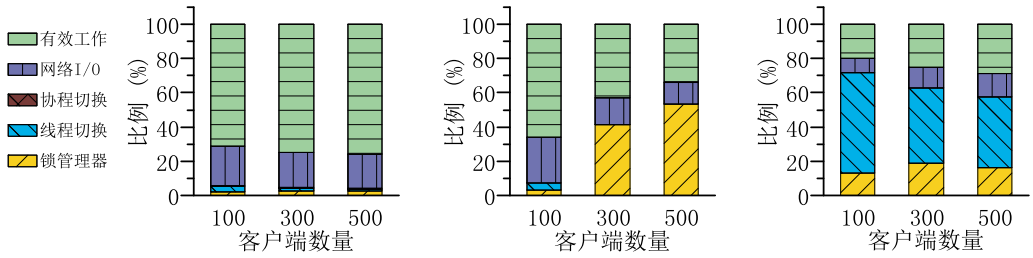


图 4.11: Micro: 调整访问倾斜度（归一化）

高效的调度无冲突的负载。当达到最大的 CPU 使用率时，它们的峰值吞吐率约为 45k tps。另一方面，在各个测试案例下，*Lock-Table* 的吞吐率始终维持在 15k tps 左右。它的性能仅为 *iLock* 的三分之一左右。这主要还是因为 *Lock-Table* 花费了较多的 CPU 时间在线程上下文切换和维护集中式锁表上。（2）在写密集的负载下，当同时并发 500 个事务时，*iLock* 的性能约是 *Row-Locking* 的 5.3 倍，是 *Lock-Table* 的 1.6 倍。随并发事务数量不断增加时，*iLock* 的性能最终收敛并稳定在 40k tps 左右。*Row-Locking* 的性能在并发度为 150 的时候达到峰值（约为 28 ktps），之后持续下降。以上性能趋势的原因主要是随着并发度增大后，负载中的访问冲突也随之增加了。因为 *iLock* 能够高效的处理冲突，所以性能保持稳定。此时，性能收敛的主要原因是：大量的冲突访问限制了实际的事务并发度。另一方面，*Row-Locking* 处理加锁冲突时会不断重试 SQL 请求，造成 CPU 的浪费，所以性能下降了。*Lock-Table* 在处理只读负载和写密集负载时表现出的性能很接近。虽然写密集的负载中会出现更多的访问冲突，限制实际可以达到的并发度，但这没有对 *Lock-Table* 的性能产生显著的影响。这也说明了 *Lock-Table* 的性能不是受限于事务并发度，而是受限于较高的 CPU 开销。

4.4.2.4 数据访问分布

本实验通过控制数据访问的分布来控制负载中产生的访问冲突。图 4.10 展示了调整 Zifpian 分布中的倾斜度参数 θ 对性能产生的影响。该实验使用的负载是图 4.9 中用到的写密集型负载。并发的客户端数量默认为 200。可以看到，所有方



(a) *iLock* (b) *Row-Locking* (c) *Lock-Table*

图 4.12: CPU 时间分析

法的性能都随着访问分布的倾斜度增大而减少。随着访问倾斜度的增大，部分“热”数据频繁被许多事务访问。这会导致负载中产生更多的访问冲突，造成更多的加锁阻塞，限制实际的并发度。观察图中的数据可以看到，当 $\theta < 0.6$ 时，负载中的访问冲突较少，各个方法的性能主要受限于它们对 CPU 计算资源的使用效率。当 $\theta > 0.6$ 时，负载中的冲突量变得不可忽视。此时，*Row-Locking* 的性能出现了快速的下降。该趋势说明了 *Row-Locking* 在解决冲突方面的不足。当 $\theta \geq 0.8$ 时，*iLock* 和 *Lock-Table* 的吞吐率非常接近。此时，负载中存在了大量的冲突。大量加锁请求被阻塞。实际被并发处理的请求数量较少。它们的吞吐率均受限于较低的事务并发度。图 4.11 是各个测试案例下，原始结果使用 *iLock* 的吞吐率归一化后得到的结果。它更加清楚的体现了各个方法间的区别。在处理冲突较少的负载时，诸如 *iLock* 和 *Row-Locking* 这类轻量级的机制能表现出很好的性能。在处理冲突较多的负载时，诸如 *iLock* 和 *Lock-Table* 这类能够高效地处理加锁冲突的锁管理会表现出更好的性能。总体上，*iLock* 在不同的负载下均能保持较好的吞吐率。

4.4.2.5 CPU 使用分析

以上实验从宏观的角度分析了不同方法的特点。图 4.12的实验对各个方法的微观行为进行了分析。这里主要使用了 VTune [6]来分析事务执行引擎将 CPU 时间主要用于了哪些模块。该实验使用的负载是 TPC-C。数据库中生成了 200 个仓库的数据。本实验通过调整并发的客户端数量来控制负载中的冲突率。观察图 4.12(a)可

[6]VTune 是 Intel 提供的 CPU 分析工具。

以发现：*iLock* 在锁管理器的维护、线程上下文切换以及协程切换方面耗费的 CPU 时间非常少。这意味着 *iLock* 能够非常高效地处理网络交互或者加锁冲突导致的阻塞。网络读写耗去了可观的 CPU 时间。这部分主要包含了对查询请求的反序列化、对查询结果的序列化，以及通知网络读写线程响应客户端。*iLock* 将大部分的 CPU 时间（约 70%）用于有效的事务处理工作。这部分主要包括了访问索引、写新记录等。图 4.12(b) 给出了 *Row-Locking* 在不同模块的 CPU 使用情况。当并发运行 100 个客户端时，负载中包含的冲突访问很少。*Row-Locking* 将大部分的 CPU 时间用于有效的事务执行。随着并发客户端数量的增加，负载中产生的访问冲突不断增加。*Row-Locking* 浪费了大量的 CPU 时间（约 41.3%-53.2%）在加锁上（例如，重试失败的加锁请求，检查记录的锁状态）。因此，它在处理冲突较多的负载时性能较差。图 4.12(c) 分析了 *Lock-Table* 的 CPU 使用情况。在不同的实验设置下，它最耗费 CPU 时间的部分始终是线程上下文切换（约 41%-58%）。这主要是因为它采用了 Transaction-To-Thread 的执行模型。处理每个 SQL 请求都会导致一次上下文切换。切换耗费的 CPU 时间甚至大于 SQL 处理耗费的时间。此外，集中式锁表的维护也耗费了较多的 CPU 时间（约 13%-16%）。因此，*Lock-Table* 性能较差的主要原因是大量的上下文切换以及维护锁表。

4.5 本章小结

本章主要研究了如何在内存数据库上处理交互式事务的问题，主要贡献包括：提出了一种 SQL-To-Coroutinue 执行模型来高效地处理不同类型的阻塞；提出了一种轻量级的锁管理器来有效地调度事务间的访问冲突。基于 TPC-C 和微型基准测试的实验证实了本章提出的方法能够有效地提升内存数据库系统处理交互式事务的能力。

第五章　精准数据访问优化

互联网的快速发展导致应用在数据库中保存的数据呈爆发式增长，数据规模达到了 TB 级别甚至 PB 级别。庞大的数据量导致了单台服务器无法有效地管理所有的数据。因此，这带来了分布式数据库系统的快速发展 [89]。为了更好的满足应用的数据存储需求，许多分布式存储系统都采用日志合并树 (LSM-Tree) 来组织数据。Cedar 同样采用了该结构来组织数据以支撑海量数据的管理需求。LSM-Tree 将所有的数据记录组织到多个不同的存储结构中：一个 Memtable 以及若干个 SSTable。其中，Memtable 是一个基于内存的存储结构，主要优化了数据写入的能力；而 SSTable 是基于磁盘的存储结构，以较低的成本提供更大的存储容量。在写入数据时，写入内容首先会被追加到 Memtable 中，然后周期性的批量转换为 SSTable 格式，并与已有的 SSTable 合并。相对于传统的存储机制（如：B-Tree）[90, 91]，LSM-Tree 提供了更好的写入能力。因此，它被多数分布式存储系统采用，例如 BigTable [7, 92] 和 Cassandra [17]。在这些系统中，Memtable 和 SSTable 被分别保存在内存和分布式文件系统中 [93] 。

然而，LSM-Tree 在一定程度上牺牲了数据读取的性能。在 LSM-Tree 中，一条数据记录可能在 Memtable 或者若干 SSTable 中保存了不同时期写入的多个版本。读取一条记录需要首先查询 Memtable ，然后查询多个 SSTable 直到获得记录的最新版本。事实上，在多次查询中，只有返回最新版本的那次查询是有意义的，对其它存储结构的查询都是不必要的。在分布式数据库中，读取 LSM-Tree 的问题变得更加的严重。许多分布式数据库（例如：Megastore [94] 和 Percolator [95]）会在存储层上构建分布式查询引擎来增加 SQL 或者事务功能。查询节点通过网络交互从存储节点中拉取数据。这直接导致：从分布式 LSM-Tree 存储引擎中读取一条数据记录会产生多次的网络通信。根据之前的分析，许多网络通信是不必要的，它们徒然地增加了数据访问的延迟。

基于对 LSM-Tree 上数据访问局限性的理解，现有工作提出了若干优化策略。例如，BigTable，LevelDB 等系统会周期性地合并 LSM-Tree 中的多个 SSTable [96]。这有利于减少在查询过程中需要访问的 SSTable 数量。另一方面， [7, 97, 98] 在 SSTable 上构建布隆过滤器 [99]。查询节点可以缓存这些布隆过滤器并通过这些结构来判定是否需要访问远程节点上的 SSTable。然而，这些技术都只能尽可能的减少对 SSTable 的访问。它们都不能判定一次查询请求是否需要访问 *Memtable* 。事实上，过滤对 Memtable 的无效访问是非常重要的。一方面，Memtable 负责了所有的数据写入，在 Memtable 上执行一次无效的数据读取造成了计算资源的浪费，降低数据写入的性能。另一方面，Memtable 仅包含了最近一小段时间内写入的数据。而数据库中大量记录是被保存在 SSTable 中。对这些记录而言，Memtable 上没有它们最新的写入内容，访问该结构是没有意义的。因此，存在较大比例的 Memtable 访问是不必要的。

针对访问分布式 LSM-Tree 存在的不足，本章提出一种精准数据访问技术。基于 LSM-Tree 的分布式数据库系统可以被抽象为图 5.1 所示的结构。存储层包含一个 Memtable 和若干个 SSTable；查询层包含若干个处理单元，简记为 P-node 。这些角色分布在不同的节点上。本章提出的精准数据访问技术使 P-node 准确地选择一个需要访问的节点，在多数情况下仅访问该节点即可返回正确的查询结果。由于 Memtable 的数据时刻在发生变化，P-node 很难在不进行远程通信的前提下确认 Memtable 没有所需的记录，因此，为 LSM-Tree 设计精准数据访问技术具有较高的挑战性。综上，本章工作包含了以下贡献：

- 设计了一种具有较低维护和同步代价的布隆过滤器，使得 P-node 能够快速的判定 Memtable 是否包含指定的数据记录，并在 Memtable 被更改后以较低的代价更新布隆过滤器。
- 设计了一种租约管理机制，使得当 Memtable 上写入新数据后，P-node 能够及时的更新本地缓存的布隆过滤器。该机制保证了使用精准数据访问策略能够保证 可线性化 *(linearizability)* [23] 的读写操作一致性。

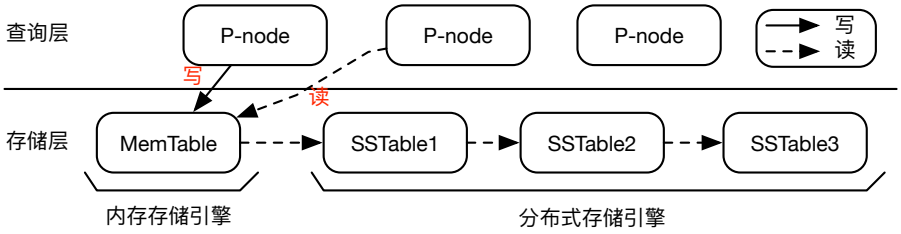


图 5.1: 分布式日志合并树上的数据组织与访问

- 在开源数据库上验证本章提出的策略，在处理 YCSB 的负载时读写性能整体提高了 6 倍以上。

本章内容按如下顺序组织：第一节主要回顾 LSM-Tree 的机制，并形式化的提出精准数据访问策略。第二节主要提出一种具有较低更新和同步代价的布隆过滤器。第三节提出一种基于租约 [100] 的分布式数据同步机制，保证在 P-node 能够及时更新本地的布隆过滤器以反应 Memtable 写入的最新数据。第四节给出在 YCSB 负载上的实现结果，验证该方案的有效性。

5.1 问题与分析

本节简要的回顾分布式系统中 LSM-Tree 的存储设计，并深入地分析它在数据访问方面的不足，最后，形式化地定义了分布式日志合并树上的精准数据访问。

5.1.1 存储模型

LSM-Tree 将存储的数据按照写入的时序划分为若干部分（见图 5.1）。其中，Memtable 对应了增量部分的数据。不同 SSTable 对应了不同时间段上写入数据库的内容，所有 SSTable 共同构成了基线数据。

Memtable 存储在内存中，服务对数据的读写请求。在 LSM-Tree 中，写入的数据只能追加到 Memtable 中。内存存储保证了高效的写入性能。另一方面，数据的组织通常采用针对内存访问优化的索引结构，例如 Bw-tree[101] 。处理插入 (insert) 请求时，新数据记录将直接写入到 Memtable 中；处理更新 (update) 请求时，记录的新

版本将会写入到 Memtable 中；处理删除（delete）请求时，对应的记录在 Memtable 中会被标记为删除。

由于 Memtable 是完全保存在内存中的，当其所在的进程或服务器故障时，数据可能会丢失。为了避免发生数据丢失，Memtable 为所有的写入操作生成 redo 日志，并在写入内容提交前将对应的日志记录持久化到磁盘中，即写前日志 (Write Ahead Logging) [60, 102, 103]。值得注意的是，任意写入操作产生的数据仅在对应的 redo 日志进入磁盘后才被提交，并可供后续的读取操作读取。由于现有的磁盘设备每秒仅能处理相对有限的读写数量 (IOPS)。对每条 redo 日志执行一次磁盘写入是非常昂贵的。因此，数据库系统通常使用成组提交 (group commit) [73, 104] 来优化日志的写入。该技术将一段时间内产生的日志记录缓存在内存中，然后通过一次写入操作将这些数据批量地持久化到磁盘中。

SSTable 存储在磁盘上，内容不可变，根据记录的主键大小顺序存储数据记录。SSTable 是通过冻结一个活跃的 Memtable 产生的。当一个 Memtable 达到一定的大小时，它会停止接受新的数据写入请求，在当前所有写入操作提交后转换为 SSTable 格式，并转移存储到其他节点的磁盘中。所以，SSTable 是一个只读结构。简而言之，LSM-Tree 首先将数据写入到 Memtable 中；然后当写入数据量达到一定规模后，将 Memtable 转换为 SSTable 并迁移到磁盘中。随着时间的推移，SSTable 会不断的产生。在图 5.1 中，SSTable-1 是最新冻结 Memtable 产生的，而 SSTable-3 是最早生成的结构。不同的 SSTable 包含了不同时间段上写入数据库的数据。

数据访问. 图 5.1 解释了查询单元 P-node 如何从分布式 LSM-Tree 中读取一条数据记录（假定主键为 k）。这里，P-node 需要依次访问 Memtable ，SSTable-1，SSTable-2 以及 SSTable-3 直到发现主键为 k 的记录为止。理想情况下，记录 k 存储在 Memtable 中，这时 P-node 仅需要一次远程数据访问。最坏情况下，记录 k 最后一次更新保存在 SSTable-3 中。这时，P-node 需要进行 4 次远程数据访问。目前，主要有以下两种技术来优化 LSM-Tree 上的数据访问。

数据访问优化. (i) 一种优化的思路是尽可能的将多个连续的 SSTable 合并。

这样可以整体上减少可能需要访问的 SSTable 数量。系统可以在后台读取多个 SSTable，为每条记录保留其最新的版本，并写入到一个新的 SSTable 中。在合并完成后，系统可以仅保留合并产生的新 SSTable 实例，删除所有被合并的旧实例。合并可以将 SSTable 的数量控制在一个较小的范围内，由此减少读取操作可能需要访问的 SSTable 数量。(ii) 另外一种思路是在 P-node 上缓存每个 SSTable 的布隆过滤器 [99] 。现有的研究显示布隆过滤器能够有效的判定一个集合是否包含某个成员。该结构保证 100% 的召回率，但判断结果存在较小比例的假阳性。如果某成员确实存在于集合中，布隆过滤器返回的判断结果必然为 *真*。通过查询 SSTable 的布隆过滤器可以判定某条记录在 SSTable 中是否存在。这可以正确且有效地过滤大量不必要的 SSTable 访问。由于 SSTable 是只读的结构，对应的布隆过滤器在创建后不再改变，因此，P-node 可以直接在本地缓存该结构，而不需要担心缓存内容的一致性问题。在尝试从 SSTable 读取记录 k 前，它可以首先查询对应的布隆过滤器，判定 k 是否存在。如果判定结果为不存在，此时 SSTable 上必然不包含记录 k，P-node 可以省略这次远程数据访问。

以上两种优化策略都致力于减少对 SSTable 的访问。然而，每个查询请求都不可避免的需要访问 Memtable 。因此，这里提出的精准查询技术主要研究的问题是如何过滤无效的 Memtable 访问。

5.1.2 问题定义

考虑分布式 LSM-Tree 上的一次读取操作 *Read(k)*（其中 k 为查询记录的主键取值）。直观的说，如果 P-node 能够准确的判定 Memtable 以及 SSTable 是否包含记录 k，那么它就可以直接从相应的结构中拉取数据，而不需要进行其他无用的网络交互。根据上文的分析，已有的工作能够有效的减少和过滤对 SSTable 的访问。因此，下文将主要分析：给定一个查询请求，如何判定是否需要访问 Memtable 。为了便于讨论，下文假设系统中存储了一个 Memtable 和一个 SSTable。

精准数据访问技术是：使一个 P-node 在不与存储节点通信的前提下，自主的

确定执行某个查询请求应该直接访问 Memtable 还是 SSTable。形式化的，该问题可以采用以下方式描述：

定义 5.1.1 (精准数据访问). 给定一个内容不断变化的结构 *Memtable* m，它包含记录的主键集合记为：$\mathcal{K}_m = \{k_1^m, k_2^m, \dots\}$，以及一个内容不变的结构 *SSTables*，它包含记录的主键集合记为：$\mathcal{K}_s = \{k_1^s, k_2^s, \dots\}$，节点 *P-node* 在尝试读取主键为 k 的记录 r 前，可以计算以下查询的结果：

$$q(k) = \begin{cases} m & k \in \mathcal{K}_m, \\ s & k \notin \mathcal{K}_m \bigwedge k \in \mathcal{K}_s, \\ none & else. \end{cases}$$

在完成以上计算后，P-node 能够直接访问 $q(k)$ 确定的存储服务器来获得记录 r。在上述公式中，计算 $k \in \mathcal{K}_s$ 是否成立是比较容易的。这可以通过在 P-node 上缓存 SSTable 的布隆过滤器实现。但是，计算 $k \in \mathcal{K}_m$ 是否成立是很困难的。核心问题在于判断在一个远程的动态变化集合中是否存在指定的元素。一个直观的思路是构建 Memtable 的布隆过滤器（记为 $\mathcal{B}_m$），并将该结构同步到 P-node 上。在处理读取请求时，P-node 查看本地的布隆过滤器来判断 Memtable 是否存储了所需的记录。这里存在两个问题。一方面，由于 Memtable 是存储在远程节点上，$\mathcal{B}_m$ 需要通过网络同步到 P-node 上。但该结构大小通常达到 MB 级别。通过网络频繁的同步该结构会很容易耗尽网络带宽。另一方面，由于 Memtable 在持续接受数据写入，它的内容随着时间变化，相应地，它的布隆过滤器 $\mathcal{B}_m$ 也会持续更新。这导致 P-node 在缓存了 $\mathcal{B}_m$ 的拷贝（记为 $\mathcal{B}'_m$）后，$\mathcal{B}_m$ 可能已经被更新了。此时，$\mathcal{B}'_m$ 的内容与 $\mathcal{B}_m$ 不再完全一致。如果 P-node 使用不一致的 $\mathcal{B}'_m$ 判断某个记录是否存在，这可能导致读取操作无法达到可线性化的操作一致性（即：读取操作没有返回最新的写入内容）。例 5.1.2 给出了一个例子来说明不一致读取发生的情景。

例 5.1.2. 在 *P-node* 获取了 $\mathcal{B}_m$ 的拷贝后，记录 r 被插入到 *Memtable* 中，并将 $\mathcal{B}_m$ 中对应的比特位从 0 改为 1。在下一次同步 $\mathcal{B}_m$ 前，$\mathcal{B}'_m$ 中对应的比特位仍为 0。此

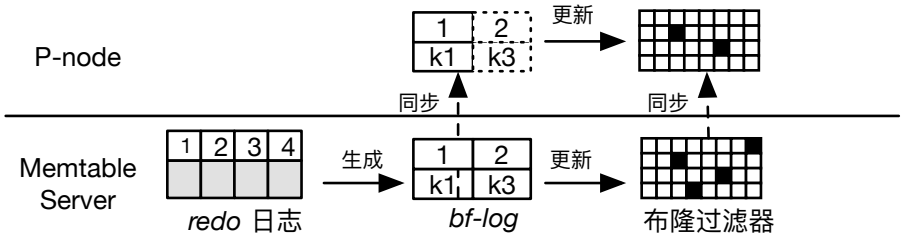


图 5.2: 布隆过滤器的更新与同步

时，*P-node* 尝试读取 r，检查 $\mathcal{B}'_m$ 会得到 r 不存在的判定结果。由此，*P-node* 跳过对 *Memtable* 的访问，最终，读取操作没有返回 r 在 *Memtable* 中的最新写入内容。

针对以上两个方面，第二节主要讨论如何高效的维护和同步 Memtable 的布隆过滤器。第三节主要给出一种基于租约的管理机制，使得 P-node 能够即时地更新本地缓存的布隆过滤器，确保读取操作能够达到可线性化的一致性。

5.2 远程数据访问过滤机制

本节主要介绍如何为 Memtable 构建一个布隆过滤器 $\mathcal{B}_m$，并降低对该结构的更新和同步代价。首先，一个值得注意的问题是：布隆过滤器不允许删除已经插入的元素。那么，当一条记录从数据库中删除时，应该如何恰当的更新 $\mathcal{B}_m$。事实上，在 LSM-Tree 上删除一条记录，并不是直接从 Memtable 或者 SSTable 将相应的记录删除；而是在 Memtable 中为被删除的记录添加一个 deleted 标记。当后续的操作访问该记录，并读取到 deleted 标记时即可知对应的记录逻辑上已经被删除了。因此，删除操作可以被视作一次特殊的更新操作。综上，Memtable 不会删除任何已存在的记录，因此使用布隆过滤器来保存 Memtable 包含的主键信息是可行的。

为 Memtable 构建布隆过滤器 $\mathcal{B}_m$，一种非常直接的方法是：对任意 Memtable 中存在的主键 $k_i^m \in \mathcal{K}_m$，都将 k_i^m 插入到 $\mathcal{B}_m$ 中。但这种方式会造成非常昂贵的更新和同步代价。不妨考虑一次插入操作 (insertion)。当一条记录被成功插入到 Memtable 中时，同时需要更新 $\mathcal{B}_m$ 中对应的哈希比特位为 1；进一步的，对 $\mathcal{B}_m$ 的更新需要同步到 P-node 上的拷贝（$\mathcal{B}'_m$）中。可以看到，任意插入操作都会引起

整个布隆过滤器的更新与同步。这会产生可观的 CPU 和网络代价。因此，本节提出一种更加有效的方式来构建 Memtable 的布隆过滤器。该策略能够完全消除插入 (insertion) 新数据带来的更新与同步代价，从而极大的降低 CPU 与网络开销。

5.2.1 方案概览

图 5.2 解释了布隆过滤器是如何更新和同步的。当 Memtable 上发生写入操作时，写入内容的 redo 日志会生成并持久化到磁盘中，以防在系统故障时发生数据丢失。系统在将一组 redo 日志写入磁盘前，会根据每一条 redo 日志构造对 $\mathcal{B}_m$ 的更新操作。这些更新操作相当于是 $\mathcal{B}_m$ 上的 *redo 日志*（记为 bf-log）。为了减少同步产生的网络开销，Memtable 服务器不是直接将 $\mathcal{B}_m$ 通过网络发到其他节点上的，而是将 bf-logs 通过网络传输到其他 P-nodes 上。而后，一个 P-node 通过执行（回放) 相同的 bf-logs 序列来同步本地缓存的布隆过滤器副本 $\mathcal{B}'_m$。

5.2.2 布隆过滤器的更新

整体上，Memtable 可能被以下写入操作修改：insertion，update，deletion。下文将主要介绍执行不同类型的操作是如何更新 $\mathcal{B}_m$ 的。这里假定被写入操作修改的数据记录为 r，它的主键为 k：

update. update 更新数据库中已经存在的记录。在该操作执行前，记录 r 可能处于两种存储状态：(i) $k \notin \mathcal{K}_m$，此时 r 仅存储在 SSTable 中。该操作会在 Memtable 中插入对 r 的修改。此时需要生成一条主键 k 的 bf-log，并在 $\mathcal{B}_m$ 中添加 k 的存在；(ii) $k \in \mathcal{K}_m$，那么在此之前有一个写入操作在 Memtable 中插入了该记录，并相应的更新了 $\mathcal{B}_m$。所以，本次操作不需要更新 $\mathcal{B}_m$。

deletion. deletion 删除数据库中已经存在的记录。上文已经说明了 LSM-Tree 是将 deletion 作为特殊的 update 处理的。该操作会为记录 r 添加 $deleted$ 标记，代表对应的记录在逻辑上已经被删除了。所以，deletion 操作的处理流程与 update 相同。在 deletion 执行前 (i) 如果 $k \notin \mathcal{K}_m$，那么 r 是首次写入到 Memtable 中。一条包含主键 k 和删除标记 $deleted$ 的记录被会插入到 Memtable 中。此时，k 依然需要被添

表 5.1: $\mathcal{B}_m$ 的更新规则

记录状态	insertion	update	deletion
$k \in \mathcal{K}_m$	×	×	×
$k \notin \mathcal{K}_m$	×	√	√

加到 $\mathcal{B}_m$ 中。这代表了 Memtable 中包含对该记录的最新“修改”（被删除）；(ii) 如果 $k \in \mathcal{K}_m$，那么已有操作在 Memtable 上添加了对 r 的修改，并且相应了更新了 $\mathcal{B}_m$ 的状态。所以本次操作不需要更新 $\mathcal{B}_m$。

insertion. insertion 向数据库中插入一条逻辑上不存在的记录。在该操作执行前，记录 r 可能处于以下状态：

(i) $k \in \mathcal{K}_m$，由于 r 逻辑上不存在，所以 Memtable 保存的是 r 的 $deleted$ 标记。所以，之前已经有 deletion 操作适当地更新了 $\mathcal{B}_m$ 的状态。本次操作不需要更新 $\mathcal{B}_m$；(ii) $k \notin \mathcal{K}_m \cap k \notin \mathcal{K}_s$，即 Memtable 和 SSTable 中都没有存储 r。一个 P-node 通过查询 SSTable 的布隆过滤器可以很容易的判定 $k \notin \mathcal{K}_s$ 是否成立。当 $k \notin \mathcal{K}_s$ 成立时，或者 r 存储在 Memtable 中；或者 r 逻辑上不存在。无论是哪种情况，P-node 访问 Memtable 就足够了，而不需要额外访问 SSTable。因此，可以添加一条数据访问规则来避免 insertion 操作对 $\mathcal{B}_m$ 的修改：*当 $k \notin \mathcal{K}_s$ 成立时，无论 $k \in \mathcal{K}_m$ 是否成立，P-node 读取记录 r 时必定访问 Memtable*。可以看到，在这种情况下，执行读取操作并不需要检查 $\mathcal{B}_m$ 来确定 $k \notin \mathcal{K}_m$ 的判定结果。所以，在这种情况下，insertion 不需要更新 $\mathcal{B}_m$ 的状态。

(iii) $k \notin \mathcal{K}_m \cap k \in \mathcal{K}_s$，这意味了 Memtable 中没有存储 r 的修改。另一方面，r 在逻辑上不存在，此时 SSTable 必然也没有存储 r 的数据（否则，若 r 存储在 SSTable 中，为了避免重复插入相同的记录，insertion 将无法执行）。因此，必然存在 $k \notin \mathcal{K}_s$。这与假设 $k \in \mathcal{K}_s$ 矛盾。这种情况不会出现。

综合以上三种情况，当 insertion 修改 Memtable 时，$\mathcal{B}_m$ 不需要被更新。

表 5.1 总结了不同的写入操作在何种情况下需要更新 $\mathcal{B}_m$。可以看到，仅当 update 和 deletion 操作修改一条在 Memtable 中不存在的记录时才需要更新 $\mathcal{B}_m$。

5.2.3 数据访问算法

本节将介绍基于上述布隆过滤器的数据访问算法。若主键 k 在 $\mathcal{B}_m$（$\mathcal{B}_s$）中所有对应的哈希比特位为 1 时，则记为 $k \in \mathcal{B}_m$（$k \in \mathcal{B}_s$）。这里 $\mathcal{B}_s$ 是 SSTable 的布隆过滤器，而 $\mathcal{B}_m$ 是根据第 5.2.2 节中的方法维护的布隆过滤器。为了便于讨论，此时假设布隆过滤器中没有假阳性的判定结果。假阳性的问题会在后文中讨论。根据主键 k 与 $\mathcal{B}_m$ 和 $\mathcal{B}_s$ 的关系，可以划分出以下四种情形：

1. 如果 $k \notin \mathcal{B}_m \cap k \notin \mathcal{B}_s$，那么或者记录 r 不存在，或者 r 刚被插入到 Memtable 中（注意：插入操作不会更新 $\mathcal{B}_m$）。无论具体是哪种情况，P-node 都直接访问 Memtable 。若实际上 r 不存在，P-node 访问 Memtable 是不必要的。但实际负载中不会包含大量访问不存在记录的读取请求。因此，这不会对性能造成大影响。

2. 如果 $k \notin \mathcal{B}_m \cap k \in \mathcal{B}_s$，那么 r 仅存储在 SSTable 中，在 Memtable 中没有任何修改。此时，P-node 访问 SSTable。

3. 如果 $k \in \mathcal{B}_m \cap k \in \mathcal{B}_s$，那么 r 先是存储在 SSTable 中，而后被更新并在 Memtable 中保存了新版本。此时，P-node 访问 Memtable 中保存的最新版本。

4. $k \in \mathcal{B}_m \cap k \notin \mathcal{B}_s$ 这种情景不会出现。这是因为当满足 $k \notin \mathcal{B}_s$ 时，记录 r 仅可能是被 insertion 操作添加到 Memtable 中的。而在 $\mathcal{B}_m$ 的更新规则中，执行 insertion 操作不会修改 $\mathcal{B}_m$。即便后续 update 或者 deletion 操作尝试修改 r，由于此时存在 $r \in \mathcal{K}_m$，这些操作也不会修改 $\mathcal{B}_m$。所以，当 $k \notin \mathcal{B}_s$，势必存在 $k \notin \mathcal{B}_m$。这与假设 $k \in \mathcal{B}_m$ 矛盾。因此，这种情景不会出现。

基于以上的讨论，表 5.2 总结了在布隆过滤器不同状态下，P-node 应该访问的存储结构。注意，此时假设布隆过滤器不存在假阳性。

针对假阳性的处理. 布隆过滤器允许假阳性发生。这里主要存在两种不同类型的假阳性：

表 5.2: 布隆过滤器的状态和记录存储的位置

$k \in \mathcal{B}_m$	$k \in \mathcal{B}_s$	存储位置	解释
0	0	Memtable	记录不存在，或者刚被插入到 Memtable 中
0	1	SSTable	记录存储在 SSTable 中
1	0	-	该状态不可能出现
1	1	Memtable	在 SSTable 和 Memtable 均存储了一个版本

- $\mathcal{B}_m$ 的假阳性是指：存在 $k \in \mathcal{B}_m$ ，但访问 Memtable 没有读取到记录 r。这意味着记录 r 仅可能存在于 SSTable 中。此时如果存在 $k \in \mathcal{B}_s$，那么 P-node 还需要进一步访问 SSTable。如果 $k \notin \mathcal{B}_s$，那么可以确定该记录不存在。

- $\mathcal{B}_s$ 的假阳性是指：存在 $k \in \mathcal{B}_s$ ，但访问 SSTable 没有读取到记录 r。这意味着记录 r 仅可能存在于 Memtable 中。此时如果 $k \in \mathcal{B}_m$ 成立，P-node 需要访问 Memtable；即便 $k \notin \mathcal{B}_m$ 成立，P-node 仍然需要访问 Memtable 。这是因为当 r 是由 insertion 操作写入 Memtable 的时，而该操作不会将 k 插入 $\mathcal{B}_m$ 中。

算法 9: 精准数据访问算法

输入: 1. 查询主键 k　2. Memtable m_1 和 $\mathcal{B}_m$　3. SSTable s_0 和 $\mathcal{B}_s$
输出: 记录 r

```
1  if k ∈ B_m or k ∉ B_s then
2      read r from m_1 ;
3      if k ∈ B_s then
4          alter = s_0;
5  else
6      read r from s_0 ;
7      alter = m_1;
8  if r is null then
       /* 再次访问以处理可能的假阳性 */
9      read r from alter;
10 return r;
```

精准数据访问算法. 根据以上的设计，算法 9 给出了具体的精准数据访问的伪代码。根据上文的分析，当且仅当 $k \in \mathcal{B}_m \cup k \notin \mathcal{B}_s$ 满足时，P-node 会首先尝试访问 Memtable 。所以，1）如果上述条件满足，那么 P-node 尝试从 Memtable 中读取记录 r（行 2-3）。如果此次访问没有读取到该记录，并且满足 $k \in \mathcal{B}_s$，那么 P-node

需要继续访问 SSTable 以应对 $\mathcal{B}_m$ 的假阳性（行 4-5 和 9-10）。2）如果条件不满足，那么 P-node 尝试从 SSTable 中读取记录（行 7）。如果没有读取到，P-node 继续访问 Memtable 以应对 $\mathcal{B}_s$ 的假阳性（行 8-10）。

5.2.4 同步机制

P-node 需要缓存 Memtable 和 SSTable 上维护的布隆过滤器。对于 SSTable 的布隆过滤器（$\mathcal{B}_s$），P-node 可以仅同步一次，并一直使用本地缓存直到对应的 SSTable 被合并删除。这主要是因为，SSTable 和 $\mathcal{B}_s$ 均是只读的结构，不会被改变。而针对 Memtable 维护的 $\mathcal{B}_m$，P-node 需要不断的更新本地缓存的副本 $\mathcal{B}'_m$ 以应对 $\mathcal{B}_m$ 和 Memtable 的变化。一种直接的同步方式是将 $\mathcal{B}_m$ 从 Memtable 所在的服务器通过网络直接发送到 P-node 上。通常情况下，为了编码百万级元素的存在情况，布隆过滤器需要使用若干 MBs 的存储空间。显然，即便在局域网内，频繁的发送如此庞大的结构也是不合适的。

轻量级的同步机制. 下文介绍一种轻量级的方式来同步 $\mathcal{B}_m$。它主要依赖于在 Memtable 和 P-node 之间发送布隆过滤器 $\mathcal{B}_m$ 上的增量修改（即本节概览里提及的 bf-log）。在 P-node 上，缓存的 $\mathcal{B}'_m$ 可以通过回放 bf-logs 来与数据源 $\mathcal{B}_m$ 同步。分析 $\mathcal{B}_m$ 的更新策略可以看到：仅有少量部分的操作会修改 $\mathcal{B}_m$ 并产生 bf-log。所以，实际产生的 bf-log 数量会远少于 redo 日志的数量（所有的写操作都会产生 redo 日志）。这意味着，同步 bf-log 带来的网络开销会远小于基于日志同步的数据库复制 [39, 105]。后者在提升分布式系统的容错性和可用性方面被广泛使用，其产生的网络开销在实际业务中也是可接受的。

实现方式. 根据生成的顺序，每个 bf-log 都被分配了一个单调递增的编号。该编号代表了 $\mathcal{B}_m$ 增量更新的顺序。在 Memtable 进程内，最近生成的 bf-log 被保存在一个环形缓冲区中。同步过程中，P-node 将目前收到最大的 bf-log 编号 N 发送给 Memtable 的进程。后者将所有编号大于 N 的 bf-log 返回给 P-node 。由于环形缓冲区只拥有有限的存储空间，一些最新的 bf-log 会覆盖缓冲区内部分最老的数

据。如果部分 P-node 需要的增量更新数据被覆盖了，Memtable 进程会向 P-node 返回完整的 $\mathcal{B}_m$。这种情况主要发生在一个 P-node 首次加入系统的时候。此时，它将尝试直接从 Memtable 进程拉取完整的布隆过滤器 $\mathcal{B}_m$。

5.3 分布式一致性维护

在经过一次同步后，$\mathcal{B}_m$ 可能会发生更新，导致和 P-node 上缓存的拷贝（$\mathcal{B}'_m$）出现差异。当 $\mathcal{B}'_m$ 的内容落后 $\mathcal{B}_m$ 很多时，可能某条记录在 Memtable 中存储了新的版本，但是 P-node 使用算法 9 访问该记录时会忽略掉 Memtable 中的版本，并从 SSTable 中返回一个老的版本。针对这个问题，本节讨论如何保证精准数据访问技术的强一致性，或称为 可线性化 (linearizability) [23] 。

5.3.1 可线性化

可线性化是一种一致性模型。它要求满足以下两点：（1）每个读写操作都存在一个 线性化时间点，位于操作开始和结束时间点之间；（2）每个操作等同于在它的线性化时间点上立刻执行完成。如果所有的读取操作都访问了 Memtable 和 SSTable，那么可线性化是很容易保证的。在这种情况下，读取操作的线性化点是它实际读取 Memtable 的时间点；写入操作的线性化点是在它实际提交对 Memtable 修改的时间点。此时，根据线性化时间点排序，所有的读写操作执行效果等同于按该次序依次执行，即：读取操作总是返回在它之前最后一次写入操作产生的修改。

然而，这里提出的精准数据访问技术改变了部分读取操作的执行方式。考虑一个读取操作 $R_x(r)$ 尝试返回记录 r 的最新写入内容，并在查看本地的 $\mathcal{B}'_m$ 后决定忽略对 Memtable 的访问。这里，$R_x(r)$ 的线性化时间点可以被设定为它查看 $\mathcal{B}'_m$ 的时机。在这个时间点上，(i) 如果 Memtable 没有存储记录 r 的任何版本，即在此之前没有提交过对 r 的写入操作，那么 R_x 仍然可以认为是可线性化的。因为，它仅访问 SSTable 就可以返回 r 的最新版本；(ii) 如果 Memtable 存储了 r 的修改，即在此之前某个写入操作执行了，此时，R_x 是不可线性化的。因此，它没有返回

Memtable 中保存的最新修改。总体上而言，当 R_x 查看 $\mathcal{B}'_m$ 时，r 在 Memtable 上不存在，那么 R_x 是可线性化的。在进一步讨论如何保证 R_x 的可线性化前，本节首先定义 Memtable 和一个布隆过滤器间的 *合理构建* 关系（见定义 5.3.1）。

定义 5.3.1. *对于 Memtable m 而言，当满足以下条件时，一个布隆过滤器 $\mathcal{B}$ 被认为是 合理构建 的：对 m 中的任意记录，它的主键 k 都根据表 5.1 给出的规则正确的添加（或未添加）到 $\mathcal{B}$ 中。*

事实上，在 R_x 的可线性化时间点上，如果对于 Memtable 而言，$\mathcal{B}'_m$ 是 *合理构建* 的，那么记录 r 必然不存在于 Memtable 中。该命题的正确性如下：

证明. 假设在读取操作 $R_x(r)$ 的可线性化时间点上，$\mathcal{B}'_m$ 对于 Memtable 是 *合理构建* 的，并且 r （主键为 k）存在于 Memtable 中。

由于 R_x 没有访问 Memtable，那么必然有 $k \notin \mathcal{B}'_m$ 并且 R_x 成功的从 SSTable 中读取到了该记录。由于 r 在 SSTable 的确存在，它只可能是通过 update 或者 deletion 操作在 Memtable *首次*产生了修改内容。根据 5.1 中给出的更新规则，这两个操作都会将 k 添加到 $\mathcal{B}'_m$ 中。但事实上 $k \notin \mathcal{B}'_m$。这意味着 $\mathcal{B}'_m$ 对于 Memtable 而言并不是 *合理构建* 的（与假设矛盾）。因此，假设不成立。原命题是正确的。 □

综上，如果 $R_x(r)$ 在精准数据访问算法中使用一个 *合理构建* 的 $\mathcal{B}'_m$ 省去了对 Memtable 的访问，那么 r 在 Memtable 中必然不存在，$R_x(r)$ 的执行是可线性化的。

显然，保证可线性化的关键是适当地更新 $\mathcal{B}'_m$ 的内容，使得在它被 P-node 使用期间，相对于远程的 Memtable 是 *合理构建* 的。下文将给出一种基于租约的机制来正确的同步 $\mathcal{B}'_m$。在引入租约机制前，下文将必要地分析 Memtable 和 $\mathcal{B}_m$ 在时序上是如何被写操作修改的。其中，图 5.3 被用于解释本节的内容。

5.3.2 成组提交

在 Memtable 上，为了提升日志写磁盘的性能，连续的写操作是成组提交的。一次成组提交按如下过程执行：

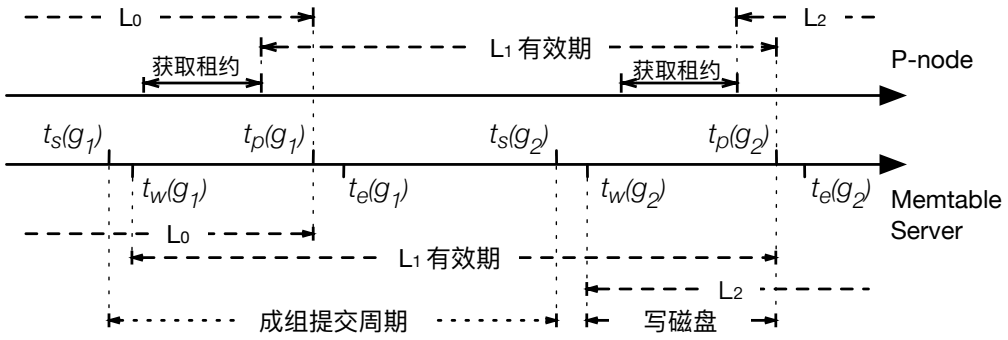


图 5.3: 成组提交与租约管理

1）日志生成。每个写操作都会产生 redo 日志，并将其缓存在内存中。后台有一个写盘线程会周期性的将内存缓冲区中的日志批量写入到磁盘中。这个周期称为**成组间隔**（例：图中 $t_s(g_1)$ 到 $t_s(g_2)$ 的一段时间）。

2）**开始阶段**始于一组 redo 日志已经生成了（起始时刻记为 $t_s(g_x)$）。该阶段会生成这组日志对应的 bf-log，并根据这些 bf-log 来修改 $\mathcal{B}_m$（例如，图中 $t_s(g_1)$ 到 $t_w(g_1)$ 的时间段）。该阶段终止时间记为 $t_w(g_x)$。之后，后台写盘线程会将这组 redo 日志持久化到磁盘中。

3) **写盘阶段**的启动时刻记为 $t_w(g_x)$。此时，写线程会将一组日志写入磁盘中（例：图中 $t_w(g_1)$ 到 $t_p(g_1)$ 的一段时间）。通常，对于通用的硬盘（HDD），该阶段需要花费若干个毫秒来完成写盘操作。

4）**发布阶段**的启动时刻记为 $t_p(g_x)$。它始于一组 redo 日志已经写入磁盘后。此时，本次成组提交包含的写操作产生的修改内容可以加入到 Memtable 中，并被后续读取操作访问了（例：图中 $t_p(g_1)$ 到 $t_e(g_1)$ 的时间段）。然后，本次成组提交完成。终止时刻记为 $t_e(g_x)$。数据库系统等待下一次成组提交的开始。

数据不变性. 在上述过程中，$\mathcal{B}_m$ 和 Memtable 在一段时间内都会保持不变。考虑两次连续的成组过程 g_1 和 g_2，$\mathcal{B}_m$ 从 $t_w(g_1)$（即已经用 g_1 的 bf-log 更新了 $\mathcal{B}_m$）到 $t_s(g_2)$（即尚未用 g_2 的 bf-log 更新 $\mathcal{B}_m$）这段时间内是保持不变的。而 Memtable 在 $t_e(g_1)$（即 g_1 的修改内容已经发布了）到 $t_p(g_2)$（即 g_2 的修改内容尚未发布）这段时间内是保持不变的。由于 Memtable 和 $\mathcal{B}_m$ 存在一段时间的不变性，这使得我

们有可能通过设计一种租约 [100] 机制来保证精准数据访问的强一致性。

5.3.3 租约管理

租约定义. 一份租约 L_x 是 Memtable 给出的保证，并由多个 P-node 持有。它包含了以下内容：一个不变的布隆过滤器 $\mathcal{B}'_m$（$\mathcal{B}_m$ 在某个时刻的版本）以及一个过期时间 t_x。它保证在 t_x 时间到达前，$\mathcal{B}'_m$ 对于远程的 Memtable 而言是 *合理构建* 的。所以，在租约给定的时间内，使用租约内给出的 $\mathcal{B}'_m$ 能够保证精准数据访问的可线性化性质。

一种直接的租约设计是：它在一次成组结束的时候开始，到下一次成组的发布阶段前结束。在这段时间内，Memtable 的内容是保持不变的。例如，在图 5.3 中，在 $t_e(g_1)$ 到 $t_p(g_2)$ 这段时间内，P-node 可以使用 $t_w(g_1)$ 时刻的 $\mathcal{B}_m$ 版本。显然，该版本的布隆过滤器对这段时间内 Memtable 而言是 *合理构建* 的。但是，这个策略的缺陷在于：每个租约都是在前一次租约结束后才开始。当 P-node 上的租约过期时，新的租约还没有开始。这就意味着，在 P-node 接收到新的租约前，它是无法执行精准数据访问的。这会使得本章设计的算法变得不够实用。因此，下文将设计一种更好的策略，使得连续的两个租约在时间上存在重叠。

租约设计. 租约可以始于一次成组提交刚更新了 $\mathcal{B}'_m$（即 $t_w(g_x)$ 时刻），并结束于下一次成组提交进入发布阶段前（即 $t_p(g_{x_1})$）。例如，在图中 L_1 开始于 $t_w(g_1)$ 并结束于 $t_p(g_2)$，并且它的布隆过滤器 $\mathcal{B}'_m$ 为 $t_w(g_1)$ 时刻的 $\mathcal{B}_m$。可以证明这样产生的租约也是正确的。

证明. 不妨假设 g_0 包含了 g_1 之前所有已提交的写入操作。在 $t_w(g_1)$ 到 $t_p(g_2)$ 之间，Memtable 存在两个版本，而 $\mathcal{B}'_m$ 由 g_1 和 g_0 生成的 bf-log 修改产生的。

1）在 $t_p(g_1)$ 之前，Memtable m_0 仅包含 g_0 提交的数据。此时，$\mathcal{B}'_m$ 对 m_0 而言是 *合理构建* 的，因为 $\mathcal{B}'_m$ 同时包含了 g_0 和 g_1 生成的 bf-log。对于 m_0 而言，$\mathcal{B}'_m$ 额外包含了 g_1 产生的 bf-log。但这不影响 $\mathcal{B}'_m$ 对 m_0 而言是 *合理构建* 的。

2）在 $t_p(g_1)$ 之后，Memtable m_1 包含了 g_0 和 g_1 提交的数据。此时，$\mathcal{B}'_m$ 恰好

是 g_0 和 g_1 结束后产生的。

在以上两种情况中，$\mathcal{B}'_m$ 对 Memtable 而言都是 *合理构建* 的。

□

在当前的设计中，两个连续的租约在时间上是重叠的。对于 P-node 而言，在它当前正在使用的租约过期前，新的租约已经可以获得。图 5.3 中，L_0 和 L_1 在 $t_w(g_1)$ 到 $t_p(g_1)$ 这段时间内是互相重叠的（即：g_1 在写盘阶段）。通常，这段时间持续 10 毫秒左右。对于 P-node 而言，在 L_0 彻底过期前，它有充分的时间从 Memtable 上获取新的租约 L_1。

值得注意的是：在两个租约重叠期间，它们包含的两种 $\mathcal{B}'_m$ 都可以在精准数据访问算法中使用。

5.3.4 租约实现

为了实现租约管理，在集群中需要使用 Precision Time Protocol (PTP) [1] 协议来同步各个节点的本地时钟。在局域网中，该协议能将时钟误差控制在 50 us 以内。整体上，租约管理按以下步骤进行：（1）P-node 发送一个租约申请请求到 Memtable 服务器尝试获得一份租约；（2）Memtable 服务器在接受到申请请求后，它会向 P-node 回复当前最新生成的租约以及所有用于更新 $\mathcal{B}'_m$ 所需的 bf-logs。（3）P-node 在接受到回复后回放 bf-logs 来更新本地维护的 $\mathcal{B}'_m$ 并更新租约的过期时间。（4）在当前租约快要过期时，P-node 再次发送新的租约申请请求。下文将重点讨论如何生成租约，如何获取租约，以及检查租约是否即将过期。

租约生成. 一份租约 L_x 在 $t_w(g_x)$ 时刻生成，它包含了当前最大的 bf-log 编号 N 以及过期时间 t_x。该租约中使用的 $\mathcal{B}'_m$ 可以通过回放所有编号不大于 N 的 bf-log 产生。它的过期时间 t_x 可以设置为 $t_p(g_{x+1})$。该时间点为下一个成组提交进入发布阶段的时刻。然而，该时间点不是提前可知。它仅能根据将 *当前时间*，*成组间隔*，以及 *写盘延迟* 三者相加来估算（例如，图 5.3 中的 L_1）。这里的估算忽略了部分本

[1] en.wikipedia.org/wiki/Precision_Time_Protocol

地处理时间（$t_s(g_x)$ 至 $t_w(g_x)$），这是因为这部分时间相对较短。成组间隔的数值由数据库系统的配置给出。写盘延迟可以由上一次成组写盘花费的时间估计。

提交等待. 由于 t_x 是估计产生的，它可能比 $t_p(g_{x+1})$ 的真实值更大，也可能更小。1) 如果 $t_x > t_p(g_{x+1})$，那么 Memtable 应该禁止 g_{x+1} 发布更新的修改内容。否则，由于 L_x 还没有过期，此时发布新的写入内容可能会导致不一致的读取。所以，g_{x+1} 的发布阶段需要阻塞到 t_x 时刻之后。该机制被称之为**提交等待**。2）如果 $t_x < t_p(g_{x+1})$，那么租约 L_x 的过期时间再在下一个成组提交进入发布阶段之前。此时，该租约不会阻塞下一个成组的发布阶段。发生提交等待会导致写入的延迟增长。为了避免这种现象的发生，我们倾向于使用写盘延迟估计值的下限来计算 t_x，以保证 $t_x < t_p(g_{x+1})$。值得注意的是：选择较小的 t_x 不会影响正确性，因为租约 L_x 包含的布隆过滤器在 $t_p(g_{x+1})$ 时刻达到前总是可用的。

租约获取. 在布隆过滤器的同步中，P-node 从 Memtable 服务器拉取一份租约以及所有编号在 $(N_1, N_2]$ 之间的 bf-logs（其中，N_1 是 P-node 目前已经收到过的最大 bf-log 编号，N_2 是新租约中指定的 bf-log 编号）。通常而言，当目前使用的租约在 400 us 内即将过期时，P-node 可以尝试从 Memtable 服务器申请一份新的租约。该数值的选择主要是基于局域网内节点间完成一次网络来回需要的时间。后者在千兆网络中通常需要 200 us。通过保证通信的时间短于租约过期的时间，P-node 能够在当前租约过期前完成同步和更新。

在算法 9 中，P-node 首先需要检查当前的租约是否达到了过期时间，来确定对应的布隆过滤器是否可以使用。这里存在一个问题，即节点之间的时钟是存在误差的。租约 L_x 的过期时间 t_x 是由 Memtable 服务指定的。在判断 L_x 是否过期时，t_x 应该与当前 Memtable 服务器上的系统时间 t_m 进行比较。由于该判断是由 P-node 完成的，所以需要由 P-node 来计算当前 t_m 取值的上限。事实上，P-node 可以计算 t_m 的上限为 $t_l + \delta$。这里 t_l 是 P-node 的本地系统时间，而 δ 是 PTP 协议下节点间时钟允许的最大差值。

5.4 实验与分析

本节重点对比精准数据访问算法和已有 LSM-Tree 上的数据访问机制的性能。下文首先介绍实验的环境和方法，然后通过不同的测试分析说明精准数据访问算法的优势和特点。

5.4.1 实验环境

实验在开源数据库 Cedar 上实现了本章设计的精准数据访问技术。该系统包含两层：查询处理层和存储层。查询处理层由若干 P-nodes 构成。在存储层中，Memtable 是存储在称之为 T-node 的节点上的，而 SSTable 按范围分区后分散存储在多个称之为 S-nodes 的节点上。所有实验均运行在一个包含 20 个节点的集群中。每个节点拥有两个 2.0 GHz 6-Core E5-2620 处理器，共 192GB 内存和 1TB 的硬盘。所有节点通过千兆网交换机连接。

实验使用 YCSB 基准测试。该测试被广泛用于验证分布式数据库系统的读写性能。在所有测试中，数据库中存储了 1,000,000 条记录。默认情况下，95% 的记录保存在 SSTable 中，而 5% 的记录保存在 Memtable 中。下文中，存在独立的实验分析数据的存储分布对性能产生的影响。由于本文提出的技术主要是优化数据读取的性能，因此，测试负载对客户端每秒发送的读取请求数量不作限制，而每秒发送的写入请求默认至多为 50,000 次/秒。最后，被访问的数据服从均匀分布。下文中，存在独立的实验分析倾斜的数据访问分布对性能产生的影响。实验主要对比三种不同的数据访问策略：

- NDA 使用简易的数据访问策略。P-node 首先尝试从 Memtable 读取数据，如果没有读取到记录，那么它继续访问 SSTable [18]。
- BDA 使用布隆过滤器避免不必要的 SSTable 访问 [97, 98]（详见 5.1）。
- PDA 为本章提出的精准数据访问技术。

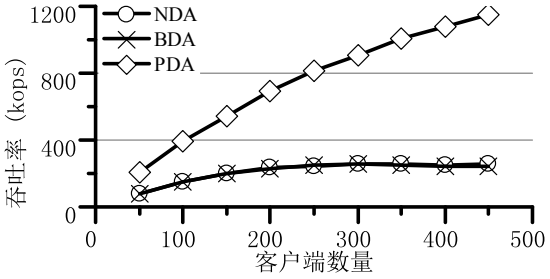


图 5.4: 并发度-只读负载

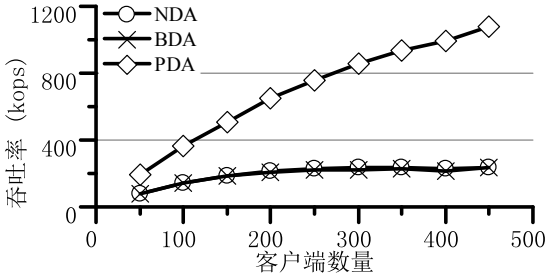


图 5.5: 并发度-每秒 50k 次写入

此外，因为 SSTable 是不可修改的，P-node 会缓存从 SSTable 读取到的数据。这样的缓存机制可以有效的减少对 SSTable 的远程访问。最后，实验中的布隆过滤器大小 2MB，并使用 4 个哈希函数。实验采用每秒处理的读取请求数量（operation per second, ops）来对比不同方法的性能。

5.4.2 实验结果与分析

5.4.2.1 并发度

图 5.4， 5.5， 5.6 给出了不同方法在不同的客户端并发量下的性能趋势。首先，图 5.4 给出了处理只读负载的性能。整体上，PDA 在所有的测试案例下均具有最好的性能。当连接的客户端数量为 450 时，它的吞吐率能够达到 1100k ops。该性能是 NDA 和 BDA 的六倍。NDA 和 BDA 的性能起初随着客户端数量增长而提升，并在 Memtable 服务器过载后收敛。对于这两种数据访问策略，Memtable 服务器更加容易成为性能瓶颈。这主要是因为所有的读取操作都会访问 Memtable 。另一方面，PDA 的性能随着客户端数量增加而持续上升，且不会受限于 Memtable 访问而进入瓶颈期。图 5.5 和 5.6 在 YCSB 负载中分别添加了每秒 50k 和 100k 次的写入请求。此时，NDA 和 BDA 的峰值性能随着负载中加入更多的写入操作而下降。该现象主要的原因是 NDA 和 BDA 性能受限于 Memtable 服务器有限的处理能力。而额外引入的写入请求同样主要是由 Memtable 服务器负责处理的，它们消耗了可观的计算资源。PDA 在各个测试案例中的性能也略微下降了。这是因为更多的记录被并发执行的写入操作添加到了 Memtable 中。这导致 PDA 实际上能够过

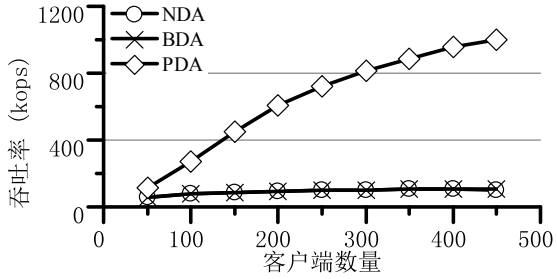


图 5.6: 并发度-每秒 100k 次写入

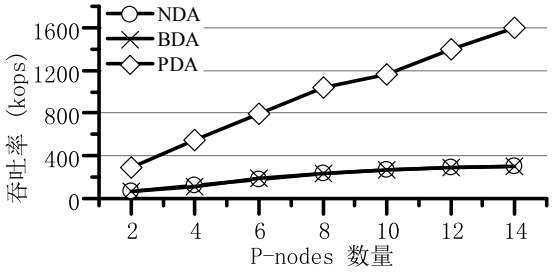


图 5.7: 扩展性-只读负载

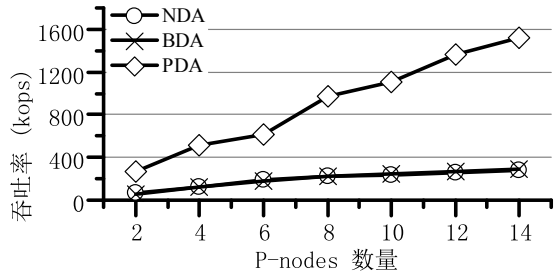


图 5.8: 扩展性-每秒 50k 次写入

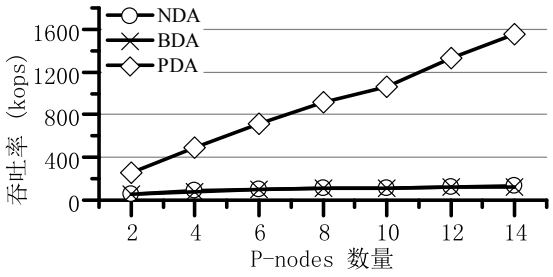


图 5.9: 扩展性-每秒 100k 次写入

滤的 Memtable 访问减少了。

此外，一个重要的现象是 NDA 和 PDA 在所有的测试案例（包括后续的实验）中都表现出了相近的性能。这主要是因为 P-node 上维护了对 SSTable 的缓存。在充分预热后，多数对 SSTable 的读取请求实际上访问的都是 P-node 的本地缓存。这极大的提升了访问 SSTable 的效率。因此，这使得基于布隆过滤器来减少 SSTable 访问所具有的优化效果大幅度下降了。

5.4.2.2 扩展性

图 5.7， 5.8， 5.9 给出不断增加 P-node 数量时各个方法的性能。每个测试案例展示了通过调整客户端数量后获得的最优性能。随着部署的 P-nodes 数量增加，PDA 的同步代价会随着增加。但 PDA 的性能依然随着 P-nodes 的数量呈线性增长的趋势。由此可见，相对于 PDA 过滤大量无效 Memtable 访问带来的收益，它引入的布隆过滤器同步和租约管理代价是可以忽略不计的。另一方面，NDA 和 BDA 在部署 10 个 P-nodes 的时候达到了性能的峰值。此时，Memtable 服务器的处理能力被大量无效的访问消耗了。图 5.7， 5.8 对应的测试中引入一定量的写入请求。由

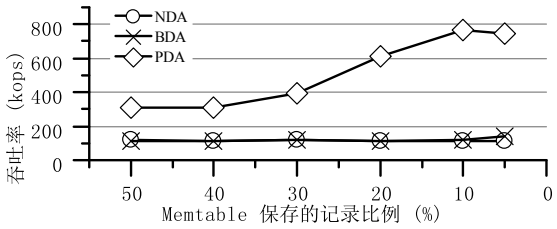


图 5.10: 存储分布对性能的影响

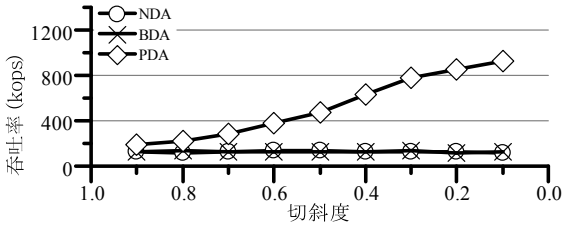


图 5.11: 访问分布对性能的影响

于 Memtable 服务器处理写入请求带来的开销，NDA 和 BDA 的性能均下降了，而 PDA 依然保持了稳定的性能。

5.4.2.3 存储分布

在之前的实验中，约 5% 的数据记录在 Memtable 中存储了最新的数据版本。图 5.10 显示了调整 Memtable 存储的记录比例对性能的影响。当 50% 的记录在 Memtable 存储了最新的版本，PDA 的吞吐率能够达到约 300k ops。随着该比率不断下降，PDA 的性能持续上升。相较而言，NDA 和 BDA 对这个参数不敏感。若一条记录在 Memtable 中保存了最新的版本。PDA 实际执行的数据访问和 BDA 是完全相同的。所以，当 Memtable 中存储的记录比例上升了，PDA 的性能会越来越接近于 NDA/BDA。但是，即便存在 50% 的记录需要从 Memtable 读取，PDA 依然保持了约 200% 的吞吐率提升。在实际的负载中，Memtable 不会保存数据库中较大比例的记录。当一个 Memtable 变大时，系统通常会冻结 Memtable 并将其转换为 SSTable 以减少内存资源的消耗。

5.4.2.4 访问分布

图 5.11的实验调整了负载中数据访问分布的倾斜程度。在 YCSB 中，数据请求的参数是通过 Zipfian 分布产生的。该分布通过参数 θ 来调整生成数据分布的倾斜程度。当 θ 越大时，分布越倾斜。在一个倾斜的分布中，部分“热”数据会被频繁的更新和读取，而一些“冷”数据则很少被访问。当访问分布非常倾斜时（$\theta = 0.9$），PDA 能够达到约 187k ops 的吞吐率，而 NDA/BDA 的性能约为 128k ops。此时，PDA 产生的性能提升相对不显著（约 1.46 倍）。这主要是因为在非常倾斜的访问分

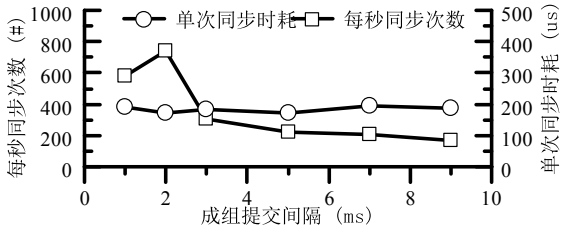


图 5.12: 网络同步代价

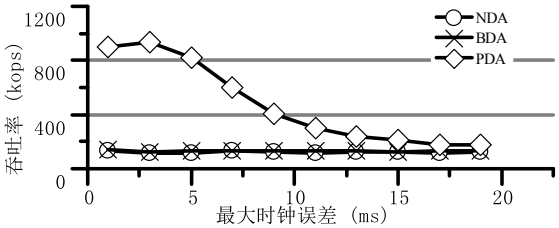


图 5.13: 时钟倾斜对性能的影响

布下，多数被频繁读取的记录同时也在被频繁的更新。这些记录需要从 Memtable 上读取。随着访问分布的倾斜度下降（减小 θ），PDA 的性能不断上升。当 $\theta = 0.1$ 时，PDA 的性能约为 921k ops。该数值约为 $\theta = 0.9$ 时的 4.93 倍。

5.4.2.5 同步代价

图 5.12 分析了使用 PDA 产生的布隆过滤器同步代价。P-node 会在一个租约即将过期时向 Memtable 服务器申请新的租约和对布隆过滤器的增量更新。在 Memtable 服务器一侧，它会根据成组间隔和写盘时间确定新的租约过期时间。租约的刷新频率主要受到成组间隔的影响。图 5.12 列出了在不同的成组间隔下触发租约同步的频率以及每次同步耗费的时间。首先，每次同步的延迟总是维持在 200 us 左右。这主要取决于 P-node 与 Memtable 服务器间进行一次网络通信来回的时间。可以看到，单次同步延迟是远小于成组的间隔的。其次，总体上同步触发的频率随着成组间隔的变长而变小。当 Memtable 按 2ms 进行一次成组提交时，每个 P-node 平均每秒会进行 700 次布隆过滤器和租约的同步。然后，随着成组间隔的不断变长，同步频率持续下降。在 9ms 的成组间隔下，每个 P-node 触发同步的频率仅约为 200。一个异常案例是使用 1ms 的成组间隔。在这个测试案例中，同步频率同样要小于使用 2ms 成组间隔的案例。这主要是因为，使用一个很小的成组间隔会导致频繁的写盘行为。然而，磁盘（HDD）通常仅支持有限次的每秒写盘次数。频繁的写盘导致平均写盘延迟上升了。最终，Memtable 根据更大的写盘延迟生成了持续时间更长的租约，从而降低了每秒租约同步的次数。

5.4.2.6 时钟倾斜

在精准数据访问中，所有的服务器都依赖于 PTP（Precision Time Protocol）来同步本地时钟。在同一机房中，它能够将服务器之间的时钟误差控制在微秒级内。如前文所述，时钟误差会影响每个 P-node 上 $\mathcal{B}'_m$ 的可以使用的时长。如果时钟误差较大，P-node 可能会过高的估计 Memtable 服务器的时间，并认为本地的租约已经过期了。此时，P-node 会停止使用精准数据访问算法。

图 5.13 分析了时钟误差对性能的影响。该实验模拟了两个服务器之间最大的时钟误差。为了实现以上模拟，我们首先假定所有服务器的时钟时间经过 PTP 同步后是完全相同的。当一个 P-node 获取本地时间时，系统会返回当前时间和一个随机数之和。随机数位于 $[-\delta, \delta]$ 间，是根据一个截尾正态分布生成。本实验逐步增大 δ 并分析它对性能的影响。当 δ 从 1ms 上升至 3ms 时，PDA 的吞吐率变化很小。这主要是因为该实验中成组提交每 5ms 发生一次。生成的租约时长在 5ms 以上。此时，在一份租约被认为过期前，P-node 能够获取到一份新的租约。随着误差进一步上升，吞吐率开始下降。这是因为 P-node 由于过高的估计 Memtable 服务器的时间，它会很快的认为当前租约已经过期了。而新的租约还没有生成。此时，精准数据访问算法将不能使用。即便在时钟误差很大的情况下（大约 20ms ），PDA 的吞吐率（约为 160k）仍略微高于 BDA（约 120k）或 NDA（约为 110k）。

5.5 本章小结

本章针对分布式 LSM-Tree 提出了一个精准数据查询技术。它主要是面向需要存储海量数据的应用。这些应用的数据库非常的庞大（TB 级别以上）。使用分布式 LSM-Tree 组织庞大的数据能够有效的提升写入性能。而本文提出的精准数据查询技术能够有效的提升分布式 LSM-Tree 上的读取性能。通过在分布式环境中同步若干代价低廉的结构，精准数据访问算法能够有效的过滤大量远程数据访问。实验证明本章提出的方法对性能有较大的提升。

第六章　总结与展望

互联网的快速发展带来了庞大的用户数量。这使得应用要求后台的数据库系统能够支撑更大规模的数据存储，并且提供更高吞吐量的事务处理能力。针对这两方面的问题，现有分布式数据库系统或者采用无共享的架构，或者采用了共享一切的架构。然而，前者要求上层业务是可以良好的分库分表的，并且负载中不会存在大量的分布式事务。后者通常依赖于高端的网络设备来保证节点之间快速的数据交换。根据当前的研究现状，我们可以得出以下结论：在不假设业务和负载类型的前提下，基于普通的网络设备和商用服务器，设计高性能的分布式事务型数据库系统依然是一大难点。

6.1　研究总结

基于业务的需求以及当前的研究现状，本文给出了一种新型的分布式事务型系统 Cedar。它不要求业务可以良好地分区，也不依赖于高端的硬件设备。因此，它能够适用于多数应用，并且以较低的部署成本上实现了较高的事务处理性能。它的主要特点是具有可横向扩展的数据存储能力，同时也拥有高性能的事务处理能力。可扩展的数据存储是基于分布式存储引擎和周期性地数据合并实现的。高性能的事务处理是基于内存事务引擎实现的。最后，系统使用了一个分布式查询层来负责 SQL 处理和事务执行，提供可横向扩展的计算能力。本文主要设计和优化了该系统的事务处理模块，核心贡献可以总结如下：

- 针对分布式日志合并树上的事务处理，本文设计了新型的并发控制、数据合并算法以及远程数据访问策略。本文为 Cedar 设计了多版本乐观并发控制协议以及基于写前日志的数据恢复策略。在事务模块的基础上，本文设计了无阻塞的数据合并算法。即便数据合并会持续地改变数据库中大量记录的存储位置，但执行该操作不会阻塞或中断任何并发的事务请求。最后，精细化

的远程数据访问策略有效地减少了事务执行过程中节点间的网络交互。基于 TPC-C 的基准测试显示：使用普通的商用服务器和以太网设备时，Cedar 相对于商用系统 MySQL-Cluster、VoltDB 以及原型系统 Tell 具有约 5 倍以上的吞吐率提升。

- 针对内存事务引擎上的交互式事务处理，本文设计了新型的事务执行引擎和两阶段锁管理器来减少执行和调度交互式事务产生的 *CPU* 代价。一方面，事务执行需要在客户端和服务器之间进行多次网络交互。本文设计一种基于协程的执行模型来有效地并发执行大量事务请求，并且尽可能减少事务并发和网络交互导致的 CPU 的阻塞或线程上下文切换。另一方面，网络交互产生的延迟大大增加了事务的持续时间，并引起更加频繁的事务冲突。本文设计了一个轻量级、多核可扩展的锁管理器来高效地识别和调度冲突的操作。实验验证了在不同的负载设置下，本文提出的方案始终优于现有的技术。

- 针对分布式日志合并树上的只读请求处理，本文提出了一种精准数据访问算法来减少处理只读请求产生的网络通信。该算法主要包括一种可快速更新和同步的布隆过滤器，以及一种分布式租约管理机制。前者能以较低的代价将一个写节点保存的记录集合信息同步到查询节点。查询节点访问该布隆过滤器来避免无效的远程数据访问。后者用于保证查询节点上使用的布隆过滤器不会错误地过滤必要的远程访问。它确保使用精准数据访问算法优化的只读请求依然能够达到强一致性 [23] 。实验结果显示：精准数据访问算法相对于已有策略能够实现约 6 倍以上的吞吐率提升。

6.2 未来展望

本文主要从事务处理和数据访问的角度优化了系统的处理能力。为了进一步完善系统的功能以及优化处理不同负载时的性能，后续可以探索的方向包括且不局限于以下方面：

1. **可串行化的隔离级别**。本文在分布式日志合并树的系统架构上设计和实现了快照隔离的事务隔离级别。完整的事务语义要求系统提供可串行化的隔离级别。未来工作需要考虑如何在 Cedar 的架构上支持事务的可串行化。实现该隔离级别面临的挑战主要包括：降低内存事务引擎上并发控制产生的 CPU 开销；减少提高隔离级别对事务并发度产生的影响；保证该功能在系统进行数据合并时的正确性。

2. **可扩展的写入能力**。Cedar 利用一个单点的内存事务引擎来保证高吞吐量的事务处理性能。随着负载压力的增大，系统向事务节点传输的数据量会增加。相应的，事务节点写入磁盘的日志量也会增加。当负载的峰值写入量增加时，事务节点的网络和磁盘写入能力会制约系统性能的提升。一种解决方案是为事务节点配置万兆以太网络和更好的硬盘设备。但这种方案不具备扩展性。因此，未来需要考虑如何让 Cedar 将提交的数据分散写入到多个节点上。

3. **高效的数据持久化**。尽管内存环境下的并发控制技术能够做的非常高效，但是基于写前日志的数据持久化和恢复技术依然需要操作低速的磁盘设备。这使故障恢复很容易成为系统性能瓶颈。当前，硬件厂商推出了非易失的内存存储设备（NVM） [106]。这些设备既能提供持久化的数据存储，也能保证接近内存访问的数据读写速度。它们是一种理想的支持数据库持久化功能的设备。未来可以考虑如何利用 NVM 来优化数据库系统的数据存储和故障恢复模块。

4. **高冲突下的交互式事务调度**。针对内存环境下的交互式事务处理，本文主要从执行模型和事务调度器两方面降低并发执行产生的 CPU 代价。但如果业务负载中存在较多的访问冲突时，事务处理实际的并发度会大大下降。此时，系统吞吐率的主要瓶颈会是较低的并发度。因此，未来需要考虑如何设计更好的并发控制协议来产生更好的事务调度计划，提升实际的并发度。

5. **数据合并优化**。Cedar 采用了分布式日志合并树来组织数据。该结构需要周

期性地对数据进行合并。当前，数据合并通常是在业务低谷的时候开始，并在经过一段较长的时间后才会结束。在此期间，系统的性能会存在一定程度的下降。因此，未来需要考虑如何加速合并的完成。一种思路是在非合并期间将增量写入分发到分布式存储引擎中，由此减少合并期间的任务量。

6. **范围读取优化**。最后，数据库系统除了支撑事务处理外，还要提供高效的数据查询功能。然而，在分布式日志合并树完成一次范围查询的代价相对较高。这主要是因为，查询请求需要同时读取分布式存储引擎上的历史快照，然后读取内存引擎中的增量修改，并将两部分数据合并。在日志合并树上完成一次范围读取的时耗会高于在 B 树 [90] 上完成相同操作。在这种存储结构上，范围数据读取的低效性容易成为整个查询处理的性能瓶颈。因此，未来需要强化日志合并树上的范围数据读取能力。一种思路是在分布式存储引擎上采用列式存储来优化历史快照的读取速度。另外一种思路是设计类似于本文提出的精准数据访问策略，来过滤部分无效的范围访问。

参考文献

[1] GRAY J, REUTER A. Transaction processing: Concepts and techniques[M]. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[2] WEIKUM G, VOSSEN G. Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery[M]. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[3] 中国互联网信息中心. 网络用户数量[EB/OL]. 2018. https://www.cnnic.net.cn/hlwfzyj/.

[4] 新华社. 春运售票 12 日迎首个高峰 12306 网站全天点击量 720 亿次[EB/OL]. 2018. http://www.xinhuanet.com/politics/2018-01/15/c_129791335.htm.

[5] 邬贺铨. 大数据时代的机遇与挑战[EB/OL]. 2013. http://www.qstheory.cn/zxdk/2013/201304/201302/t20130207_210899.htm.

[6] IDC. Data Age 2025[EB/OL]. 2017. https://www.seagate.com/cn/zh/our-story/data-age-2025/.

[7] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. TOCS, 2008, 26(2): 4.

[8] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally-distributed database[C]//OSDI. [S.l.: s.n.], 2012: 261–264.

[9] STONEBRAKER M, MADDEN S, ABADI D J, et al. The end of an architectural era:(it's time for a complete rewrite)[C]//PVLDB. [S.l.]: VLDB Endowment, 2007: 1150–1160.

[10] THOMSON A, DIAMOND T, WENG S C, et al. Calvin: fast distributed transactions for partitioned database systems[C]//SIGMOD. [S.l.]: ACM, 2012: 1–12.

[11] FÄRBER F, CHA S K, PRIMSCH J, et al. Sap hana database: data management for modern business applications[J]. SIGMOD, 2012, 40(4): 45–51.

[12] KEMPER A, NEUMANN T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots[C]//ICDE. [S.l.]: IEEE, 2011: 195–206.

[13] DIACONU C, FREEDMAN C, ISMERT E, et al. Hekaton: Sql server's memory-optimized oltp engine[C]//SIGMOD. [S.l.]: ACM, 2013: 1243–1254.

[14] HARIZOPOULOS S, ABADI D J, MADDEN S, et al. Oltp through the looking glass, and what we found there[C]//SIGMOD. [S.l.]: ACM, 2008: 981–992.

[15] TU S, ZHENG W, KOHLER E, et al. Speedy transactions in multicore in-memory databases[C]//SOSP. [S.l.]: ACM, 2013: 18–32.

[16] DECANDIA G, HASTORUN D, JAMPANI M, et al. Dynamo: amazon's highly available key-value store[C]//SOSP: volume 41. [S.l.]: ACM, 2007: 205–220.

[17] LAKSHMAN A, MALIK P. Cassandra: a decentralized structured storage system [J]. SIGOPS, 2010, 44(2): 35–40.

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

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

[20] MOHAN C, LINDSAY B. Efficient commit protocols for the tree of processes model of distributed transactions[J]. SIGOPS Oper. Syst. Rev., 1985, 19(2): 40–52.

[21] KUNG H T, ROBINSON J T. On optimistic methods for concurrency control[J]. TODS, 1981, 6(2): 213–226.

[22] BERENSON H, BERNSTEIN P, GRAY J, et al. A critique of ansi sql isolation levels[C]//SIGMOD: volume 24. [S.l.]: ACM, 1995: 1–10.

[23] HERLIHY M, WING J M. Linearizability: A correctness condition for concurrent objects[J]. ACM Trans. Program. Lang. Syst., 1990, 12(3): 463–492.

[24] KNUTH D E. The art of computer programming: Fundamental algorithms: volume 1[M]. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[25] STONEBRAKER M. New opportunities for new SQL[J]. Commun. ACM, 2012, 55(11): 10–11.

[26] HELLAND P. Life beyond distributed transactions[J]. Commun. ACM, 2017, 60 (2): 46–54.

[27] SAMARAS G, BRITTON K, CITRON A, et al. Two-phase commit optimizations and tradeoffs in the commercial environment[C]//ICDE. [S.l.: s.n.], 1993: 520–529.

[28] KALLMAN R, KIMURA H, NATKINS J, et al. H-store: a high-performance, distributed main memory transaction processing system[J]. PVLDB, 2008, 1(2): 1496–1499.

[29] STONEBRAKER M, WEISBERG A. The voltdb main memory dbms.[J]. IEEE Data Eng. Bull., 2013, 36(2): 21–27.

[30] PAVLO A, CURINO C, ZDONIK S B. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems[C]//SIGMOD. [S.l.: s.n.], 2012: 61–72.

[31] LOESING S, PILMAN M, ETTER T, et al. On the design and scalability of distributed shared-data databases[C]//SIGMOD. [S.l.]: ACM, 2015: 663–676.

[32] DRAGOJEVIC A, NARAYANAN D, NIGHTINGALE E B, et al. No compromises: distributed transactions with consistency, availability, and performance[C]//SOSP. [S.l.: s.n.], 2015: 54–70.

[33] OUSTERHOUT J, AGRAWAL P, ERICKSON D, et al. The case for ramclouds: scalable high-performance storage entirely in dram[J]. SIGOPS, 2010, 43(4): 92–105.

[34] DRAGOJEVIC A, NARAYANAN D, CASTRO M, et al. FaRM: Fast Remote Memory[C]//NSDI. [S.l.: s.n.], 2014: 401–414.

[35] WEI X, SHI J, CHEN Y, et al. Fast in-memory transaction processing using rdma and htm[C]//SOSP. [S.l.]: ACM, 2015: 87–104.

[36] WANG Z, MU S, CUI Y, et al. Scaling multicore databases via constrained parallel execution[C]//SIGMOD. [S.l.]: ACM, 2016: 1643–1658.

[37] LEIS V, KEMPER A, NEUMANN T. Exploiting hardware transactional memory in main-memory databases[C]//ICDE. [S.l.: s.n.], 2014: 580–591.

[38] REN K, THOMSON A, ABADI D J. An evaluation of the advantages and disadvantages of deterministic database systems[J]. PVLDB, 2014, 7(10): 821–832.

[39] DIEGO ONGARO J K O. In search of an understandable consensus algorithm[C]//ATC. [S.l.: s.n.], 2014.

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

[41] RÖDIGER W, MÜHLBAUER T, KEMPER A, et al. High-speed query processing over high-speed networks[J]. PVLDB, 2015, 9(4): 228–239.

[42] ELDAWY A, LEVANDOSKI J, LARSON P Å. Trekking through siberia: Managing cold data in a memory-optimized database[J]. PVLDB, 2014, 7(11): 931–942.

[43] LEVANDOSKI J J, LARSON P Å, STOICA R. Identifying hot and cold data in main-memory databases[C]//ICDE. [S.l.]: IEEE, 2013: 26–37.

[44] Facebook Open Source. MyRocks[EB/OL]. 2017. http://myrocks.io/.

[45] ALEXANDRE V, ANURAG G, DEBANJAN S, et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases[C]//SIGMOD. [S.l.: s.n.], 2017: 1041–1052.

[46] MOHAN C, LINDSAY B. Efficient commit protocols for the tree of processes model of distributed transactions[J]. SIGOPS Oper. Syst. Rev., 1985, 19(2): 40–52.

[47] THOMSON A, ABADI D J. The case for determinism in database systems[J]. VLDB, 2010, 3(1-2): 70–80.

[48] SERAFINI M, MANSOUR E, ABOULNAGA A, et al. Accordion: elastic scalability for database systems supporting distributed transactions[J]. PVLDB, 2014, 7 (12): 1035–1046.

[49] TAFT R, MANSOUR E, SERAFINI M, et al. E-store: Fine-grained elastic partitioning for distributed transaction processing systems[J]. PVLDB, 2014, 8(3): 245–256.

[50] SHUTE J, VINGRALEK R, SAMWEL B, et al. F1: A distributed SQL database that scales[J]. PVLDB, 2013, 6(11): 1068–1079.

[51] CHANDRASEKARAN S, BAMFORD R. Shared cache-the future of parallel databases[C]//ICDE. [S.l.]: IEEE, 2003: 840–850.

[52] LAHIRI T, SRIHARI V, CHAN W, et al. Cache fusion: Extending shared-disk clusters with shared caches[C]//VLDB. [S.l.: s.n.], 2001: 683–686.

[53] JOSTEN J W, MOHAN C, NARANG I, et al. Db2's use of the coupling facility for data sharing[J]. IBM Systems Journal, 1997, 36(2): 327–351.

[54] GOEL A K, POUND J, AUCH N, et al. Towards scalable real-time analytics: an architecture for scale-out of olxp workloads[J]. PVLDB, 2015, 8(12): 1716–1727.

[55] ROSENBLUM M, OUSTERHOUT J K. The design and implementation of a log-structured file system[J]. ACM Trans. Comput. Syst., 1992, 10(1): 26–52.

[56] TAN W, TATA S, TANG Y, et al. Diff-index: Differentiated index in distributed log-structured data stores[C]//EDBT. [S.l.: s.n.], 2014: 700–711.

[57] TPC. TPC-C[EB/OL]. 2017. http://www.tpc.org/.

[58] Oracle Corporation. MySQL Cluster[EB/OL]. 2017. http://www.mysql.com/products/cluster/.

[59] BAILIS P, DAVIDSON A, FEKETE A, et al. Highly available transactions: Virtues and limitations[J]. VLDB, 2013, 7(3): 181–192.

[60] MOHAN C, HADERLE D, 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.

[61] LAMPORT L. The part-time parliament[J]. TOCS, 1998, 16(2): 133–169.

[62] WU Y, CHAN C Y, TAN K L. Transaction healing: Scaling optimistic concurrency control on multicores[C]//SIGMOD. [S.l.]: ACM, 2016: 1689–1704.

[63] YAN C, CHEUNG A. Leveraging lock contention to improve oltp application performance[J]. PVLDB, 2016, 9(5): 444–455.

[64] ROY S, KOT L, BENDER G, et al. The homeostasis protocol: Avoiding transaction coordination through program analysis[C]//SIGMOD. [S.l.: s.n.], 2015: 1311–1326.

[65] MUCHNICK S S. Advanced compiler design and implementation[M]. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[66] VOSSEN G. Database transaction models[M]//VAN LEEUWEN J. Computer Science Today: Recent Trends and Developments. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995: 560–574.

[67] KENNEDY K, MCKINLEY K S. Maximizing loop parallelism and improving data locality via loop fusion and distribution[C]//Languages and Compilers for Parallel Computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994: 301–320.

[68] KENNEDY K, MCKINLEY K S. Loop distribution with arbitrary control flow [C]//Proceedings Supercomputing. [S.l.: s.n.], 1990: 407–416.

[69] Alibaba Oceanbase. Oceanbase[EB/OL]. 2015. https://github.com/alibaba/oceanbase.

[70] WHITE B, LEPREAU J, STOLLER L, et al. An integrated experimental environment for distributed systems and networks[C]//OSDI. [S.l.: s.n.], 2002: 255–270.

[71] DIFALLAH D E, PAVLO A, CURINO C, et al. Oltp-bench: An extensible testbed for benchmarking relational databases[J]. PVLDB, 2013, 7(4): 277–288.

[72] HARIZOPOULOS S, ARGYROS T, BONCZ P A, et al. Database architecture (r)evolution: New hardware vs. new software[C]//ICDE. [S.l.: s.n.], 2010: 1210.

[73] DEWITT D J, KATZ R H, ET AL. Implementation techniques for main memory database systems[C]//SIGMOD. [S.l.: s.n.], 1984: 1–8.

[74] SIRIN U, TÖZÜN P, POROBIC D, et al. Micro-architectural analysis of in-memory OLTP[C]//SIGMOD. [S.l.: s.n.], 2016: 387–402.

[75] WANG T, KIMURA H. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores[J]. VLDB, 2016, 10(2): 49–60.

[76] PAVLO A. What are we doing with our lives?: Nobody cares about our concurrency control research[C]//SIGMOD. [S.l.: s.n.], 2017: 3–3.

[77] HORIKAWA T. Latch-free data structures for DBMS: design, implementation, and evaluation[C]//SIGMOD. [S.l.: s.n.], 2013: 409–420.

[78] GOTTEMUKKALA V, LEHMAN T J. Locking and latching in a memory-resident database system[C]//VLDB. [S.l.: s.n.], 1992: 533–544.

[79] GRAY J, LORIE R A, PUTZOLU G R, et al. Granularity of locks and degrees of consistency in a shared data base[C]//Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems. [S.l.: s.n.], 1976: 365–394.

[80] YU X, PAVLO A, SÁNCHEZ D, et al. Tictoc: Time traveling optimistic concurrency control[C]//SIGMOD. [S.l.: s.n.], 2016: 1629–1642.

[81] THOMASIAN A. Two-phase locking performance and its thrashing behavior[J]. ACM Trans. Database Syst., 1993, 18(4): 579–625.

[82] JUNG H, HAN H, FEKETE A, et al. A scalable lock manager for multicores[J]. TODS, 2014, 39(4): 29.

[83] REN K, THOMSON A, ABADI D J. Lightweight locking for main memory database systems[C]//VLDB: volume 6. [S.l.: s.n.], 2012: 145–156.

[84] JOHNSON R, PANDIS I, AILAMAKI A. Improving OLTP scalability using speculative lock inheritance[J]. PVLDB, 2009, 2(1): 479–489.

[85] PANDIS I, JOHNSON R, HARDAVELLAS N, et al. Data-oriented transaction execution[J]. PVLDB, 2010, 3(1): 928–939.

[86] 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.

[87] JOHNSON R, ATHANASSOULIS M, STOICA R, et al. A new look at the roles of spinning and blocking[C]//DaMoN. [S.l.: s.n.], 2009: 21–26.

[88] AILAMAKI A, LIAROU E, TÖZÜN P, et al. How to stop under-utilization and love multicores[C]//ICDE. [S.l.: s.n.], 2015: 1530–1533.

[89] CHEN J, CHEN Y, ET AL. Big data challenge: a data management perspective[J]. Frontiers of Computer Science, 2013, 7(2): 157–164.

[90] BAYER R, MCCREIGHT E M. Organization and maintenance of large ordered indices[J]. Acta Inf., 1972, 1: 173–189.

[91] GRAEFE G. Write-optimized b-trees[C]//VLDB. [S.l.: s.n.], 2004: 672–683.

[92] LevelDB Community. LevelDB[EB/OL]. 2017. http://leveldb.org/.

[93] GHEMAWAT S, GOBIOFF H, LEUNG S T. The google file system[C]//SOSP: volume 37. [S.l.: s.n.], 2003: 29–43.

[94] BAKER J, BOND C, ET AL. Megastore: Providing scalable, highly available storage for interactive services[C]//CIDR: volume 11. [S.l.: s.n.], 2011: 223–234.

[95] PENG D, DABEK F. Large-scale incremental processing using distributed transactions and notifications.[C]//OSDI: volume 10. [S.l.: s.n.], 2010: 1–15.

[96] AHMAD M Y, KEMME B. Compaction management in distributed key-value datastores[J]. PVLDB, 2015, 8(8): 850–861.

[97] SEARS R, RAMAKRISHNAN R. blsm: a general purpose log structured merge tree[C]//SIGMOD. [S.l.: s.n.], 2012: 217–228.

[98] GREMILLION L L. Designing a bloom filter for differential file access[J]. Commun. ACM, 1982, 25(9): 600–604.

[99] BLOOM B H. Space/time trade-offs in hash coding with allowable errors[J]. CACM, 1970, 13(7): 422–426.

[100] GRAY C, CHERITON D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency[J]. SIGOPS Oper. Syst. Rev., 1989, 23(5): 202–210.

[101] LEVANDOSKI J J, LOMET D B, ET AL. The bw-tree: A b-tree for new hardware platforms[C]//ICDE. [S.l.: s.n.], 2013: 302–313.

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

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

[104] HELLAND P, SAMMER H, LYON J, et al. Group commit timers and high volume transaction systems[C]//High Performance Transaction Systems. [S.l.: s.n.], 1987: 301–329.

[105] GRAY J, HELLAND P, ET AL. The dangers of replication and a solution[C]// SIGMOD. [S.l.: s.n.], 1996: 173–182.

[106] Wiki Community. Intel 3D XPoint[EB/OL]. 2018. https://en.wikipedia.org/wiki/3D_XPoint.

致　　谢

六年的求学和研究时间匆匆而过。回忆自己刚入学时候的状态，才发现经过这些年我居然有了这么多的成长和改变。此刻，我非常庆幸自己能够来到这个平台，认识许多优秀的师长和同学。

博士学业的完成离不开许多师长对我的指导。首先，我深深地感谢两位指导老师：钱卫宁教授和王晓玲教授。在研究过程中，我有过找不到方向的彷徨，实验进展不顺利的焦虑以及论文被多次拒稿的失落。在遇到这些困难时，两位导师给了我许多帮助和支持。钱老师用深刻的思考、清晰的逻辑帮助我定下了研究的路线，理清了许多问题的解决方式。钱老师在研究中给予的大量支持是我能够完成博士工作的基础。王老师为我提供了许多学习的机会和资源，大大开阔了我的知识面。她的支持和鼓励也坚定了我前进的信心。同时，我也要感谢李飞飞老师、周煊老师和胡卉芪老师。三位老师在论文的写作方面给了我很多的帮助和指导。在合作中，我看到了优秀研究者严谨、专注和精益求精的工作方式。这也激励了我在今后用更高的准则来要求自己。此外，也非常感谢周敏奇老师、蔡鹏老师、张蓉老师、张召老师、高明老师、金澈清老师、宫学庆老师、余海萍老师和沙朝锋老师在日常学习中给我的指导。得益于周傲英老师辛苦搭建的科研平台，我才有机会接触到这么多优秀的研究者。非常感谢周老师提供的这个学习和工作平台。

这些年，我也有幸结识了许多同学。这里，我首先要感谢几位师兄：林煜明，王立，胡灏继，徐辰和夏帆。在几位师兄的帮助下，我逐步找到了研究的感觉。感谢周欢、王冬慧、黄建伟和屈兴同学在研究最繁忙阶段给予的协助，谢谢你们分担我的压力、包容我的情绪。感谢丁国浩、肖冰、张子豪帮助我完善我的论文。非常有荣幸能和毛嘉莉、章志刚和刘辉平三位博士一起毕业。此外，特别感谢王科强和李永峰这些年和我分享快乐，排解烦恼。两位让我的博士生涯变得更多彩。感谢庞艳霞、孔超、董绍婵、张磊、顾伶、晁平复、张新洲、宋乐怡、康强强等同学。和你们一起学习的日子非常的充实快乐。最后，也非常感谢实验室的各位小伙伴：郭进伟、李宇明、段慧超、朱燕超、储佳佳、张春熙、方祝和、庞天泽、翁海星、张晨东、樊秋实、余楷、余晟隽、祝君、周楠、王雷、王嘉豪、张燕飞、李捷荧、徐石磊、王彦朝等同学日常的陪伴和帮助。

最后，非常感谢我的父母、姐姐多年的关怀和理解。你们的理解和支持让我能够全心全意地学习和研究。感谢女友多年的等待、陪伴和照顾。谢谢你排解我的焦虑、舒缓我的紧张情绪，为我带来了平静和快乐。有你的未来是我拼搏的动力。

攻读博士学位期间发表的学术论文

[1] **Tao Zhu**, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, Huiqi Hu: Towards a Shared-Everything Database on Distributed Log-Structured Storage. In Proceedings of *2018 USENIX Annual Technical Conference*, 2018 (Accepted).

[2] **Tao Zhu**, Donghui Wang, Huiqi Hu, Weining Qian, Xiaoling Wang, Aoying Zhou: Interactive Transaction Processing for Main-Memory Database System. In Proceedings of *the 23rd International Conference on Database Systems for Advanced Applications*, 2018 (Accepted).

[3] **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*, 2017 (Accepted).

[4] **Tao Zhu**, Huiqi Hu, Weining Qian, Aoying Zhou, Mengzhan Liu, Qiong Zhao: Precise Data Access on Distributed Log-Structured Merge-Tree. In Proceedings of *APWeb-WAIM Joint Conference on Web and Big Data* (2) 2017: 210-218.

[5] **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.

[6] **Tao Zhu**, Yuming Lin, Ji Cheng, Xiaoling Wang: Efficient Diverse Rank of Hot-Topics-Discussion on Social Network. In Proceedings of *the 15th International Conference on Web-Age Information Management* 2014: 522-534.

[7] Huiqi Hu, **Tao Zhu**, Xuan Zhou, Weining Qian, Aoying Zhou. In-memory Transaction Processing: Towards Efficiency and Scalability Issues. (Submitted to *Knowledge and Information Systems*).

[8] 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.

[9] Yuming Lin, **Tao Zhu**, Hao Wu, Jingwei Zhang, Xiaoling Wang, Aoying Zhou: Towards online anti-opinion spam: Spotting fake reviews from the review sequence. In Proceedings of *International Conference on Advances in Social Network Analysis and Mining* 2014: 261-264.

[10] Yuming Lin, **Tao Zhu**, Xiaoling Wang, Jingwei Zhang, Aoying Zhou: Towards online review spam detection. In Proceedings of *the 21st International Conference on World Wide Web(Companion Volume)* 2014: 341-342.

[11] 林煜明, 王晓玲, **朱涛**, 周傲英. 用户评论的质量检测与控制研究综述. 软件学报,2014,25(3):506-527.

[12] 林煜明, **朱涛**, 王晓玲, 周傲英. 面向用户观点分析的多分类器集成和优化技术. 计算机学报,2013,36(8):1650-1658.