中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

关系型数据库到区块链的数据同步方法研究与实现

作者姓名：王博

指导教师：刘淘英 副研究员 中国科学院计算技术研究所

学位类别：工学硕士

学科专业：计算机软件与理论

培养单位：中国科学院计算技术研究所

2020 年 6 月

The Research and Implementation of Data Synchronization Approaches From Relational Database to Blockchain System

A thesis submitted to the

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Science in Engineering

in Computer Software and Theory

By

Wang Bo

Supervisor: Associate Professor Liu Taoying

Institute of Computing Technology, Chinese Academy of Sciences

June, 2020

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关保存和使用学位论文的规定，即中国科学院大学有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

作为一种新型的数据存储方式，区块链系统具有去中心化、开放、透明、不可篡改、可信的特性，在物联网、供应链、跨境金融、教育、医疗健康等领域有着广阔的应用空间。

在这些领域，使用关系型数据库存储数据的 IT 系统已经非常普及。由于区块链相比数据库系统事务处理能力有限，且存储系统的迁移将带来较高的开发成本，一种较为现实的区块链系统的应用模式是：保留原系统，并将遗留系统的数据实时同步至区块链系统。

为了实现这一模式，学界已经出现了高效的同步中间件的研究工作。该工作将原事务合并、压缩后上链，同步性能非常优秀，但依然存在以下不足：1）链上只存储合并、压缩后的数据，实际读取数据时必须经过中间件；2）中间件中存在较多的线程空转及争用现象，同步性能有一定优化空间；3）对低负载场景同步性能不够稳定；4）对存在 DDL 查询的场景会有数据同步错误的问题。

针对上述不足，本文提出了数据同步的两种模式：链上模式和链下模式，并对两种模式各提出了一种高效的数据同步方法。

链上模式指所有待同步数据都将存储在区块链上。本文提出的方法将多个原始事务映射为一个区块链事务，有效降低了区块链系统在共识过程中的网络传输及数据校验开销，提升了系统的吞吐量。

链下模式指允许部分数据存储在中间件。本文对学界已有的同模式方法进行了改进，解耦了计算与 IO 逻辑，引入了合并打包机制，有效降低了线程空转及争用造成的额外开销，提升了系统的吞吐量及稳定性。

对以上两种方法，本文选用了 MySQL 和 Hyperledger Fabric 进行了数据同步系统的实现。经测试，在本文的实验环境下，链上模式方法系统极限 TPS 达到 8000，相比直接上链的基准方法性能提高 16 倍；链下模式方法系统极限 TPS 达到 16000，相比目前同模式最佳方法提高 14%。

关键词：区块链，数据库，数据同步，在线事务处理

Abstract

As a new type of data storage system, the blockchain system has the characteristics of decentralization, openness, transparency, non-tampering and credibility, which attracted much attention in the fields of IoT, supply-chain, cross-border finance, education, e-health.

In these areas, IT systems that use relational databases to store data are very popular. Compared with the database system, the transaction processing capacity of the blockchain system is limited, and the migration of the storage system will bring higher development cost. Therefore, a more realistic application mode is synchronizing data from database to the blockchain system in real time.

For this issue, there is a middleware approach which greatly improved the throughput of synchorinaztion. In this work, the middleware merges and compresses the original transactions and write them to the blockchain, which achieve excellent performance. However, there are still some shortcomings: 1) The blockchain system only stores the merged and compressed data which can't provide data query service independently; 2) There are many thread contention and idling phenomenon which influence the performance; 3) The synchronization performance is not stable for low-load scenarios; 4) DDL queries will cause data update error.

To address above problems, this paper proposes two types of middleware-based data synchronization approaches: on-chain type and off-chain type.

The on-chain type means that all data to be synchronized will be stored on the blockchain. The on-chain approach maps multiple original transactions to a blockchain transaction, which effectively reduces the network transmission and data verification overhead of the blockchain system during the consensus process and improves the system throughput.

The off-chain type allows part of data to be stored in the middleware. The off-chain approach improves a related work, decouples CPU-intensive logic and I/O-intensive logic and proposes merge packing mechanism which effectively reduces the extra over-

head caused by thread idling and contention and improves the system throughput and stability.

For every approach, this paper implement a data synchronization system between MySQL and Hyperledger Fabric. In experiments, the peak TPS of the on-chain type approach reaches 8000, which is more than 16 times of the baseline approach, and the peak TPS of the off-chain type approach reaches 16000, which makes 14% improvement over the original method.

Keywords: Blockchain, Database, Data Synchronization, Online Transaction Processing

目 录

图形列表

表格列表

第 1 章　绪论

1.1　研究背景及意义

近年来，区块链技术发展十分迅速，在学界和产业界得到了广泛关注。最初，区块链技术主要应用于加密货币领域，包括比特币等。自以太坊提出智能合约以后，区块链系统开始被看作一种通用的数据存储方式。相比传统的存储方式，如文件系统、数据库，区块链系统具有去中心化、开放、透明、不可篡改、可信的特性，在物联网、供应链、跨境金融、教育、医疗健康等诸多领域有着广阔的应用空间。

目前的应用实践中，根据使用场景的不同，区块链应用的存储模型可以分为链上和链下两种。链上模型指全部业务数据都存储在区块链系统上。链下模型指部分业务数据存储在区块链系统中，部分存储在区块链系统外。链上模型更适用于业务数据较少，事务并发量小的场景。链下模型是一种权衡，链上一般仅存储非常关键的，需要保证可信的数据，系统的事务处理能力更强，但是链下的数据依然存在被篡改的风险。

在这些应用领域中，传统的中心化架构的 IT 系统已经非常普及。以物流场景为例，过去配送货物的各环节数据一般统一存储在物流企业内部的 IT 系统中，同时以网页、移动应用等形式供客户、物流各环节员工等查询。对于这类 IT 系统，最为常见的存储介质是关系型数据库，如 MySQL，Oracle，Microsoft SQL Server 等。

然而，使用区块链应用直接替代传统 IT 系统依然存在诸多问题。区块链应用既要保留原有系统的核心功能，同时需要改变底层数据的存储机制，使数据具有不可篡改的特性。这就意味着，尽管既有业务流程没有发生变化，但在区块链应用的开发过程中依然会伴有大量的既有业务逻辑迁移及修改。除了既有业务逻辑读写机制的转变外，对于链上模型的区块链应用，还应将原系统中的全部数据迁移至区块链系统；对于链下模型的区块链应用，还应将原系统中的部分数据迁移至区块链系统，同时开发新的数据库和区块链的混合读写接口供上层使用。这意味着极大的迁移成本。另外，相比业界常用的数据库，现有区块链系统的存储成本相对高昂，事务处理性能也非常有限，较为常用的系统如以太坊的 TPS

约为 20、Hyperledger Fabric 的 TPS 约为 500。

出于性能、存储成本及迁移成本的考虑，一种更为现实的应用模式是，原有的业务操作流程不变，同时使用基于数据库的传统 IT 系统与区块链应用，实时地将数据库的全部或部分数据同步到区块链系统。从职能上，区块链系统用于存储需要保证可信的、用于各组织间交换的数据。传统 IT 系统用于各组织的内部信息管理，同时触发区块链系统的数据更新。实现这一模式需要解决的关键问题是，如何高效地将数据库上的数据实时同步至区块链系统。

因此，高效的关系型数据库到区块链的自动数据同步方法，可以非常有效地降低区块链系统的开发和使用成本，对区块链技术在各实际场景的应用具有极大的意义。

1.2 国内外相关研究

近年来，得益于分布式共识算法和可信的 P2P 网络，区块链成为了一种非常理想的公共账本的存储方式。除了比特币 [1]、以太坊 [2] 等加密货币，在许多领域已经出现了区块链系统的应用实践，一些较为典型的案例包括移动安全 [3]，物联网 [4]，个人隐私保护 [5]，供应链 [6]，电子健康 [7] 等。

但是，在实际应用中，区块链系统依然存在事务处理能力低、存储成本高的问题。经过对现有文献的检索，目前国内外的相关研究工作主要可以分为三类：对区块链系统的性能改进、基于区块链技术的数据库、使用中间件手段加速上链。

如何改进区块链系统的性能是目前的研究热点，主要思路是改进分布式共识协议，这也是区块链系统的性能瓶颈所在。对非许可链场景的代表案例，如 [8] 提出了一种基于 PoW（工作量证明，Proof of Work）的共识协议，[9] 提出了一种基于 PoS（权益证明，Proof of Stake）的共识协议，[10] 提出了一种基于 BFT（拜占庭容错，Byzantine Fault Tolerant）的共识协议，[11] 提出了一种基于 PoW 和 PoS 的混合共识协议。[12] 提出了一种基于 PoW 和 BFT 的共识协议。而对于许可链场景，如 Hyperledger Fabric[13] 和 Corda[14]，可以借助认证信息使用比非许可链协议的更高效的 CFT（Crash Fault Tolerant）类协议，如 Raft 协议 [15]。但是，此类研究带来的性能收益往往伴随着对原区块链系统一致性或安全性的弱化，而且，其底层存储模型一般不为关系型，需要开发者基于底层存储支持关

系型查询。对传统 IT 系统与区块链系统共同使用的情况，需要进行遗留系统业务逻辑的迁移，应用成本高。

基于区块链技术的数据库系统，代表案例是 BigChainDB[16]。它的性能非常优秀，但是其数据模型与关系型数据模型不同，同样不适用于与传统 IT 系统共存的场景。

[17] 提出了一种中间件设计，通过将多个原始事务合并、压缩后提交上链的方式，提升了同步系统的吞吐量。它的同步性能非常优秀，提供了对关系型模型的支持，不侵入遗留系统，降低了应用成本。它的特点在于链上只存储合并、压缩后的数据，实际读取链上数据时必须经过中间件，因此，它的设计并不适用于数据量小、并发量低的场景。而且，它的性能瓶颈在 SQL 解析于事务合并，没有充分挖掘区块链的写入性能，有一定的优化空间。

1.3　本文研究内容

根据对国内外相关研究的分析，在目前区块链技术的发展阶段，设计中间件加速数据上链的方案有很大的实际意义。

结合区块链系统的链上、链下存储模型概念，本文将探索两种不同模式的基于中间件的数据自动上链方法：

1. 链上模式，待同步数据将以原始、无损压缩或某种规则变换后的形式全部存储在区块链上，区块链系统本身可以通过智能合约提供相应的数据查询能力，适用于业务数据量小、并发量低的场景；

2. 链下模式，允许部分原始数据或中间数据存储在中间件，数据查询时可以通过中间件获取额外信息，适用于业务数据量相对较大、并发量相对较高的场景。

最终，本文将基于两种模式的同步方法，选用实际应用较为广泛的关系型数据库 MySQL 与联盟链 Hyperledger Fabric 实现数据同步系统，测试系统同步速度、上链时延等指标，对本文提出的数据同步方法进行评价和分析。

1.4　论文章节安排

论文一共分为五章：

第一章绪论。本章将介绍关系型数据库与区块链数据同步方法的研究背景、研究意义、国内外研究现状及本文主要的研究内容。

第二章相关技术介绍。本章将介绍本文所引用的技术，包括关系型数据库MySQL的主从复制原理，区块链技术原理，联盟链Hyperledger Fabric的架构、特点、事务流等。

第三章系统设计。本章将首先介绍本文所研究的关系型数据库与区块链数据同步方法的设计目标，分析关键问题，并提出解决方案。然后，本章将讨论SQL解析模块的设计，介绍解析模块支持的SQL查询范围，关系型数据模型到Key-Value数据模型的映射方式，解析模块的并发控制策略等；最后，本章将提出关系型数据库与区块链数据同步方法的链上模式和链下模式，分别介绍其通用架构、防宕机制，并相应的提出一种加速上链的方法。

第四章系统实现与性能测试。本章将首先介绍一种关系型数据库MySQL到区块链系统Hyperledger Fabric数据同步系统的实现，该系统的实现将基于本文提出的链上、链下两种模式的加速上链方法。然后，本章将介绍对该数据同步系统的性能测试情况，包括测试负载、基准测试、链上、链下两种模式的同步性能。最终，从写入性能、上链时延等角度对本文提出的链上、链下模式加速上链方法进行评价，分析各系统参数对系统同步性能的影响。

第五章总结与展望，总结本文的工作及后续需要解决的问题。

第 2 章　相关技术介绍

本章将介绍本文的系统设计、系统实现中使用的相关技术原理，包括 MySQL 的主从复制原理、区块链技术原理，以及业界流行的联盟链系统 Hyperledger Fabric。

2.1　MySQL 主从复制原理

一般来讲，关系型数据库会提供主从节点的数据复制功能。在 MySQL 中，每次事务的成功提交，会打印一条或多条更新日志，记录本次 MySQL 状态更新的 SQL 查询。最终，在 MySQL 上并行执行的事务会根据实际的提交顺序被串行化记录，这类日志被称为 binlog。

MySQL binlog 分为三种格式：STATEMENT、ROW、MIXED。

对 STATEMENT 格式的 binlog，其中记录的是 MySQL 执行的涉及数据修改的原始 SQL 查询，可能涉及多行数据的更新。这类方法的优点是日志量小，缺点是记录原始的执行 SQL 查询的同时，必须传递相应的执行上下文信息，才能保证 SQL 查询在 Slave 节点的正确执行。但随着 MySQL 功能的扩增，容易出现复制时的问题。

对 ROW 格式的 binlog，其中记录的是与原始 SQL 查询等价的，仅涉及单行数据更新的 SQL 查询。其优点是不容易出现复制问题，缺点是原有的一条 SQL 查询可能多应多条日志，日志量较大。

MIXED 格式为 ROW 格式和 STATEMENT 格式的混合，MySQL 会根据 SQL 查询决定采用 ROW 格式或是 STATEMENT 格式记录，如对于表的修改操作将使用 STATEMENT 格式，对于 UPDATE、DELETE 等涉及多行数据更新的查询，会使用 ROW 格式。总的来讲，出于安全的考虑，实践中较为常用的 binlog 格式是 ROW 格式。

MySQL 的主从复制功能就是基于主从节点间的数据复制协议传递 binlog 得以实现。如图2.1所示，其数据同步过程分为以下三步：

1. Master 节点在本地记录 binlog，同时触发数据更新事件；

2. Slave 节点捕获 Master 节点发送的数据更新事件，通过 IO 线程向主节点

拉取新的 binlog 并落盘到本地，被称为 Relay Log；

3. Slave 节点在本地解析 Relay Log，顺序执行其中的 SQL 查询，实现与 Master 节点的数据同步。主从节点将保持最终一致。

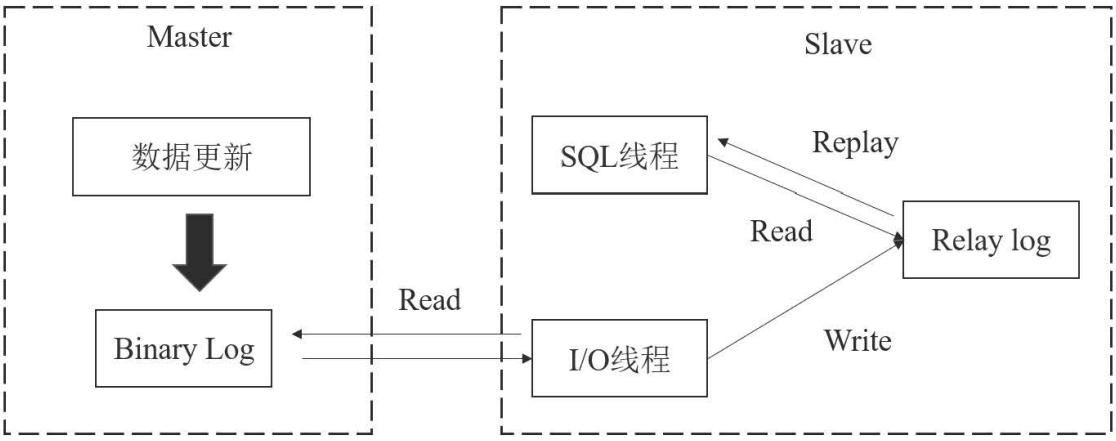

图 2.1 MySQL 主从复制流程

Figure 2.1 MySQL Master-Slave Replication

基于 MySQL 的以上主从复制机制，一种可以实时捕获 MySQL 的数据更新的方法是：实现 MySQL 数据复制协议，伪装 MySQL Slave 节点与 Master 节点进行通信，获取 binlog 更新事件。这种方法兼容性好，实时性强，在业界实际应用较为广泛，也出现了一些基于该原理的成熟开源项目，如阿里巴巴开发的 Canal、大众点评开发的 binlog2sql 等。

2.2 区块链技术简介

本节将介绍区块链系统的分类以及体系结构。

2.2.1 区块链系统分类

根据参与条件的不同，区块链可以分为公有链、私有链、联盟链三类。其中，公有链的是完全开放的，允许所有人加入网络。私有链是封闭的，只有所有者指定的机器才能加入。联盟链的开放程度介于公有链和私有链之间，通常由多个组织共同建立、管理和使用。

进一步的，根据区块链是否设有准入机制，区块链又可以分为许可链、非许可链两大类。非许可链即开放的公链。非许可链需要授权，包括私链和联盟链。

2.2.2　区块链体系结构

如图2.2[18] 所示，[18] 提出区块链体系结构可以分为应用层、控制层、共识层、数据层、网络层五层模型。

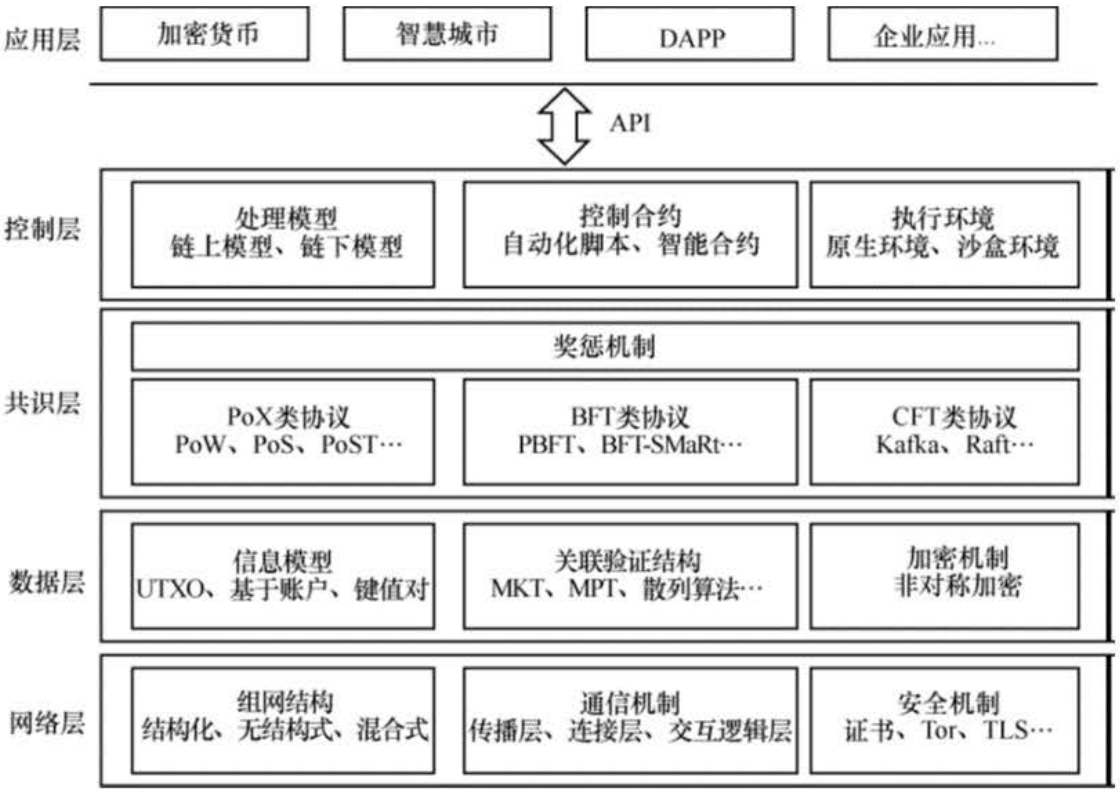

图 2.2 区块链体系结构

Figure 2.2 Fabric Transaction Processing

应用层指基于区块链系统的实际应用，如加密货币、跨境金融、物流等场景。

控制层包括区块链系统的事务处理模型、控制合约及控制合约的执行环境。其中，事务处理模型主要根据数据分布的不同，分为链上模型和链下模型。链上模型指全部数据存储在区块链。链下模型指部分数据存储在区块链内，部分数据存储在区块链外。值得注意的是，链下模型可以引入更高性能的链外存储，提升系统事务处理能力，但是需要精心设计数据在链上和链下的分布，保证数据的安全性。控制合约用于为区块链提供可编程能力，封装各类脚本、代码、智能合约。其中，智能合约是目前最为常用的方式。智能合约由用户定义后装载到区块链，即可被其他参与者发现。通过智能合约，无需中介就可以执行可追溯、不可逆转的交易，保证了交易的可靠性。以以太坊为例，以太坊的智能合约可以通过多种图灵完备的语言开发，如 Solidity。在以太坊上开发应用程序，只需要根据

业务逻辑编写相应的智能合约代码，然后编译后部署到区块链即可。控制合约的执行环境主要包括原生环境和沙盒环境。原生环境指合约代码与节点系统紧耦合，沙盒环境指为系统执行合约时将为其启动一个虚拟环境，如 Docker，预防了恶意代码攻击系统，比原生环境更加安全。

共识层主要包括保证各节点间数据一致性的共识算法，这也是区块链技术的核心与目前阶段学界研究的热点。区块链系统的容错问题是一个拜占庭系统容错问题。拜占庭系统指节点不仅可能出现宕机等故障，而且可能存在被人恶意操控的情况，即系统中存在作弊节点。目前，在区块链系统中应用较为广泛的共识算法可以分为 PoX[19]、BFT、CFT 三类。PoX 类共识算法以奖惩机制驱动，代表算法为 PoW、PoS 算法等。以 PoW 算法为例进行介绍，它的共识原理是：各个节点通过求解哈希函数 SHA256 竞争下一个区块的写入权，节点的算力不同，找到符合条件的哈希函数的输入值的概率也不同，如果想要对区块链进行修改，需要连续多次成功求解哈希函数 SHA256, 这需要付出巨大的算力成本，甚至超过攻击区块链取得的收益，从而可以认为该算法是足够安全的。它的潜在问题是：花费算力越多将预期收益越高，这类系统的大规模应用将导致大量的算力和能源浪费。BFT 类共识算法是指解决拜占庭系统容错问题的传统方法，包括 PBFT 等。PoX 类共识算法、BFT 类共识算法较为适合公链场景。而对于私链、联盟链等许可链场景，区块链系统可以借助准入机制规避作弊节点，使用效率较高的 CFT 类算法即可实现共识。CFT 类的常用算法包括 Paxos，Raft，ZAB 等。

数据层指代区块链底层的数据结构。最为经典的区块链底层实现即由区块组成的链表。新增的区块将在将加入链表的尾部，与前序区块通过一个哈希指针进行连接。哈希指针中包含了数据的哈希值，用于数据校验，同时也像常规指针一样，指向数据存储地址。这种设计的意义在于，通过一个哈希指针即可校验之前的全部区块，便捷的校验方法提高了篡改数据的难度，增强了区块链的安全性。

网络层规定了节点间通信的机制、系统的组网结构、安全认证机制等。

2.3 Hyperledger Fabric 简介

Hyperledger 是由 IBM 等公司发起的，面向企业级应用场景的分布式账本平台。作为 Hyperledger 的子项目，Fabric 是一个非常流行的联盟链系统。区别于

比特币、以太坊等公链系统，Fabric 具有以下特点：对交易参与者有严格的身份认定；加入交易网络需要获得许可；交易吞吐量较高，通常优于常用的公有链；交易有较强的时效性，确认延迟低；交易记录等数据的安全性和私密性强。本节将介绍 Fabric 的系统架构、事务处理流程。

2.3.1　Fabric 系统架构

在物理层面，Fabric 主要由客户端、peer 节点、排序节点组成。

peer 节点是 Fabric 网络的基本元素，主要包含账本和链码。其中，账本指记录各种交易信息的区块链。同时，为了优化只读事务的处理速度，Fabric 在 peer 节点引入了状态数据库。其具体实现为 CouchDB 或 LevelDB。每次 peer 节点的账本数据更新，均会对应的修改本地的状态数据库。在处理只读事务时，可以直接在状态数据库读取数据，返回结果。链码是 Fabric 对智能合约的实现。在 Fabric 中，链码的运行环境是 Docker，充分保证了 peer 节点的安全性。

客户端将通过 Fabric 提供的 SDK 与 peer 节点交互，提交事务。

排序节点是 Fabric Kafka 共识机制实现的关键。客户端向 peer 节点提交的交易，经过验证后都将发送到排序节点。排序节点通过 Kafka 对事务进行排序，最终打包成区块，广播到各 peer 节点，实现交易的成功写入。

作为联盟链系统，Fabric 主要面向多组织间的可信的数据管理。在逻辑层面，Fabric 引入了组织和通道的概念。每个组织将包含若干 peer 节点，共同维护组织内的账本。Fabric 的身份认证也是以组织为基本元素，每个组织将配有独立的 MSP（Membership Service Provider）。而通道的本质是 Fabric 网络上的私链，每个组织可以加入多个通道，维护多个账本。在通道内的交易对通道外的节点不可见，保证了数据的私密性。

另外，Fabric 模块化设计非常良好。许多组件支持可插拔，可以结合实际场景进行选择。例如，MSP 支持不同组织定义不同的身份认证逻辑，共识算法支持 PBFT、Raft 等。

2.3.2　Fabric 事务处理流程

如图2.3所示，Fabric 处理一个事务的流程分为以下六步：

1. Client 向背书节点发起提案；

2. 背书节点验证交易，确认发起交易的 Client 是否有权限、交易格式是否正

确、该交易之前是否已提交过、签名是否合法。在验证通过后，背书节点将根据提案的参数，调用相应的链码，模拟执行交易。最终把模拟的结果回应给 Client；

3. Client 收到背书节点对提案的回应并进行检查。如果是仅涉及数据查询的提案，直接返回收到的结果，事务处理结束。如果涉及数据写入，则检查背书签名是否合法，如果合法则准备提交给排序节点；

4. Client 将背书签名、背书节点对提案的回应打包提交到排序节点；

5. 排序节点收到消息，等待出块。出块后，区块将被广播到相关 peer 节点。在 Fabric 的配置中有两个出块参数：PreferredMaxBytes 和 MaxMessageCount。排序节点出块的条件是：如果目前接收的交易数据量大于 PreferredMaxBytes，或交易数超过 MaxMessageCount，则产生一个区块。

6. peer 节点收到区块，验证区块，更新账本，通知 Client 事务已提交。

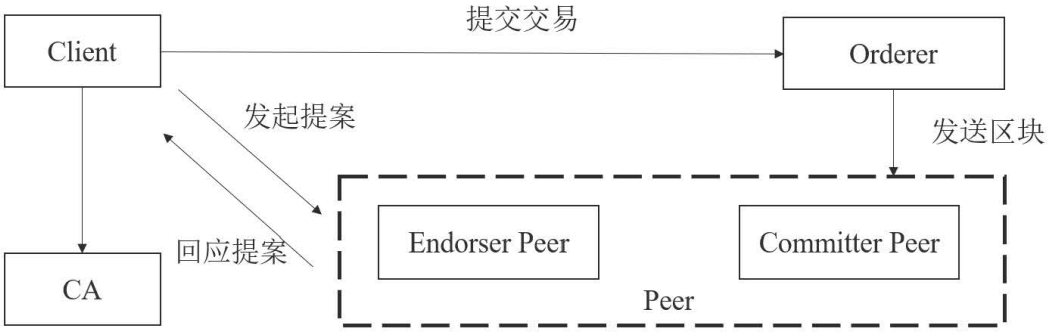

图 2.3 Fabric 事务处理流程

Figure 2.3 Fabric Transaction Processing

第 3 章　系统设计

系统间的数据同步，即数据源中的数据更新可以自动、实时、可靠地被待同步系统获取。待同步系统与数据源中的数据将保持最终一致。

本章将根据关系型数据库到区块链系统的数据同步问题的实际需求，确定本文的系统设计目标，再分析其中的关键问题及解决方案，最终提出本文的一种数据同步方案设计。

3.1　问题分析

3.1.1　设计目标

3.1.1.1　原子性、顺序性、一致性前提

目前来讲，区块链系统与关系型数据库间存在性能鸿沟，为了更好的满足实际需求，本文所研究的关系型数据库与区块链系统间的数据同步方法，将侧重于提高系统的同步性能。以关系型数据库作为数据源有以下两个特点：

1. 关系型数据库的事务处理有原子性的特点，即一个事务涉及的所有数据变更必须同时成功或失败。对于同构的关系型数据库间的同步，一般会有保留该特性的同步机制，保证中间状态的查询的正确性。

2. 本文只讨论单个关系型数据库实例作为数据源的情况，它的数据更新记录将是有序的。

对此，出于性能的考虑，在本文讨论的数据同步方法中，将不考虑关系型数据库事务处理的原子性。也将不保证区块链系统严格使用数据源的更新顺序进行更新，仅保证数据的最终一致性。这意味着：

1. 可能存在数据库中某个涉及多行数据更新的成功提交的事务，部分行数据成功上链，但部分行数据迟迟未在区块链中提交；

2. 可能存在数据库中的某表或是某表中的某行，在数据库系统中更新时刻较晚，但在区块链系统中更新时刻较早。

本文将基于以上对原子性、顺序性和一致性的要求，探索高效稳定的数据同步方法。

3.1.1.2 支持 SQL 解析

由于关系型数据库与区块链系统是异构系统，实现数据同步需要建立不同存储模型间的映射关系。这样，数据同步的过程将主要分为三个阶段：获取关系型数据库的数据更新，将关系型的数据更新映射到区块链的存储模型，写入区块链。

业界常用的关系型数据库如 MySQL、PostgreSQL、Oracle 等均有主从复制功能，用于同构系统的数据同步。这类功能的实现一般基于传递主数据库节点打印的数据更新日志。对不同的数据库，数据更新日志的格式并不统一，主要可以分为两类：WAL 日志和 SQL 日志。使用 WAL 日志进行主从复制的代表如 PostgreSQL。对 WAL 类日志，一般根据数据库定制的方式打印，需要使用同样的方式进行解析后获得数据表或行的更新情况。使用 SQL 类日志进行主从复制的代表如 MySQL。SQL 类日志将使用 SQL 查询的形式描述数据表或行的更新情况。从节点接收到日志后，将通过在本地执行 SQL 查询的方式进行数据同步。

可以发现，对不同类型的关系型数据库，获取数据更新日志的方法和 WAL 类更新日志的处理方法各不相同。在开发数据同步系统时，这类逻辑必须根据使用的关系型数据库类型定制。而 SQL 类日志的处理方法有较强的通用性。

结合上述情况，本文将设计一个 SQL 解析模块。其功能是将接收到的 SQL 查询形式的数据更新消息，解析为映射到区块链存储模型的数据更新消息。

3.1.1.3 适用不同量级的场景

实际的应用场景是非常多样的，各场景间业务数据量、数据更新的频繁程度有较大差异。如中小企业内部的设备管理系统，其部分数据更新并不频繁，远远达不到关系型数据库的极限性能，甚至直接使用区块链系统就可以满足需求。而对大型企业使用、运营的系统，投票系统等场景，数据更新极为频繁。

因此，为了满足不同场景的需求，并进一步发掘同步性能，本文将根据数据存储分布的不同，设计两种模式的数据上链方式：

1. 链上模式。全部待上链数据均存储在区块链，区块链系统本身可以提供数据查询能力；

2. 链下模式。在保证数据安全的前提下，允许部分数据存储在中间层，区块链系统本身可以不具备完整的数据查询能力，而需要通过中间层来获取数据。

3.1.2　评估方法

本文将使用两个指标评价数据同步方法的同步性能：数据的同步时延以及系统吞吐量。其中，数据的同步时延是指，待同步系统成功同步某版本数据的时刻与数据源数据更新到该版本的时刻间的差值。系统的吞吐量是指，数据同步系统每秒可以同步的更新消息数。

除了上述指标的峰值性能外，本文同样关注数据同步方法的稳定性。在实际场景中，数据更新频率往往是随时变化的。对于系统极限处理能力以下的流量，数据同步方法的上链时延应当稳定在一个可接受的数值，同步吞吐量应约等于输入流量。

3.1.3　关键问题及解决方案

结合上文分析，本文的关键问题是：1) 如何设计 SQL 解析模块; 2) 链上模式同步如何加速上链；3) 链下模式同步如何加速上链。

3.1.3.1　如何设计 SQL 解析模块

SQL 解析模块的功能是，将接收到的 SQL 查询解析为区块链存储模型的数据更新消息。考虑区块链系统的主流的存储模型是 Key-Value，本文将基于 Key-Value 存储设计 SQL 解析模块。

该模块的上游是 SQL 类更新日志。以 MySQL 为例，其更新日志被称为 binlog，共有三种形式：STATEMENT、ROW、MIXED。MIXED 格式的日志将由 STATEMENT 和 ROW 格式日志混合而成。其中，STATEMENT 格式记录原始 SQL 查询，缺失上下文信息。ROW 格式这类将原始 SQL 查询转换为行数据更新的日志形式较为适合数据更新使用。

因此，应当支持的关系型查询将包括表相关的 DDL 语句，如 CREATE TABLE、DROP TABLE 等，以及只涉及单行数据更新的 DML 语句，如 INSERT、DELETE 等。

该模块的输出是 Key-Value 更新消息流。因此，在设计上主要需要解决关系型到 Key-Value 存储模型的映射、待支持 SQL 查询的解析算法两个子问题。对这两个问题，NewSQL 数据库领域已经有了较为成功的实践范例，如 TiDB。

值得注意的是，对 SQL 类更新日志，有一个特点是，下游节点将会严格顺序执行各 SQL 查询。特别对 STATEMENT 格式的日志，只有严格顺序执行才能

保证更新后的数据会与数据源一致。然而，对于 ROW 模式的日志，其实各 SQL 查询具有幂等的特性。基于本文设计目标中的一致性、顺序性前提，在 SQL 解析模块可以多线程解析提高吞吐量。但是，涉及同一行数据的更新冲突问题，还应当记录原始顺序作为版本号，在写入前应比较版本号来判断是否需要进行本次更新。

但是，对所有 SQL 查询打乱顺序解析，依然可能出现错误更新。如果 SQL 更新流中出现 DDL 查询，则该表中的所有原始顺序落后于该 DDL 查询的 DML 查询，必须在该 DDL 查询执行后才能执行。考虑关系型数据库的实际使用中，DDL 查询远少于 DML 查询，在解析模块添加相应的并发控制策略即可解决。

3.1.3.2 链上模式加速上链

链上模式同步的目标是，区块链系统可以独立提供数据查询的能力。这意味着区块链系统中存储的数据，必然是源头的关系型数据库数据在某种规则下的唯一映射。

通过更新日志的解析，从上游获取的关系型数据库 SQL 更新日志将被转换为适用区块链底层存储模型的更新消息列表。一种显而易见的方案是，区块链系统提供一个智能合约，可以写入一个更新消息。同时，自动同步系统将上游传来的每个更新消息，调用智能合约提交到区块链。显然，该方案的性能将取决于区块链系统的事务处理能力。根据上文的介绍，区块链系统的事务处理流程一般非常复杂。以 Hyperledger Fabric 为例，从客户端发送一个事务到该事务成功被提交，需要经过背书节点、排序节点、peer 节点的层层验证和转发。考虑 Fabric 的极限 TPS 大约为 500，对一些非许可链性能甚至更低，在实际场景中，这种直接上链方案的同步性能一般难以接受。

经由以上分析，一种同步性能的优化思路是，将多个原始更新消息映射为一个区块链事务，共同提交上链。相比原始事务到区块链事务一对一映射的方法，多对一的方法网络传输和验证的总数据量几乎没有改变，但节点间进行网络通信的次数将有效降低，系统吞吐量将得到一定改善。同时，这会导致同步时延有一定程度的升高。

3.1.3.3　链下模式加速上链

链下模式是同步性能和系统功能完备性、查询性能的一种权衡，适用于并发量特别高、查询性能要求相对低的场景。[17] 是典型的链下模式的数据加速上链方案，该工作使用的区块链系统底层存储模型为 Key-Value，对关系型数据，它的同步流程是：解析原始 SQL 日志，得到 Key-Value 更新消息列表，而后将多条 Key-Value 更新消息合并、压缩处理，仅向链上写入合并、压缩后的数据，而在中间件的存储层维护原始 Key 与合并后 Key 的映射关系以及原始 Key 的数据版本信息。查询时，必须先通过中间件获取原始 Key 对应的实际存储在区块链上的合并 Key，再通过合并 Key 获取压缩数据块，最后解压数据块，才能读出需求数据。

在性能上，该工作的同步方法存在一定优化空间：

1. 工作线程执行的逻辑既包含合并、压缩等重计算的逻辑，又包含上链等重 IO 的逻辑。这将导致线程空转现象，有一定性能损失；

2. 系统启动时预设原始事务到区块链事务的映射比例、工作线程数等核心参数，系统实际运行中不会改变。在同步系统非满载的情况下，工作线程间的争用将导致实际的事务映射比例合并数低于预期，系统的同步性能不够稳定；

对于第一个问题，解决方案是将原有的工作模块拆分为合并模块和上链模块。合并模块执行合并、压缩等 CPU 密集型任务，上链模块负责调用智能合约将合并块提交上链的 IO 密集型任务。合并模块与上链模块间通过上链队列串联。每个模块使用独立的线程池执行任务，线程池使用的线程数独立调整。这样可以有效节约线程资源，减少争用和空转，提高同步性能。

对于第二个问题，解决方案是在合并线程池前增加一个打包线程，通过合并队列相连。打包线程负责接收上游的 Key-Value 更新消息，当累计消息数达到合并阈值或超时后，打包推送至合并队列。这种策略可以有效避免，非满载时工作线程争用导致的实际映射比例偏低问题。

3.2　SQL 解析模块的设计

关系型数据库与区块链系统底层的存储模型可能并不相同。因此，为了实现关系型数据库到区块链系统的数据同步，必须先将关系型数据库的数据更新记录转换为适应区块链的存储模型形式的更新消息。在本文中，该过程将被称为解

析。

经过对业界较为流行的关系型数据库的调研，许多数据库都提供了打印数据更新日志的功能。较为常见的数据更新日志格式包括 SQL 日志和 WAL 日志。例如，MySQL 提供的数据更新日志 binlog，格式为 SQL 日志；PostgreSQL 提供的数据更新日志 xlog，格式为 WAL 日志。这些数据更新日志的一个重要应用途径是实现同构数据库间的主从复制。例如 MySQL 的主从复制过程中，从节点将持续接收主节点的 binlog，依次执行 binlog 中记录的每条 SQL 查询，即可实现与主节点的数据同步。

由于 WAL 日志的解析逻辑将完全取决于关系型数据库的相关实现，而 SQL 日志的解析逻辑相对通用，再考虑到区块链系统中较为常用的数据存储模型为 Key-Value，本章将提出一种 SQL 解析模块的设计，该模块将提供将 SQL 日志解析为 Key-Value 更新消息的功能。

3.2.1 关系数据模型到 Key-Value 数据模型的映射

经过调研，业界的 NewSQL 数据库一种流行的实现方式是基于 Key-Value 数据库实现关系数据模型。本文参考了其中较为成熟的开源项目 TiDB 的设计，提出了如下关系型到 Key-Value 的映射方式。

关系型数据库中的 SQL 查询操作的逻辑对象主要包括数据库、数据表、数据行和索引。我们将它们分别映射为四类不同的 Key-Value 记录。为了不同类别间的 Key 不会产生冲突，我们将为每类记录分配一个互不相同的 Key 前缀。最终，一个关系型数据对象经映射后的 Key，将由该对象所属类别的 Key 前缀，以及可以唯一标识该对象的属性组合而成；映射后的 Value，将是一条结构化消息，记录逻辑对象的若干属性信息。

对于数据库对象，在 Key-Value 数据模型中将被映射为一条元信息记录。由于通过数据库名可以唯一标识数据库对象，我们将使用类前缀与数据库名为该记录构造 Key。Value 中将存储数据库对象的字符集等信息。

对于数据表对象，在 Key-Value 数据模型中将被映射为一条元信息记录。由于通过数据表名和该表所属的数据库名可以唯一标识数据表对象，我们将使用类前缀、数据库名、数据表名为该记录构造 Key。Value 中将存储数据表对象的列属性、主键、外键等信息。除此之外，为了有效区分每个数据行，数据表的元

信息记录中还应记录一个自增 ID 作为默认的主键。在创建数据表时，如果数据表没有设定主键，自增 ID 将被设为主键，初始值为 0。如果数据表已经设定主键，自增 ID 将被设置为空。

对于数据表中的每个数据行，在 Key-Value 数据模型中将被映射为一条数据记录。由于通过数据表 Key 和该行主键的取值可以唯一标识该数据行。我们将使用类前缀、数据表 Key、该行主键的取值为该记录构造 Key。Value 中将存储该数据行各属性的取值。

对于数据表中的索引，将在 Key-Value 数据模型中被映射为多条数据记录。索引记录的 Key 将是类前缀、数据表 Key、被索引字段的取值、数据行主键的组合，Value 将被设为空。

3.2.2　SQL 查询的解析算法设计

本文讨论的 SQL 解析模块将用于解析从关系型数据库获取的 SQL 类更新日志。因此，SQL 解析模块仅需支持 SQL 类更新日志中可能包含的 SQL 查询。首先，更新日志中中只会记录涉及数据更新的 SQL 查询。另外，关系型数据库在实际应用中一般使用 ROW 格式的 SQL 类日志。在这种情况下，当关系型数据库执行了涉及多行数据更新的 DML 查询后，记录在更新日志中的将是多条仅涉及单行数据更新的 SQL 查询，而不是原始的 SQL 查询。而且，当关系型数据库执行了非幂等更新 SQL 后，记录在日志中的将是幂等更新 SQL。例如，对数据更新查询 UPDATE TBL SET FIELD = FIELD + 1，在 ROW 格式 SQL 日志中记录的更新 SQL 将为 UPDATE TBL SET FIELD = SOME_VALUE。

综上所述，本节所讨论的 SQL 解析模块，需要支持的 SQL 查询包括：

1. DDL 查询：CREATE、ALTER、DROP；

2. DML 查询：INSERT、DELETE、UPDATE。

显然，根据上文所述的映射方式，所有待支持的语句都将被解析为唯一一条 Key-Value 记录。

结合上述背景，对于 SQL 查询的解析，将分为以下三个阶段：

1. 使用 SQL 语法分析器，将 SQL 查询转为抽象语法树；

2. 根据关系型到 Key-Value 的映射关系，生成对应的 Key-Value 记录；

3. 记录该 SQL 查询在更新日志文件内的位置，与解析后的 Key-Value 条目

共同作为一条消息推送至下游队列。

对于 DDL 查询，在第二阶段可以通过查询抽象语法树完成到 Key-Value 模型的映射。以 CREATE TABLE 语句为例，为了生成对应的 Key-Value 条目，需要获取数据库名、数据表名、各列名、各列数据类型、主键、外键等信息。这些在 ROW 格式日志记录的语句中都有记录。

但是，对 INSERT、UPDATE 语句，仅通过日志中记录的 SQL 查询可能不能完成映射。例如，在 MySQL 执行 INSERT 或 UPDATE 语句后，其打印的 ROW 格式 SQL 更新日志不包含列名称、列数据类型等信息。一行 UPDATE 查询对应的 ROW 格式日志记录如下所示：

```
UPDATE `test`.`test`
WHERE
    @1=1
SET
    @1=2
```

在这种情况下，SQL 解析模块需要引入存储系统，在运行时记录数据库的所有元信息。

另外，DROP、DELETE 语句与其他查询语义不同，需要进行特别的删除标记。一种较容易的方式是将 Value 设为空。

3.2.3 多线程解析策略

根据关系型数据库的同构系统主从复制原理，SQL 解析模块接收到的上游数据更新日志将是有序的。在同步时，同构数据库必须有序地执行日志中的 SQL 查询，才能保证和主节点数据的一致性。

本文所讨论的数据同步方法，将优先考虑同步性能，只需要保证关系型数据库和区块链系统数据的最终一致性，可以不保证原始消息提交的顺序性。可以发现，如果仅考虑 DML 查询，由于上游 ROW 格式日志中记录的 INSERT、UPDATE 语句具有幂等特性，在解析模块打乱更新日志的原有顺序，依然可以保证两系统间数据的最终一致。这样，我们就可以在解析模块可以开启多线程计算，极大提高吞吐量。

但是，对于有 DDL 查询的情况，就可能出现数据更新错误的问题。根据上节的介绍，INSERT、UPDATE 语句在解析时可能需要查询数据表的元信息，这

就导致了多线程解析仅能在两个同表的 DDL 查询的间隔中进行。然而，在实际业务场景中，DDL 查询的使用频率将远低于 DML 查询。对此，我们提出如下的多线程解析策略，以求提高解析模块的吞吐量：

1. 解析模块将由一个 IO 线程、一个解析队列，以及若干个工作线程组成；

2. IO 线程将持续消费上游发送的 ROW 格式 SQL 更新日志。每当 IO 线程接收到一个 DML 查询，将直接推送至解析队列。每当接收到 DDL 查询，将阻塞至该 DDL 查询同步成功；

3. 工作线程持续从解析队列拉取更新日志进行解析，将解析后的结果推送至下游模块等待写入区块链。

3.3　链上模式加速上链方法

区块链应用数据存储的链上模型是指，全部业务数据都将存储在区块链系统，客户端通过智能合约进行应用数据的读写。

受此概念启发，本节提出了关系型数据库到区块链系统数据同步的链上模式，并提出了一种该模式的高效的数据同步方法。

链上模式具体是指，关系型数据库待同步的数据将以某种无损的形式写到区块链系统上。区块链系统可以通过智能合约，独立地为客户端提供数据查询的功能。

3.3.1　通用架构

链上模式的数据同步系统通用架构如图3.1所示。其中，数据源是关系型数据库，待同步系统是区块链系统。数据同步中间件主要包括更新日志抓取模块、更新日志解析模块、同步控制模块三部分。

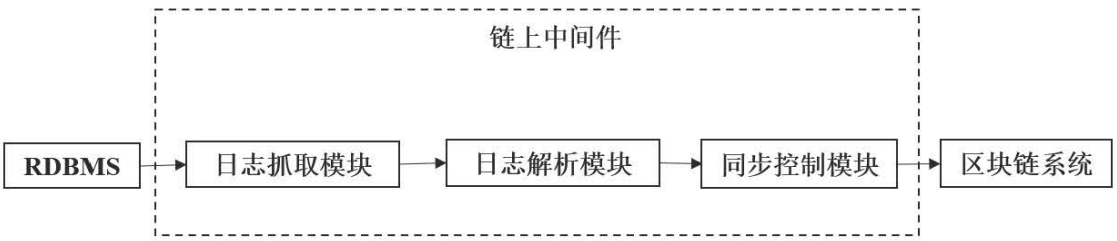

图 3.1 链上模式的数据同步系统通用架构

Figure 3.1 General Architecture of Data Synchronization System(on-chain)

数据同步的全过程分为四个阶段：

1. 关系型数据库接收客户端提交的事务，本地执行，打印数据更新日志；

2. 更新日志抓取模块获取关系型数据库的数据更新日志，落盘到本地；

3. 更新日志解析模块解析抓取到的数据更新日志，原日志被转换为区块链数据模型存储模型的更新消息；

4. 同步控制模块收集数据更新消息，调用区块链系统提供的智能合约，将所有数据写入区块链。

面对数据查询的需求，客户端应调用区块链系统提供的智能合约获取信息。

3.3.2 防宕机制

值得注意的是，为了关系型数据库与区块链系统间的数据一致性，中间件必须保证把所有更新消息可靠地送达区块链。如果中间件在同步过程中宕机，已经从上游队列消费的数据更新消息可能会丢失。因此，在中间件需要设计合理的防宕机制。本节将介绍两种适用的防宕机制：前写日志防宕机制和恢复位置防宕机制。

3.3.2.1 前写日志防宕机制

前写日志（WAL，Write Ahead Logging）防宕机制常用于关系型数据库的错误恢复。但与数据库不同的是，本文讨论的数据同步无需保证消息的顺序提交，且更新消息都是幂等的。因此，本文讨论的中间件在宕机恢复时，只需要找到所有未提交的消息重新尝试写入，不需要对已提交的消息进行回滚。

对此，我们可以提出如下日志记录方法：

1. 在中间件处理一条数据更新消息时，应在调用智能合约上链前，写入一行日志。日志中将记录该消息的更新内容、消息编号，并标记该消息是正在同步的状态；

2. 当该更新消息在区块链系统成功提交后，应再写入一条日志，标记该消息已同步成功。

基于以上日志，中间件宕机后的启动流程是：扫描全部日志，获取所有未提交的消息，重新推入中间件的打包队列。

可以发现，如果在 WAL 日志文件中，某行以前的日志中涉及的事务都已经成功提交，那么在启动时不需要扫描该行以前的日志。当同步系统启动时间较长时，记录的日志数量可能非常多。而且，实际场景中更可能出现的情况是，只有

最后若干行日志中存在未提交的事务。因此，这种设计可能会导致中间件在启动时需要扫描大量无用日志，启动速度慢。

为了提高中间件系统的重启速度，可以在 WAL 日志中引入检查点机制。这种机制是，增加一种新类型的日志：检查点日志类型。该类型的日志将记录一个位置信息，标志着：在 WAL 日志文件中，该位置以前的所有日志所记录的事务都已经成功提交。这种日志的记录方式是：同步中间件运行时，定时开启一个检查线程，从上一个检查点开始前向后扫描所有日志，确定是否存在某行之前记录的事务都已成功提交的情况，如果有，则写入一条新的检查点日志记录该位置。这样，中间件系统在重启时，只需从后向前扫描日志文件至最新的检查点位置。

3.3.2.2　恢复位置防宕机制

在本文所考虑的场景下，上游的更新消息是幂等的。基于这一特性，如果我们可以确定，某个位置前的所有更新消息已经成功提交。那么，中间件系统只要从该位置开始重新同步，依然可以保证数据的正确性。在本文中，我们将符合该特性的位置称为恢复位置。显然，在系统运行时，恢复位置并不唯一，但在某一确定时刻，一定存在恢复位置的最大值。

基于上述特性，我们可以提出一种新的防宕机制：在同步中间件运行时维护恢复位置，并定时将恢复位置持久化。在中间件重启时，直接读取最新一次记录的恢复位置，从该位置开始同步即可保证数据更新的正确性。

中间件在运行时维护恢复位置的方法是：系统初启动时恢复位置将被设置为 0。中间件将记录每条接收到的更新消息在原日志文件中的位置，以及该消息后续的同步状态。当一条更新消息成功提交到区块链时，将该消息的位置插入一个有序集合。在该有序集合进行二分查找，检查恢复位置到该消息位置的消息是否都已经成功同步。如果中间的所有消息都已经成功同步，就把恢复位置更新为该消息的位置，并将中间所有的消息从集合中移除。

相比 WAL，恢复位置机制的优点是，中间件系统不需要运行时时向磁盘写入日志，消耗更小，中间件的吞吐量上限更高。但是，这种机制也会导致部分已提交消息的重复处理，具体的范围是在恢复位置的最近落盘时间，到系统宕机时间内的提交上链消息。在实际使用中，应当结合系统的负载情况，决定恢复时间的落盘间隔。

3.3.3 加速方法

本节将介绍一种基于事务合并的高效的链上模式数据同步方法。由于在同步中间件中，更新日志的抓取模块和解析模块的设计，高度依赖系统中所使用的关系型数据库类型，本节将主要介绍该方法的同步控制模块的设计，以及区块链智能合约的设计，并讨论该方法的加速原理，给出系统参数的选择建议。

3.3.3.1 同步控制模块的设计

如图3.2所示，同步控制模块由打包模块和上链模块组成，模块间通过上链队列相连。

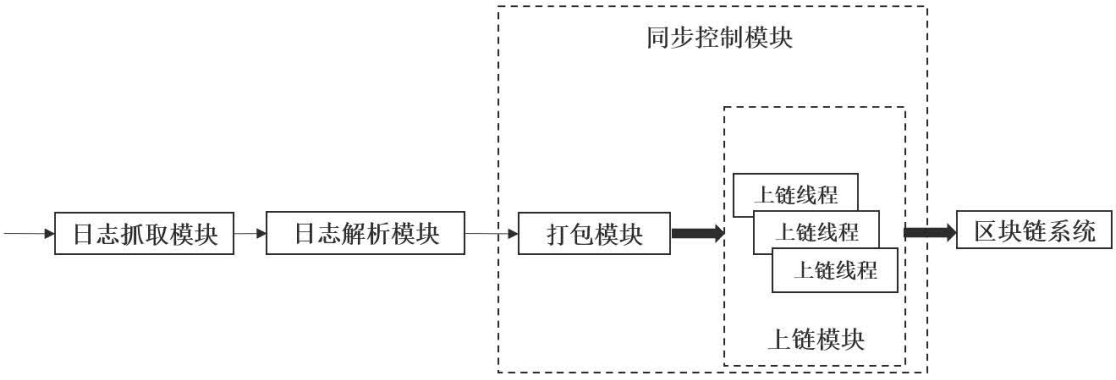

图 3.2 链上模式加速上链的同步控制模块设计

Figure 3.2 Effcient Synchronization Control Module Design(on-chain)

打包模块由一个打包线程组成。模块的参数包括最大合并数和合并超时时间。打包模块的工作流程是：打包线程持续消费上游更新日志解析模块发送的解析后的更新消息，把它们缓存在本地。每当缓存的消息数量达到预设的最大合并数，或者超过预设的合并超市时间仍未收集到足够的消息时，打包线程将触发一次打包，将缓存的消息列表发送至上链队列，最后清空缓存。

上链模块由一个 IO 线程和若干个上链线程组成。模块的参数包括最大上链线程数和失败重试时间。上链模块的工作流程是：IO 线程持续消费上游打包模块发送的消息列表。每当 IO 线程收到一个消息列表，将开启一个新的上链线程，调用智能合约将其写入区块链。其中，上链模块持有的上链线程数不会超过预设的最大上链线程数。如果上链线程调用智能合约成功，则标记该事务已经被成功提交；如果上链线程调用智能合约失败，上链线程将等待指定时间后，再次尝试提交。

3.3.3.2　区块链智能合约的设计

区块链系统将提供一个批量数据写入的智能合约。它的输入参数是待上链的更新消息列表，列表中的每个元素将是经解析的、适应区块链底层存储模型的更新消息，而且将携带数据版本信息。

该写入合约的执行逻辑是：依次处理各消息。对每条待处理的消息，首先判断该消息的数据版本是否比当前区块链系统中存储的该数据版本更新，如果该消息的数据版本比当前存储的版本更新，就按照该消息写入数据，反之，就把这条消息丢弃。

当该合约执行完成后，向上链线程返回的消息中，必须包含事务是否成功提交的信息。

此外，区块链系统还应提供类 SQL 接口的数据查询合约。但类 SQL 数据查询的设计及实现，本文将不再讨论。

3.3.3.3　加速原理及参数选择

根据上文所述，本文讨论的同步方法主要设计目标是提升同步速度，可以不保证事务的原子性和顺序性，只需要保证最终区块链上的数据可以与数据源保持一致。因此，在中间件层面，只需保证所有数据更新消息都能可靠送达区块链系统，原有的上链顺序可以被打乱。

对更新日志抓取模块，其吞吐量主要取决于关系型数据库本身，对本文讨论的数据同步方法，该模块的性能不可控。

对更新日志解析模块，其单线程情况下的吞吐量取决于解析模块本身的设计，不在本节的讨论范围之内。由于可以不考虑事务顺序提交，在该模块应当开启足量的解析线程以提高吞吐量。

根据上文所述，区块链的事务处理链路很长，需要经过多次网络通信和多节点的共同验证。打包模块将多个原始事务映射到了一个区块链事务，这种多对一的映射设计将有效降低网络通信的开销，也是实现加速上链的关键。值得注意的是，在链上模式的设计下，这种映射比例越高，区块链的写入智能合约的开销也就越大。因此，最大合并数需要合理设置，过大或过小都可能达不到较好的同步速度。

在打包模块中，另外一个参数是合并超时时间。在实际场景中，系统输入流

量的大小可能随着时间会有很大的变化。在流量较小，可能有更新消息一直在打包模块等待的情况。因此，合并超时时间应根据开发者对系统同步时延的期望设定。

对上链模块，应当开启足量的上链线程数，以达到区块链系统的满载。

3.4 链下模式加速上链方法

区块链系统每次处理事务时，都需要通过分布式共识算法，保证多节点间达成共识后，才能成功进行数据更新。因此，相比中心化存储的关系型数据库，区块链系统的事务处理性能较低，不能满足某些业务数据量大、数据更新频率高的应用场景。为了提升系统的事务处理能力，一种实践是将部分业务数据存储在区块链系统内，部分业务数据存储在其他更高性能的存储系统。在数据读写时，将根据数据所在的位置，选择相应的系统响应。这种区块链应用的数据存储方式被称为链下模型。

受此概念启发，为了进一步挖掘数据同步系统的同步性能，本节提出了关系型数据库到区块链系统数据同步的链下模式，并提出了一种该模式的高效的数据同步方法。链下模式具体是指，在数据同步时，允许在区块链外的存储系统存储部分数据。最终，区块链系统可以不具备独立的数据查询能力，不向客户端提供类 SQL 的数据查询合约，而是向中间件提供相关的数据查询合约。客户端在查询时，需要访问中间件所提供的类 SQL 数据查询接口。

3.4.1 通用架构

链下模式的数据同步系统通用架构如图3.3所示。其中，数据源是关系型数据库，待同步系统是区块链系统。数据同步中间件主要包括更新日志抓取模块、更新日志解析模块、同步控制模块、链下存储系统四部分。

数据同步的全过程分为四个阶段：

1. 关系型数据库接收客户端提交的事务，本地执行，打印数据更新日志；

2. 更新日志抓取模块获取关系型数据库的数据更新日志；

3. 更新日志解析模块解析抓取到的数据更新日志，原日志被转换为区块链数据模型存储模型的更新消息；

4. 数据同步中间件收集数据更新消息，根据既定的策略，将部分数据写入

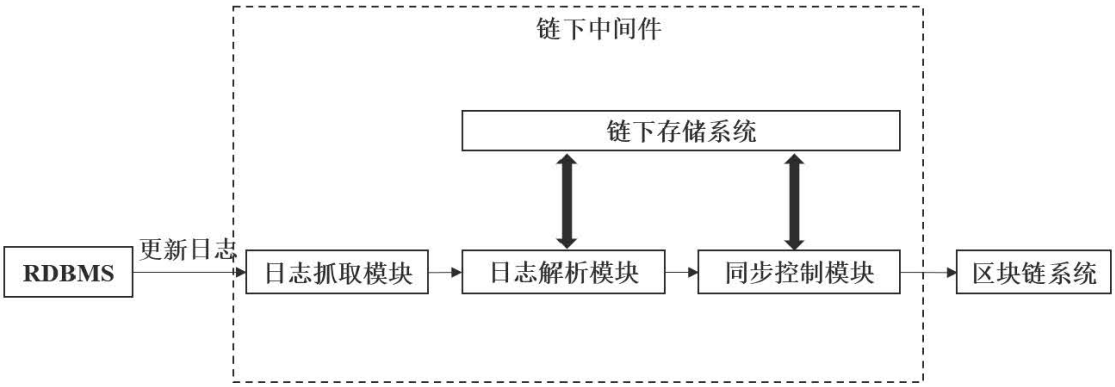

图 3.3 链下模式的数据同步系统通用架构

Figure 3.3 General Architecture of Data Synchronization System(off-chain)

链下存储模块，并调用智能合约将部分数据写入区块链系统。

面对数据查询的需求，中间件将提供类 SQL 语法的数据查询接口。中间件在执行查询时，将根据写入时的链上、链下数据分布策略，通过智能合约从区块链上读取部分数据，并链下存储模块读取另一部分数据，最终获得结果返回给客户端。

3.4.2　防宕机制

对链下模式的数据同步，涉及向两个系统中写入数据，一个是链下存储系统，一个是区块链系统。当两部分数据全部成功写入，一次同步事务才能被认为是成功提交。由于链下存储系统自主可控，写入性能远高于区块链系统。较为常见的事务处理机制是，首先将相关数据写入链下存储系统，返回成功后再尝试将其它待写数据写入区块链系统。

因此，对于链下模式的数据同步系统，防宕机制将分为两部分：未上链消息的防宕机制和链下存储系统的防宕机制。

3.4.2.1　未上链消息的防宕机制

与链上模式不同的是，链下模式的同步系统在宕机恢复时，如果只将所有未提交事务重新推入队列，系统可能不能恢复到原状态，还需要对在链下存储系统中已提交但未成功上链的更新消息进行回滚。

值得注意的是，虽然本文所考虑的场景中，上游关系型数据库的更新消息是幂等的。但这不意味着，在链下存储系统的数据写入逻辑也是幂等的。因为这些写入逻辑是需要开发者自定义的。

对于非幂等的情况，具体的回滚机制跟具体写入逻辑的设计与链下存储系统的选择高度相关。本节将主要讨论数据写入逻辑是幂等的情况。

如果链下存储系统的数据写入逻辑也是幂等的，那么使用类似链上模式中介绍的前写日志防宕机制、恢复位置防宕机制均可。上游更新消息处理完成的标志将是：相关数据已经在链下存储系统和区块链系统成功提交。

这种机制有一个问题是，在宕机恢复时，需要进行多次链下存储系统的无效访问，可能启动时间会较慢。我们可以提出一种适用于链下模式的改良的前写日志机制：

1. 把数据库更新消息的同步状态分为三种：未提交、链下存储已提交、已提交；

2. 当中间件接收到消息、链下存储已提交某消息、区块链已提交某消息时，将分别写入一条对应的日志，记录该消息的状态；

3. 在中间件重新启动时，扫描 WAL 日志文件。对所有未提交状态的更新消息，重新执行链下存储系统、区块链系统的写入逻辑。对所有链下存储已提交状态的更新消息，仅重新执行区块链系统的写入逻辑。

3.4.2.2 链下存储系统的防宕机制

由于链下存储系统中也存有关键数据，为了实现数据同步中间件的防宕，对链下存储系统本身也提出了一定要求，即链下存储系统的数据不能因宕机丢失，或丢失后有恢复手段。

对于基于硬盘的数据存储系统，如 MySQL、LevelDB、RocksDB 等，在写入时数据就已经持久化到磁盘，直接满足防宕的要求。而对于内存型的数据存储系统，如 Redis、Memcached 等，就需要特别注意宕机后的数据丢失问题。其中，部分内存型数据库也提供了持久化功能，如 Redis 的 AOF 模式，这种情况下，可以开启内存数据库的持久化功能来避免数据丢失。

如 [17] 的工作中所介绍，还有一种解决思路是，允许内存型数据存储系统宕机后数据丢失，但是其数据可以通过区块链系统恢复。考虑这与数据写入逻辑的具体设计相关，本节对此不再详细讨论。

3.4.3　加速方法

由于在同步中间件中，更新日志的抓取模块和解析模块的设计，高度依赖系统中所使用的关系型数据库类型。本节将主要介绍该方法的同步控制模块的设计，以及区块链智能合约的设计，并讨论该方法的加速原理，给出系统参数的选择建议。

本节将介绍一种高效的链下模式数据同步方法。其中，链下存储系统将使用有防宕机制的 Key-Value 型存储，满足该需求的业界常用存储系统包括：Redis，LevelDB，RocksDB 等。另外，由于更新日志的抓取模块和解析模块的设计，高度依赖系统中所使用的关系型数据库类型，本节将主要介绍该方法的同步控制模块的设计，区块链智能合约的设计，以及中间件提供的数据查询设计，并讨论该方法的加速原理，给出系统参数的选择建议。

3.4.3.1　同步控制模块的设计

如图3.4所示，同步控制模块由合并模块和上链模块组成，合并模块与上链模块间通过上链队列相连。

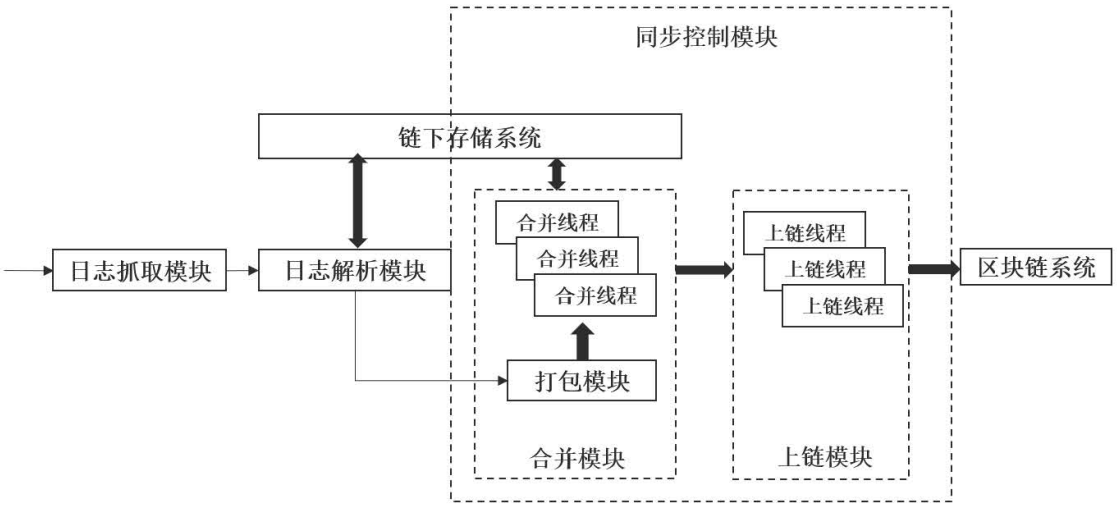

图 3.4 链下模式加速上链的同步控制模块设计

Figure 3.4 Effcient Synchronization Control Module Design(off-chain)

合并模块负责将多条原始更新消息合并为一条更新消息，并将这种原消息与新消息的映射关系写入 Key-Value 型的链下存储模块，最终将更新消息发送到上链队列。

为了便于理解，我们以 Key-Value 存储模型为例，说明一种简单的消息映射

实现方式：更新消息的 Key 将是合并 ID，Value 将是原消息列表中所有 Key 和 Value 将被拼接成的字符串。这种映射关系将在链下被存储为一条 Key-Value 记录，其中 Key 是合并 ID，Value 是所有原始消息的 Key 拼接成的字符串。可以发现，具体的映射方式将是十分多样的。更关键的是，我们还可以将合并后的 Value 进行无损压缩，降低待上链的数据量。

合并模块的参数包括合并线程数、最大合并数和合并超时时间。在中间件启动时，合并模块将开启一个打包线程，以及预设数量的合并线程。打包线程的设计与链上模式的打包模块设计相同，其工作流程是持续消费来自上游解析模块的更新消息，当收集的消息数量达到预设的最大合并数或超时后将触发打包，将该消息列表发送至合并队列。而合并线程将从队列中消费打包后的消息列表，计算消息映射并在链下存储模块写入映射关系，最终将映射后的更新消息推送至上链队列。

上链模块的设计与上文链上模式同步方法所述的上链模块设计相同，由一个 IO 线程与若干个上链工作线程组成。模块的参数包括最大上链线程数以及失败重试时间。IO 线程将持续消费上游合并模块发送的更新消息，开启上链线程将其异步提交到区块链系统。

3.4.3.2 区块链智能合约的设计

区块链系统将提供一个负责单条更新消息写入的智能合约。它的输入参数是待上链的原始消息映射后的更新消息。该写入合约执行时，将直接根据该消息的内容进行区块链系统的数据更新。

在这种设计下，如果区块链系统直接向客户端提供类 SQL 的数据查询合约，存在以下两点障碍：

1. 区块链系统上存储的是合并后的数据，且没有相关索引，必须遍历所有数据才能确认待查询数据的位置；

2. 区块链系统上存储了所有版本的数据，且没有相关索引，必须遍历所有数据才能确认最新版本。

因此，区块链系统将提供一个面向中间件的查询合约，用于返回映射后的消息。它的输入参数将是待查询消息的位置列表。该查询合约执行时，将根据传入的映射后消息的位置，依次读取对应的消息，最终共同返回给中间件。

3.4.3.3　中间件数据查询的设计

使用链下模式的数据同步方法，区块链不提供数据查询的智能合约。客户端应通过中间件提供的类 SQL 查询接口进行数据查询。中间件在执行数据查询时，将分为以下四步：

1. 解析接收到的类 SQL 查询，得到所有候选返回的数据表名、数据行主键、查询条件等信息；

2. 访问链下存储系统，获取所有候选行对应的在区块链系统上的合并数据的位置；

3. 调用区块链系统的批量查询合约，获取包含候选行的合并数据列表；

4. 中间件解合并，根据查询条件过滤候选数据行，将结果返回给客户端。

3.4.3.4　加速原理及参数选择

与链上模式相同的，对更新日志抓取模块，其性能取决于关系型数据库本身，在同步系统层面不可控。对更新日志解析模块，由于可以不考虑事务提交的顺序性，应当开启足量的解析线程提高吞吐量。

对合并模块，它的消息映射机制，相比链上模式的打包设计，不仅将多个原始事务映射到了一个区块链事务，降低了网络传输的开销，而且在区块链事务执行时仅需一次写入，在映射时还可以引入压缩机制进一步降低区块链写入的数据量，大大降低了单个区块链事务的执行开销。相比链上模式同步方法，系统吞吐量将得到有效增加。但是，如果链下存储系统的性能不够高，在最大合并数超过一个阈值后，系统的性能瓶颈可能会从区块链系统转移到合并模块。所以，在实际使用时，应当选用性能尽可能高的链下存储系统，并以此为指导设置最大合并数。此外，合并模块的处理时延也是不可忽视的，需要结合上链时延的期望，共同调整合并超时时间和最大合并数。

对上链模块，应当开启足量的上链线程数，以达到系统的极限性能。

第 4 章　系统实现与性能测试

4.1　MySQL 到 Hyperledger Fabric 数据同步系统的实现

本章将选用业界应用非常广泛的关系型数据库系统 MySQL 与联盟链系统 Hyperledger Fabric，介绍上文所提出的链上、链下模式数据同步方法的具体实现。

4.1.1　MySQL binlog 抓取模块的实现

MySQL binlog 抓取模块的功能是：伪装 MySQL 从节点，模拟 MySQL 的主从复制协议与主节点交互，获取 MySQL 的 binlog event，在其中读取 SQL 查询，将其写入磁盘，最后推入消息队列 Kafka 供下游使用。其中，Kafka 的写入模式是单 topic 单分区。

该模块与 MySQL 主节点的交互逻辑基于开源库 go-mysql[20] 实现，核心代码如下所示：

```
cfg := replication.BinlogSyncerConfig {
    ServerID: // serverid
    Flavor:   "mysql",
    Host:     // host,
    Port:     // port,
    User:     // user
    Password: // password,
}
syncer := replication.NewBinlogSyncer(cfg)
// start from (binlogFile, binlogPos)
streamer, _ := syncer.StartSync(mysql.Position{binlogFile, binlogPos})
for {
    ev, _ := streamer.GetEvent(context.Background())
    // get SQL query
    // write to disk
    // push to Kafka
}
```

MySQL binlog 抓取模块在与 MySQL 建立连接时，需要传递需求的起始位置。该位置由两部分组成：binlog 文件名、在该文件内的位置。根据 MySQL binlog 生成机制，binlog 的文件名前缀将在 MySQL 的配置文件中指定。例如前缀被设置为”mysql-bin”，那么第一个 binlog 日志文件名将为”mysql-bin.000001”。因此，

在初次同步或 MySQL 的该配置项有所改动时，日志抓取模块也需要做出相应的修改。

为了宕机可以更快恢复，binlog 抓取模块需要持续记录当前的 binlog 位置。由于 ROW 格式 SQL 日志具有幂等特性，这里的实现方式是，在内存中维护同步位置，定时将该位置以日志的形式写入磁盘。宕机恢复时从磁盘读取该位置，重新开始同步。

4.1.2 SQL 解析模块的实现

为了降低区块链系统的写入开销，本章所介绍的系统将过滤掉所有建立索引的查询。数据库、数据表、数据行到 Key-Value 模型的映射关系如表4.1所示。其中，Value 是结构化消息，存储时使用 JSON 序列化。

表 4.1 SQL 到 Key-Value 的一种映射

Table 4.1 A kind of SQL model implementation based Key-Value

关系型对象	Key 前缀	Key 生成规则	Value
数据库	d_	Key 前缀 + 数据库名	字符集
数据表	t_	Key 前缀 + 数据库名 + 表名	自增 ID，列信息，主键，外键
数据行	r_	Key 前缀 + 数据库名 + 表名 + 主键	各列名及对应的取值

为了满足 INSERT、UPDATE 语句解析时需要查询表结构的需求，在解析模块引入内存数据库 Redis。在解析过程中，所有 DDL 查询映射成的 Key-Value 记录均将写入至 Redis 中。由于此部分数据宕机不可丢失，Redis 将使用 AOF 持久化模式。

本模块涉及的系统参数包括解析线程数。具体并发控制策略上文已经有所介绍。

4.1.3 链上模式的数据同步系统实现

基于上文所述的链上模式数据同步方法，以及本章所述的通用模块实现，链上模式的数据同步系统架构如图4.1所示。

该系统需要预设的参数包括：解析线程数、最大合并数、合并超时时间、上链线程数、上链失败重试时间。数据同步的全过程及数据结构的变化如下所述：

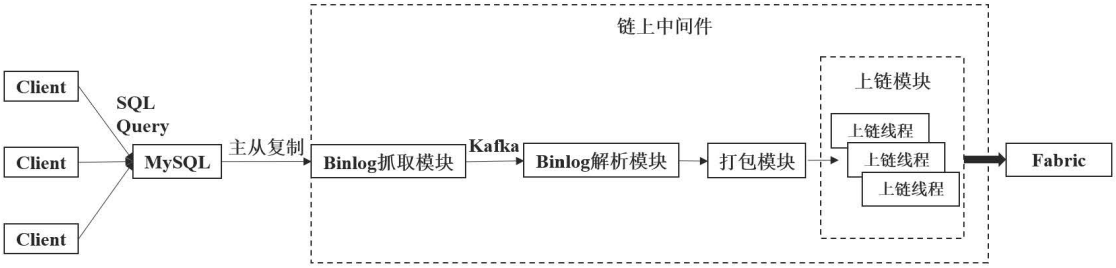

图 4.1 链上模式的数据同步系统实现

Figure 4.1 Implementation of Data Synchronization System(on-chain)

1. 日志抓取模块从 MySQL binlog Event 中提取 SQL 查询。该模块传向下游的消息包括 SQL, binlog 位置两个属性；

2. 解析模块并行地将 SQL 查询解析为 Key-Value 记录，对 DDL 查询需更新 Redis。该模块传向下游的消息包括待上链 Key-Value，binlog 位置两个属性；

3. 打包模块将上游解析后的消息打包。该模块传向下游的消息是解析后的消息列表；

4. 上链模块异步调用 Fabric 智能合约 batchInsert，传入消息列表，将数据写入区块链。

Fabric 的智能合约 batchInsert 核心代码如下所示，其中，对数据是否需要更新，将根据 binlog 位置进行判断。

```
func (t *OnChainSyncChaincode) batchInsert(stub shim.ChaincodeStubInterface, args []string)
    pb.Response {
    kvNum, err := strconv.Atoi(args[0])
    if err != nil {
        return shim.Error("kvNum invalid")
    }
    for i := 0; i < kvNum; i++ {
        kvData := DecodeKvData(args[i + 1])
        err, oldKvData := stub.GetState(kvData.Key)
        if IsNewer(kvData, oldKvData) {
            err = stub.PutState(kvData.Key, kvData.Value)
            if err != nil {
                failMsg := fmt.Sprintf("put kv idx %d fail", i)
                return shim.Error(failMsg)
            }
        }
    }
    return shim.Success(nil)
}
```

4.1.4 链下模式的数据同步系统实现

基于上文所述的链下模式数据同步方法，以及本章所述的通用模块实现，链下模式的数据同步系统架构如图4.2所示。

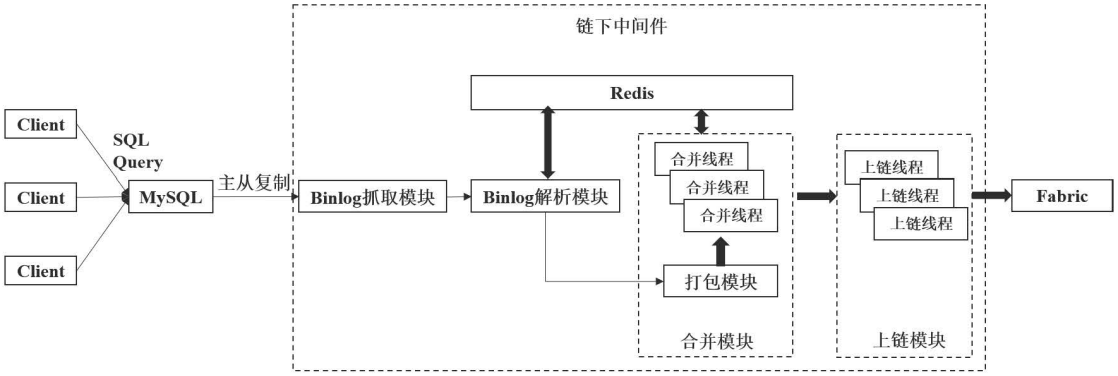

图 4.2 链下模式的数据同步系统实现

Figure 4.2 Implementation of Data Synchronization System(off-chain)

该系统需要预设的参数包括：解析线程数、最大合并数、合并线程数、合并超时时间、上链线程数、上链失败重试时间。数据同步的全过程及数据结构的变化如下所述：

1. 日志抓取模块从 MySQL binlog Event 中提取 SQL 查询。该模块传向下游的消息包括 SQL, binlog 位置两个属性；

2. 解析模块并行地将 SQL 查询解析为 Key-Value 记录，如果是 DDL 查询则更新 Redis。该模块传向下游的消息包括待上链 Key-Value，binlog 位置两个属性；

3. 打包线程将上游解析后的消息打包。该模块传向下游的是解析后的消息列表；

4. 合并线程将消息列表转换为一条 Key-Value 记录。对该记录，Key 为合并块的自增 ID，Value 为原消息列表的序列化后进行 lz4 压缩得到的字符串。而后，合并线程将在链下存储系统逐条尝试更新已经被合并的原始 Key-Value 记录，更新时会根据 binlog 位置判断是否是过时消息，如果是更新的记录，就将（原始 Key，合并 Key）写入。其中，链下存储系统使用 Redis，与 SQL 解析模块共享一个实例。该模块传向下游的消息是待上链的 Key-Value；

5. 上链模块异步调用 Fabric 智能合约 singleInsert，将上游合并后的 Key-Value 记录写入区块链。

其中，singleInsert 合约的逻辑是写入一条 Key-Value 记录到 Fabric。

4.2　性能测试与分析

本章将对上文所述 MySQL 到 Hyperledger Fabric 的数据同步系统进行性能测试与分析，评价本文提出的数据同步方法，探索各模块参数与系统性能的关系。

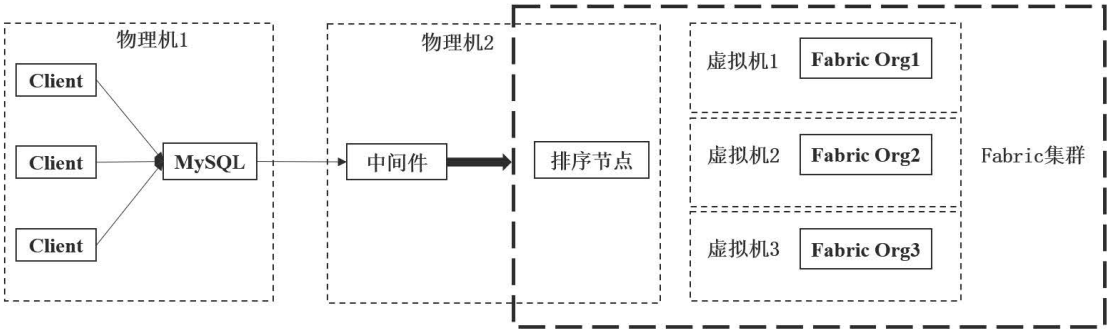

图 4.3 实验集群

Figure 4.3 Cluster in Experiment

4.2.1　实验环境

实验系统中使用的软件包括：MySQL、Hyperledger Fabric，Redis。其中 MySQL 版本为 5.6，Redis 的版本为 5.0.4，Fabric 版本为 1.2。

其中，对 Fabric 设置了三个组织，记为 Org1，Org2，Org3。上链合约所在的 channel 将包含 Org1、Org2 两个组织。Fabric 的出块相关配置为：Perferred Block Size 为 512KB，Max Block Message 为 200，Batch Timeout 为 2 秒。

实验集群的整体情况如图4.3所示。MySQL 与模拟负载的客户端将在物理机 1 上运行。物理机 1 的配置为：CPU 为 Intel Xeon E5620(2.40GHz)，16 核，内存为 32GB，操作系统为 Centos7.2。中间件、排序节点将同时部署在物理机 2 上。物理机 2 的配置为：CPU 为 Intel Xeon E5-2620v2(2.10GHz)，24 核，内存为 94GB，操作系统版本为 Centos7.6。三个组织节点分别部署在三台虚拟机上，其宿主机 CPU 为 Intel Xeon E5610(2.40GHz)，每台虚机分配 4 核，内存 16GB，操作系统版本为 Centos7.6。

4.2.2 使用负载介绍

Voter[21] 是一个模拟网络投票的 OLTP 负载，其特点是单个事务较为简单，并发量较高。Voter 使用的 Schema 如表4.2所示。

表 4.2 Voter 使用 Schema

Table 4.2 Schema of Voter

数据表	字段	详情
Contestants	c_number(int32), c_name(varchar)	参赛者信息，姓名及编号
Area_code	area_code(int32), state(varchar)	各州信息，名称及编号
Votes	phone_number(bigint), state(varchar), c_number(int32)	投票信息，投票人电话，所属州及所投的参赛者编号

Voter 负载中共设计了三个事务。其中只有第一个事务是数据更新事务，用于模拟电话投票，它将造成的数据更新是插入一条投票信息到 Votes 数据表。第二个事务用于统计各州的投票热度图。第三个事务用于计算目前的参赛者排行榜。实际测试的流程是：同时开启多个客户端，每个客户端将开启一个线程模拟用户，每个用户线程将持续向待测试数据库发起投票请求。另外，对两个统计事务，实验系统将每秒调用一次，刷新排行榜和投票热度图。

对 OLTP 负载而言，不同的 Schema 或不同的事务设计间的复杂度差别极大。对于一些其他的常用 OLTP 负载，如 TPC-C[22]，复杂度极高，传统的关系型数据库处理性能都很低。以 MySQL 为例，每分钟处理的事务数一般在数千的级别。在使用这类负载的情况下，关系型数据库与区块链系统间不存在明显的性能差距，难以体现本文提出的数据加速上链策略的意义。因此，在本文的实验中，将使用 Voter 的 Schema 和数据更新事务对本文提出的数据同步方法进行性能测试。

4.2.3 SQL 解析模块性能测试

本节将对 SQL 解析模块进行性能测试。根据本文提出的方法，解析模块将把原始的 SQL 事务转换为 Key-Value 更新消息。本模块需要设定的系统参数为使用的解析线程数。

本节的测试方法是，预先把足量的 Voter 投票事务推入解析队列，记录在一定时间内解析模块所处理的 SQL 事务量和总时延，计算 TPS 及平均时延作为对解析模块的性能评价标准。

调整解析线程数从 5 到 50（间隔 5）进行测试，测试结果如图4.4所示。我们可以发现：

1. 随着解析线程数的增加，解析模块 TPS 先增加后不变；

2. 随着解析线程数的增加，解析的平均时延先降低后不变；

3. 满载情况下，峰值 TPS 大约为 40000，最低平均时延大约为 0.025 毫秒。

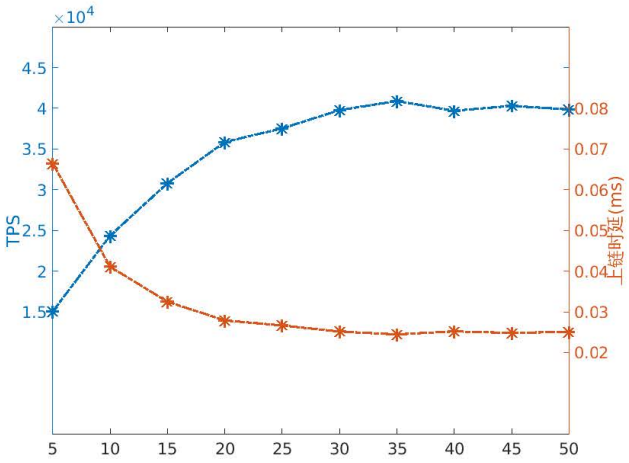

图 4.4 SQL 解析模块性能

Figure 4.4 Performance of Parsing Module

4.2.4　基准方法测试

为了说明本文所提出数据同步方法的加速效果，本节将介绍一个基准同步方法，并对它进行性能测试。

在基准同步方法中，待同步的 MySQL 数据表将使用 Voter Schema，数据将源自 Voter 中的投票事务。Fabric 将提供一个接受单个 Key-Value 写入的智能合约。

基准方法数据同步的全流程是：同步系统实时获取 MySQL 的 binlog，将每行更新日志通过 SQL 解析模块进行解析，得到一个待写的 Key-Value 对，最后开启一个上链线程异步调用智能合约，将该 Key-Value 对写入区块链。

在解析模块限定输入 TPS 为 800，设定上链线程数为 500 进行测试，同步 100 秒内 TPS 和上链时延如图4.5所示。我们可以发现，基准方法的极限同步 TPS 约为 540，最低上链时延约为 2 秒。

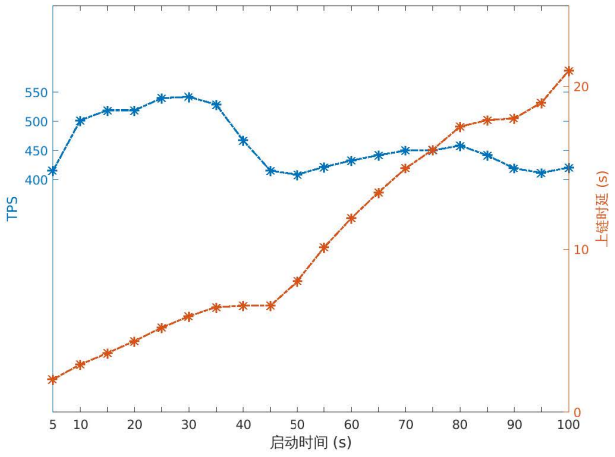

图 4.5 基准方法性能

Figure 4.5 Performance of Baseline

4.2.5 链上模式同步方法性能测试

本节将对上文提出的链上模式的数据同步方法进行性能测试，研究实际场景下的同步性能以及各系统参数对同步性能的影响。

由于合并模块的逻辑极为轻量，仅将足够数量的 Key-Value 更新消息（或超时触发）统一发送给下游。因此，合并模块显然不是系统的性能瓶颈，引入时延忽略不计，对全链路的性能测试性能结果可以认为是解析模块和上链模块串联的性能测试结果。

4.2.5.1 实时同步测试

本节将使用 Voter 负载进行实时同步测试，在解析队列处将进行限流，调整每秒输入系统的原始 SQL 事务数。根据上文介绍以及对各模块的性能测试结果，本节将研究最大合并数、上链线程数、输入 TPS 三个参数对链下模式同步方法性能的影响。

第一个实验将研究最大合并数对系统性能的影响。限定输入 TPS 为 8000，设定解析线程数为 10，合并超时时间为 1 秒，上链线程数为 50，调整最大合并数从 100 到 500（间隔 100）进行测试，同步开始 100 秒内 TPS 如图4.6所示，上链平均时延变化趋势如图4.7所示。我们可以发现：

1. 系统经过约 20 秒的启动时间达到峰值 TPS，而后同步速度趋于稳定；

2. 随着最大合并数的增加，TPS 先增加后减少。在本实验的参数设定下，最

大合并数为 200 时达到 TPS 峰值约 7800，跟上输入负载；

3. 随着最大合并数的增加，上链时延先减少后增大。这是由于同步速度跟不上输入而导致了大量任务积压。而在能跟上的情况下，合并数越小，时延越低。

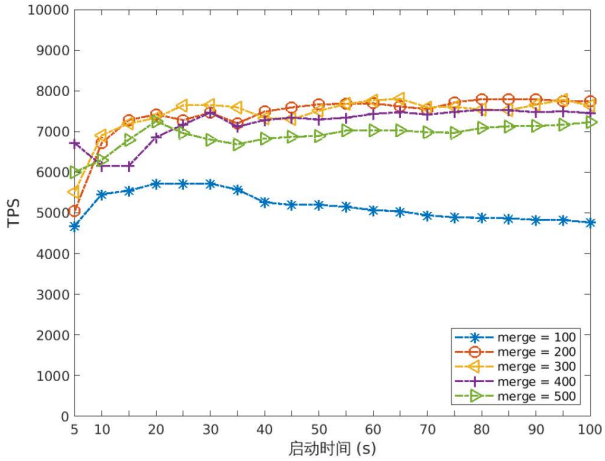

图 4.6 链上模式实时同步测试：最大合并数与 TPS 的关系

Figure 4.6 Real-time synchronization(on-chain): TPS vs Max Merge

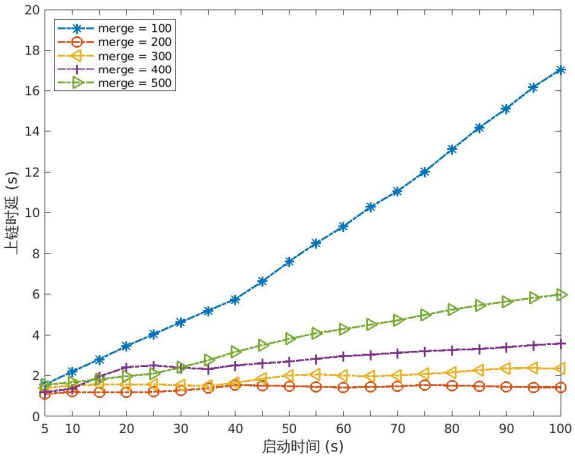

图 4.7 链上模式实时同步测试：最大合并数与上链时延的关系

Figure 4.7 Real-time synchronization(on-chain): Latency vs Max Merge

第二个实验将研究上链线程数对系统性能的影响。限定输入 TPS 为 8000，设定解析线程数为 10，最大合并数为 200，合并超时时间为 1 秒，调整上链线程数从 20 到 60（间隔 10）进行测试，同步开始 100 秒内 TPS 如图4.8所示，上链平均时延变化趋势如图4.9所示。我们可以发现：

1. 随着上链线程数的增加，TPS 先增加再不变。上链线程数达到一定值后（50），同步速度跟上输入负载；

2. 随着上链线程数的增加，时延先减少后不变。上链线程数达到一定值后（50），上链时延趋于稳定。

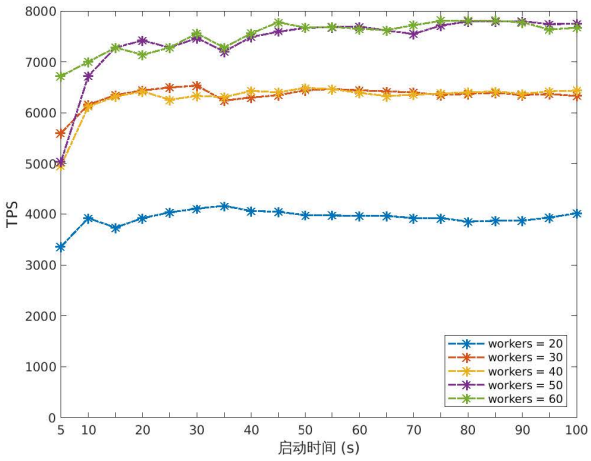

图 4.8 链上模式实时同步测试：上链线程数与 TPS 的关系

Figure 4.8 Real-time synchronization(on-chain): TPS vs Submit Workers

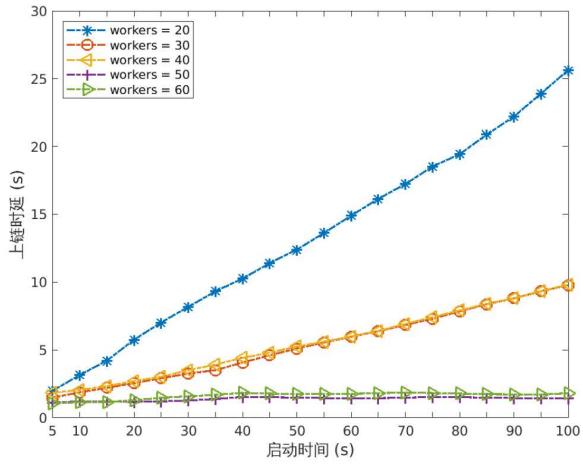

图 4.9 链上模式实时同步测试：上链线程数与上链时延的关系

Figure 4.9 Real-time synchronization(on-chain): Latency vs Submit Workers

第三个实验将研究负载对系统性能是否有影响，以及系统的极限写入性能。限定输入 TPS 为 8000，设定解析线程数为 10，最大合并数为 200，合并超时时间为 1 秒，上链线程数为 50。调整输入 TPS 从 2000 到 10000（间隔 2000）进行

测试，同步开始 100 秒内 TPS 如图4.10所示，上链平均时延变化趋势如图4.11所示。我们可以发现：

1. 系统极限 TPS 约为 8000，在该极限内的各级别流量下可以满足实时同步的需要；

2. 在系统极限以内，输入 TPS 对上链时延没有明显影响。

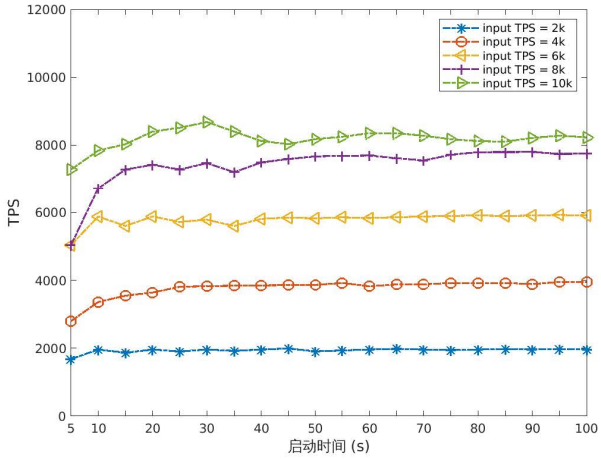

图 4.10 链上模式实时同步测试：负载与 TPS 的关系

Figure 4.10 Real-time synchronization(on-chain): TPS vs Input TPS

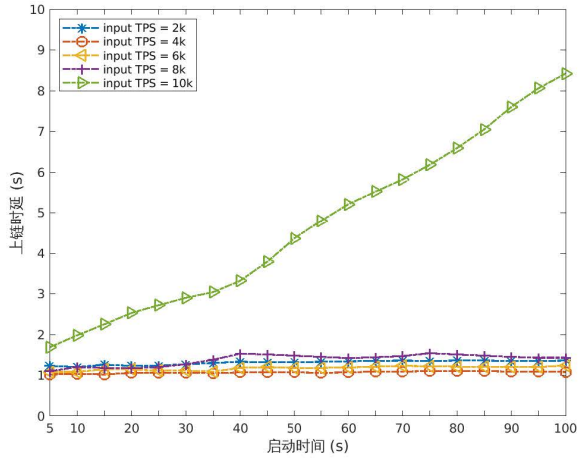

图 4.11 链上模式实时同步测试：负载与上链时延的关系

Figure 4.11 Real-time synchronization(on-chain): Latency vs Input TPS

4.2.5.2 链上模式同步方法实验小结

根据上文所述的测试结果，本文提出的链上数据同步方法，系统极限 TPS 约为 8000，相比基准方法同步速度提高了 16 倍。上链时延约为 1 秒，具有较强的实时性。在极限以内，系统可以适应各级别的负载，在同步过程中同步速度和上链时延可以保持稳定。

在实际使用进行参数设置时，最大合并数必须设定合理，设定过小或过大都将导致同步 TPS 不佳。解析线程数和上链线程数要结合负载和 CPU 情况设置足量。

根据解析模块的性能测试结果，解析模块 TPS 峰值约为 40000，而系统极限 TPS 约为 8000。这意味着链上数据同步方法的性能瓶颈在于 Fabric。这一点可以通过对 Fabric 的性能改进或调整 Fabric 出块参数得到改善，这类改进不在本文的讨论范围之内。

4.2.6 链下模式同步方法性能测试

本节将对上文提出的链下模式同步方法进行性能测试，主要方法和目标为：

1. 通过对合并模块、上链模块的满载性能测试，确定系统的性能瓶颈；

2. 通过全链路的实时同步测试，探索最大合并数、合并线程数、上链线程数等核心系统参数对系统性能的影响；

3. 通过在解析队列限流并调整负载进行测试，探索系统的极限同步性能，确定系统在极限内的各级别负载下中都有较好的同步性能。

4.2.6.1 合并模块满载性能测试

本节将对链下模式的合并模块进行性能测试。根据本文提出的方法，合并模块将把上游解析后的多条 Key-Value 更新消息合并为一条 Key-Value 更新消息，并将旧 Key 与新 Key 的映射关系存储在中间层的 Redis 中。本模块需要设定的系统参数主要是最大合并数和合并线程数。

本节的测试方法是，预先把足量的 Voter 投票事务解析后的 Key-Value 更新消息推入合并打包队列，记录在一定时间内合并模块所处理的 Key-Value 更新消息量和总时延，计算 TPS 及平均时延作为对合并模块的性能评价标准。

根据统计，上游更新消息平均 Key 长度约为 10 字节，Value 平均长度约为 150 字节。设定合并超时时间为 1 秒，调整最大合并数从 200 到 1000（间隔 200），

合并线程数从 1 到 10（间隔 1）进行测试，测试结果如图4.12和图4.13所示。我们可以发现：

1. 随着合并线程数的增加，合并模块 TPS 增加，平均时延降低；
2. 最大合并数的改变对 TPS 无明显影响，但合并数越小，平均时延越小；
3. 满载情况下，TPS 峰值大约为 27000，平均时延最低不超过 10 毫秒。

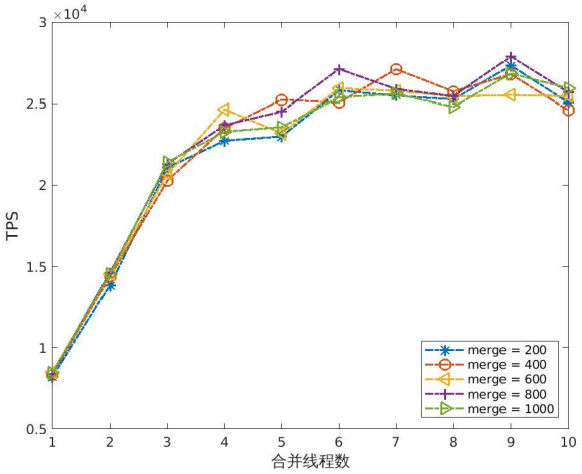

图 4.12 链下模式合并模块 TPS

Figure 4.12 TPS of Merging Module(off-chain)

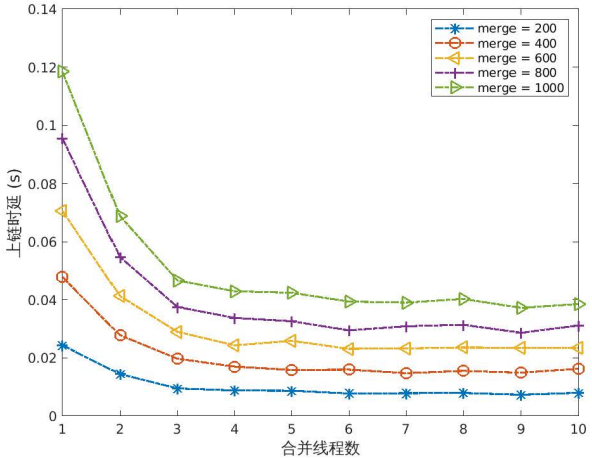

图 4.13 链下模式合并模块时延

Figure 4.13 Latency of Merging Module(off-chain)

4.2.6.2 上链模块满载性能测试

本节将对链下模式的上链模块进行性能测试。根据本文提出的方法，上链模块将把上游合并后的 Key-Value 更新消息通过调用 Fabric 智能合约写入区块链。本模块需要设定的系统参数主要是最大上链线程数。

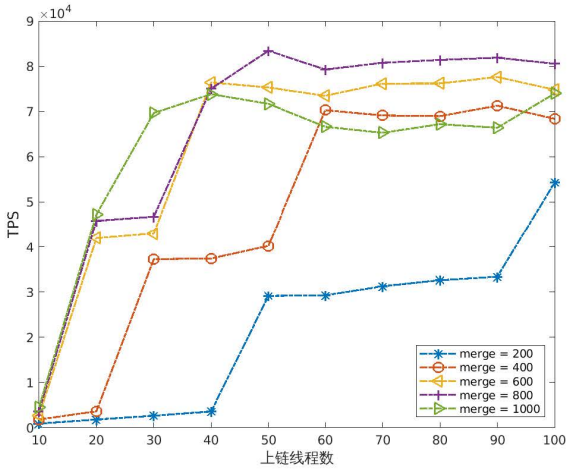

图 4.14 链下模式上链模块 TPS

Figure 4.14 TPS of Submitting Module(off-chain)

本节的测试方法是，预先将足量 Voter 投票事务解析、合并后的消息推入上链队列，记录一定时间内上链模块写入区块链的原始事务数和总时延，计算 TPS 及平均时延作为对上链模块的性能评价标准。

除了最大上链线程数外，对上链模块，影响性能的重要参数为上游传入的合并 Key-Value 消息的大小。根据本文的合并策略及本节测试方法，可以认为合并前更新消息平均 Key 长度约为 10 字节，Value 平均长度约为 150 字节，合并后的 Key 长度不超过 10 字节，Value 平均长度与块实际合并数事务数成正比。因此，本节使用上游合并模块的最大合并数作为单条上链消息大小的衡量。

调整最大合并数从 200 到 1000（间隔 200），合并线程数从 10 到 100（间隔 10）进行测试，测试结果如图4.14和4.15所示。我们可以发现：

1. 随着最大合并数、最大上链线程数的增加，上链模块 TPS 增加。当最大合并数和最大上链线程数超过一定值时，TPS 达到峰值然后不再改变；

2. 随着最大合并数和最大上链线程数的增加，上链模块时延降低。当最大合并数和最大上链线程数超过一定值时，时延达到最低值不再改变；

3. 满载情况下，上链模块峰值 TPS 超过 80000，平均时延最低不超过 10 毫秒。

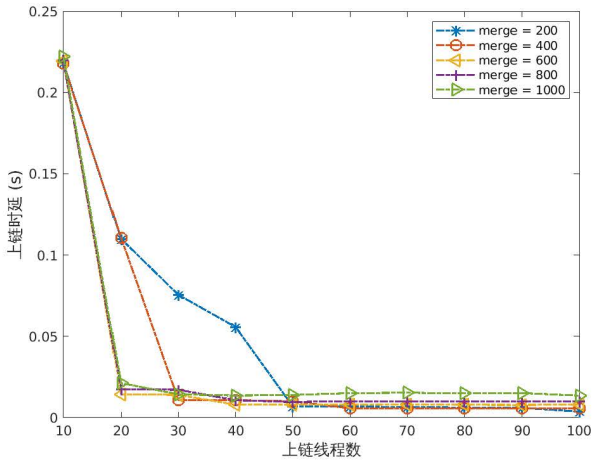

图 4.15 链下模式上链模块时延

Figure 4.15 latency of Submitting Module(off-chain)

4.2.6.3　实时同步测试

本节将对上文提出的链下模式数据同步方法进行全链路的性能测试。本节的测试方法是，使用 Voter 负载进行模拟同步测试，并在解析队列处进行限流，调整每秒输入系统的原始 SQL 事务数，探索系统在各级别流量下的同步表现及系统各模块参数对性能的影响。根据上文介绍以及对各模块的性能测试，本节将研究最大合并数、合并线程数、上链线程数、输入 TPS 四个参数对链下模式同步方法性能的影响。

第一个实验将研究最大合并数与系统性能的关系。参考上文测试结果，限定解析队列输入 TPS 为 15000，设定解析线程数为 10，合并线程数为 16，合并超时时间为 1 秒，上链线程数为 50。调整最大合并数从 200 到 1000（间隔 200）进行测试，同步开始的 100 秒内 TPS 如图4.16所示，上链平均时延变化趋势如图4.17所示。我们可以发现：

1. 系统有一定的启动时间（10 秒左右），而后 TPS 达到峰值并趋于平稳；

2. 随着最大合并数的增加，峰值 TPS 先增加后不变。最大合并数达到一定值（400）后，同步速度跟上输入速度；

3. 随着最大合并数的增加，平均时延先减少后增大。最大合并数达到一定

值（400）后，同步时延趋于稳定。

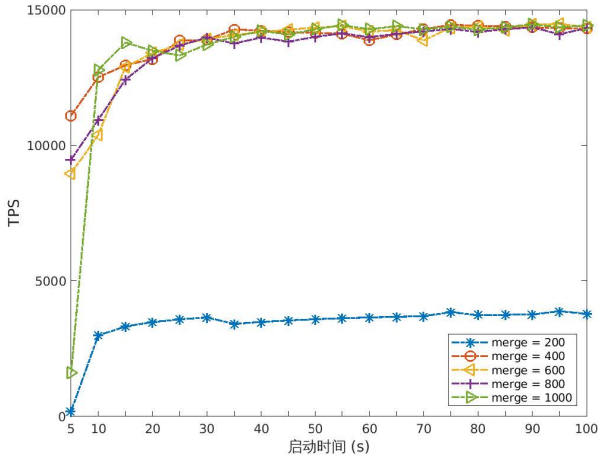

图 4.16 链下模式实时同步测试：最大合并数与 TPS 的关系

Figure 4.16 Real-time synchronization(off-chain): TPS vs Max Merge

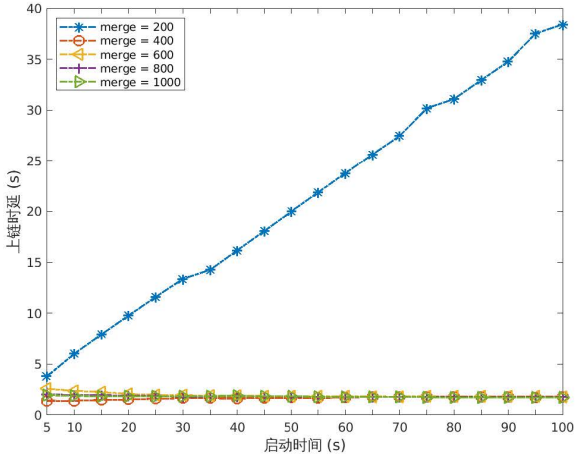

图 4.17 链下模式实时同步测试：最大合并数与上链时延的关系

Figure 4.17 Real-time synchronization(off-chain): Latency vs Max Merge

第二个实验将研究合并线程数与系统性能的关系。参考上文测试结果，限定解析队列输入 TPS 为 15000，设定解析线程数为 10，最大合并数为 400，合并超时时间为 1 秒，上链线程数为 50。调整上链线程数从 2 到 10（间隔 2）进行测试，同步开始 100 秒内 TPS 如图4.18所示，上链平均时延变化趋势如图4.19所示。我们可以发现：

1. 随着合并线程数的增加，TPS 先增加再不变。合并线程数达到一定值后（6），同步速度跟上输入负载；

2. 随着合并线程数的增加，时延先减少后不变。合并线程数达到一定值后（6），同步时延趋于稳定。

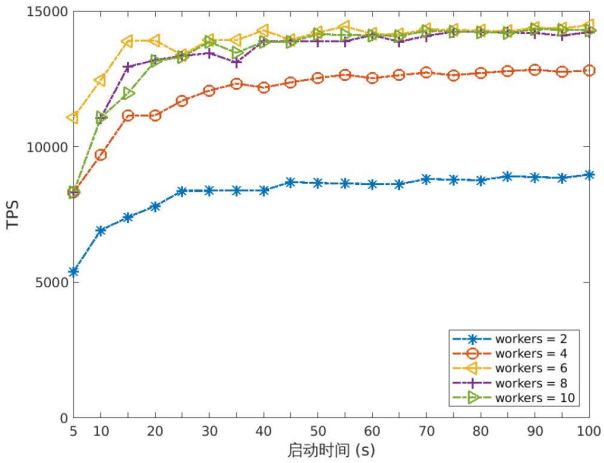

图 4.18 链下模式实时同步测试：合并线程数与 TPS 的关系

Figure 4.18 Real-time synchronization(off-chain): TPS vs Merge Workers

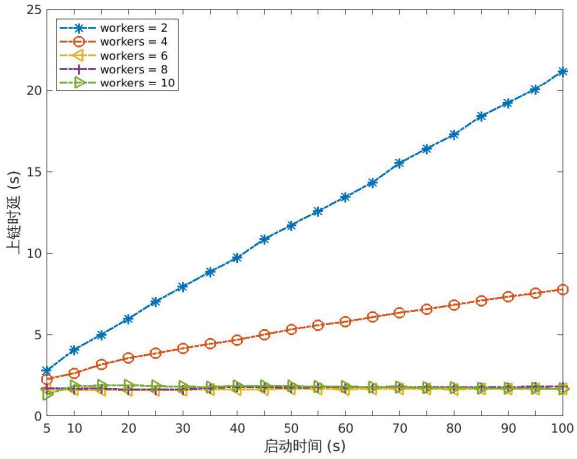

图 4.19 链下模式实时同步测试：合并线程数与上链时延的关系

Figure 4.19 Real-time synchronization(off-chain): Latency vs Merge Workers

第三个实验将研究上链线程数与系统性能的关系。参考上文测试结果，限定解析队列输入 TPS 为 15000，设定解析线程数为 10，最大合并数为 400，合并线程数为 16，合并超时时间为 1 秒。调整上链线程数从 10 到 50（间隔 10）进行测试，同步开始 100 秒内 TPS 如图4.20所示，上链平均时延变化趋势如图4.21所示。我们可以发现：

1. 随着上链线程数的增加，TPS 先增加再不变。上链线程数达到一定值后（40），同步速度跟上输入负载；

2. 随着上链线程数的增加，时延先减少后不变。上链线程数达到一定值后（40），同步时延趋于稳定。

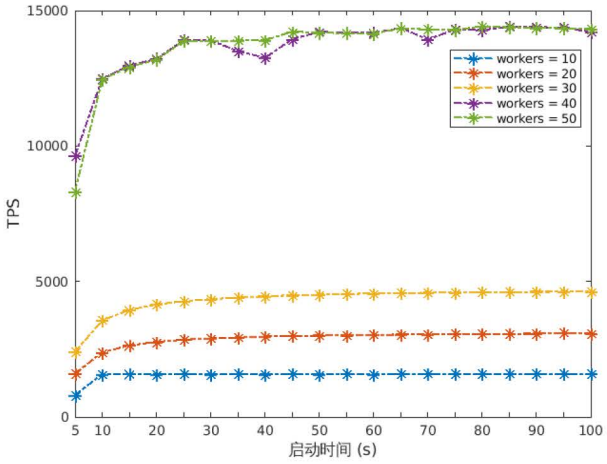

图 4.20 链下模式实时同步测试：上链线程数与 TPS 的关系

Figure 4.20 Real-time synchronization(off-chain): TPS vs Merge Workers

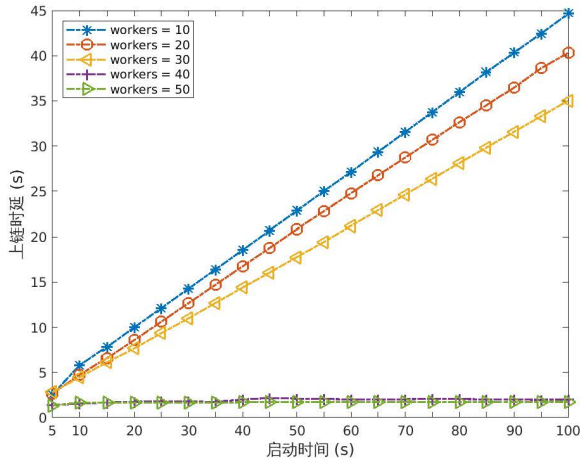

图 4.21 链下模式实时同步测试：上链线程数与上链时延的关系

Figure 4.21 Real-time synchronization(off-chain): Latency vs Merge Workers

第四个实验将研究负载是否会影响系统性能，以及系统的极限写入性能。参考上文测试结果，设定解析线程数为 10，最大合并数为 400，合并线程数为 16，合并超时时间为 1 秒，上链线程数为 50。调整输入 TPS 从 6000 到 18000（间隔

3000）进行测试，同步开始 100 秒内 TPS 如图4.22所示，上链平均时延变化趋势如图4.23所示。我们可以发现：

1. 系统极限 TPS 约为 16000，在该极限内的各级别流量下可以满足实时同步的需要；

2. 在系统极限以内，输入 TPS 越大，系统平均时延越低。

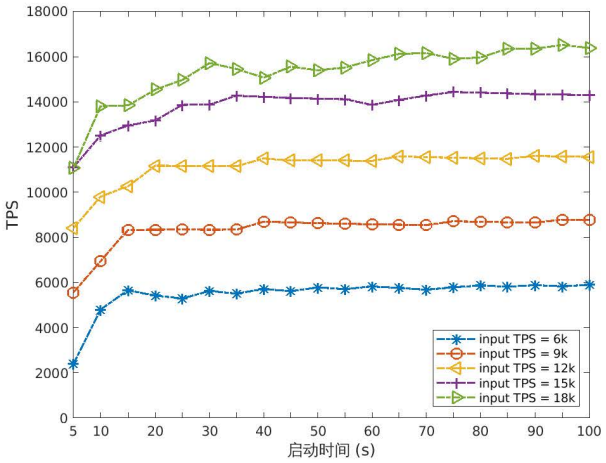

图 4.22 链下模式实时同步测试：负载与 TPS 的关系

Figure 4.22 Real-time synchronization(off-chain): TPS vs Workload

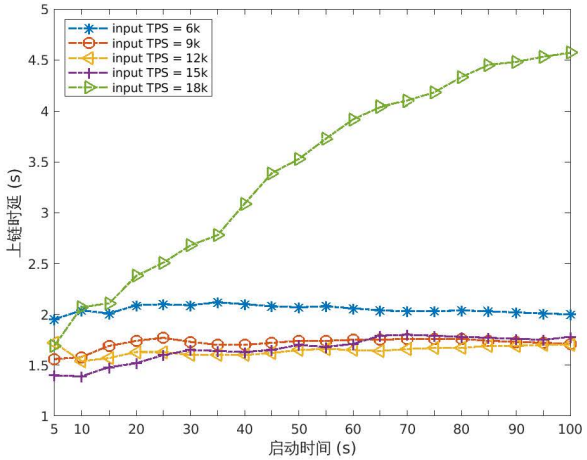

图 4.23 链下模式实时同步测试：负载与上链时延的关系

Figure 4.23 Real-time synchronization(off-chain): Latency vs Workload

4.2.6.4 链下模式同步方法实验小结

根据上文所述的测试，本文提出的链下数据同步方法，系统极限 TPS 约为 16000，相比前序工作 [17]（TPS 14000）性能提升 14%。上链时延约为 2 秒，具有较强的实时性。在极限以内，系统可以适应各级别的负载，在同步过程中同步速度和上链时延可以保持稳定。

在实际使用进行参数设置时，最大合并数设定过小将导致同步 TPS 不佳，设定过大将会使同步时延变高。合并超时时间可以保证合并模块的时延可控，但是过小的设定可能导致各合并块中包含的事务数量远低于最大合并数，间接影响同步 TPS。解析线程数、合并线程数、上链线程数要结合负载和 CPU 情况设置足量。

在实际使用进行参数设定时，如果同步 TPS 不达预期，应尝试增大最大合并数、合并线程数和上链线程数。

根据上文所述的各模块满载测试结果，在满载情况下，解析模块 TPS 约为 40000，合并模块 TPS 约为 27000，上链模块 TPS 约为 80000。各模块间通过队列串联，上游的输出 TPS 即下游的输入 TPS。这意味着上链模块将无法达到极限性能。如图4.14所示，当 Fabric 每秒接收到的数据量（与最大合并数和并发数成正比）较小时（如上链线程数小于 40），写入性能将有断崖式的下跌。这是因为 Fabric 本身的出块策略为收集到足够多的数据（实验设定为 512KB 以上）或超时（实验设定为 2 秒）后出块。这直接导致了，在实际同步测试时，系统极限 TPS 约为 16000，也没有达到各解析、合并模块的极限性能。

经过进一步的分析，我们发现在实际同步的过程中，真正的性能瓶颈在于解析模块和合并模块对 Redis 的访问。Redis 是单线程写入的内存数据库，在本文的实验环境下测试，Redis TPS 约为 50000。这将是本方法后续工作的重点。

另外，事务的上链平均时延约为 2 秒。根据上文的模块满载测试结果，解析模块的单事务处理时延约为 0.025 毫秒，测试中合并数一般不超过 1000，整体时延大约为几十毫秒。合并模块的时延大约为几十毫秒。因此，在整个链路中，绝大多数的时间花费在上链模块。满载情况下，上链模块的时延最低可以达到几毫秒。这也是由于解析、合并模块的性能有限，上链模块未满载导致性能断崖式下跌。

4.2.7　本章小结

本章介绍了 MySQL 到 Hyperledger Fabric 的链上、链下两种模式的数据同步系统实现，并对系统进行了详尽的极限性能测试，分析了各方法的性能瓶颈，研究了系统性能与各参数间的关系，相应给出了实际使用中的参数调整建议。

在本文的测试环境下，对于链上模式的数据同步方法，系统极限 TPS 约为 8000，相比基准提升 16 倍，上链时延约为 1 秒，各级别负载下均表现稳定，性能瓶颈在于上链模块。对于链下模式的数据同步方法，系统极限 TPS 约为 16000，相比前序工作提高 14%，上链时延约为 2 秒，各级别负载下均表现稳定。其性能瓶颈在于解析、合并模块对中间层 Redis 的访问，而上链模块的满载性能实际可达 80000 以上，有较大的改进空间和潜力。

第 5 章　总结与展望

5.1　本文工作总结

在区块链技术目前的发展阶段，直接使用区块链系统替换既有 IT 系统存在两大问题：

1. 既有业务逻辑本身没有变化，但需要迁移，带来极高的成本；

2. 相比传统系统中存储最常用的关系型数据库，区块链系统事务处理性能低，不适用一些更新频繁的场景。因此，一种更为现实的应用模式是：原有操作流程不变，共同使用区块链系统与遗留 IT 系统，并实时地将数据库的全部或部分数据同步到区块链系统。

针对这一问题，通过对现有研究工作的检索，本文研究了基于中间件加速上链的方法，提出了数据同步的两种模式：

1. 链上模式。全部待上链数据均存储在区块链，区块链系统本身可以提供数据查询能力；

2. 链下模式。在保证数据安全的前提下，允许部分数据存储在中间层，区块链系统本身可以不具备完整的数据查询能力，而需要通过中间层来获取数据。

在不保证事务更新的顺序性、原子性，仅保证最终一致性的前提下，对两种模式，本文各提出了一种高效的数据同步方法，并使用 MySQL 和 Hyperledger Fabric 进行了数据同步系统的实现。经测试，在本文的实验环境下，链上模式的极限 TPS 达到 8000，相比直接上链的基准方法性能提高 16 倍；链下模式的极限 TPS 达到 16000，相比目前同模式最佳方法提高 14%。

5.2　下一步研究计划

目前本文的研究工作还有许多不完善之处：

1. 无论是链上还是链下模式的同步方法，都不能保证事务的原子性。在复杂原子事务较多的场景下，这可能极为影响使用体验。如金融系统的转账场景，可能会有一方已经扣款，另一方迟迟未收到转账的情况；

2. 数据查询的支持不完善。本文研究的数据同步系统，以同步速度为第一考量，对于数据查询的性能仅进行了时间复杂度的分析；

3. 特别对于链下模式，同步性能仍有较大进步空间。实验结果显示，本文实现的 MySQL 到 Hyperledger Fabric 的数据同步系统，链下模式同步方法的性能瓶颈在于对中间层存储 Redis 的访问，而远远没有达到 Fabric 的满载状态。

对上述问题，下一步的研究可以从以下角度出发：

1. 结合实际场景，探索如何加入一定程度的原子性保证。值得注意的是，这可能需要在获取数据更新的机制层面做出改进。如果使用 ROW 模式 SQL 更新日志，原子事务对数据同步系统是不可见的。

2. 探索如何设计适用区块链系统的关系型数据查询语言。经过同步，区块链系统存储了关系型模型的数据，但考虑其系统特性与关系型数据库有较大差别，一种较合理的方法是提供类 SQL 查询的支持。

3. 优化链下模式同步方法解析模块、合并模块性能。目前的性能瓶颈是解析模块、合并模块争用中间层存储，而中间层存储处理能力不足。因此，可以从分散数据以降低争用、优化访问频次、使用更高性能的中间层存储系统三个角度进行探索。

参考文献

[1] NAKAMOTO S. Bitcoin: A peer-to-peer electronic cash system[R]. Manubot, 2019.

[2] WOOD G, et al. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum project yellow paper, 2014, 151(2014):1-32.

[3] SUANKAEWMANEE K, HOANG D T, NIYATO D, et al. Performance analysis and application of mobile blockchain[C]//2018 international conference on computing, networking and communications (ICNC). IEEE, 2018: 642-646.

[4] Christidis K, Devetsikiotis M. Blockchains and smart contracts for the internet of things[J]. IEEE Access, 2016, 4:2292-2303.

[5] Zyskind G, Nathan O, Pentland A S. Decentralizing privacy: Using blockchain to protect personal data[C]//2015 IEEE Security and Privacy Workshops. 2015: 180-184.

[6] Caro M P, Ali M S, Vecchio M, et al. Blockchain-based traceability in agri-food supply chain management: A practical implementation[C]//2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany). 2018: 1-4.

[7] Dubovitskaya A, Xu Z, Ryu S, et al. How blockchain could empower ehealth: An application for radiation oncology[C]//VLDB Workshop on Data Management and Analytics for Medicine and Healthcare. 2017: 3-6.

[8] PARK S, KWON A, FUCHSBAUER G, et al. Spacemint: A cryptocurrency based on proofs of space[C]//International Conference on Financial Cryptography and Data Security. Springer, 2018: 480-499.

[9] BENTOV I, GABIZON A, MIZRAHI A. Cryptocurrencies without proof of work[C]// International Conference on Financial Cryptography and Data Security. Springer, 2016: 142-157.

[10] GILAD Y, HEMO R, MICALI S, et al. Algorand: Scaling byzantine agreements for cryptocurrencies[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 51-68.

[11] CHEPURNOY A, DUONG T, FAN L, et al. Twinscoin: A cryptocurrency via proof-of-work and proof-of-stake.[J]. IACR Cryptology ePrint Archive, 2017, 2017:232.

[12] Kokoris-Kogias E, Jovanovic P, Gailly N, et al. Enhancing bitcoin security and performance with strong consistency via collective signing[C]//25th USENIX Security Symposium (USENIX Security 16). 2016: 279-296.

[13] CACHIN C, et al. Architecture of the hyperledger blockchain fabric[C]//Workshop on distributed cryptocurrencies and consensus ledgers: volume 310. 2016: 4.

[14] BROWN R G, CARLYLE J, GRIGG I, et al. Corda: an introduction[J]. R3 CEV, August, 2016, 1:15.

[15] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference. 2014: 305-320.

[16] BIGCHAINDB. Bigchaindb[EB/OL]. 2018. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf.

[17] 王宁. 针对区块链的中间件系统的设计与实现[D]. 中国科学院大学, 2019.

[18] 曾诗钦，霍如，黄韬，刘江，汪硕，冯伟. 区块链技术研究综述：原理、进展与应用[J]. 通信学报, 2020.

[19] Tschorsch F, Scheuermann B. Bitcoin and beyond: A technical survey on decentralized digital currencies[J]. IEEE Communications Surveys and Tutorials, 2016, 18(3):2084-2123.

[20] SIDDONTANG. go-mysql[EB/OL]. retrieved by April 2020. https://github.com/siddontang/go-mysql.

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

[22] TPC. Tpc-c[EB/OL]. retrieved by April 2020. http://www.tpc.org/tpcc/.

[23] THAKKAR P, NATHAN S, VISWANATHAN B. Performance benchmarking and optimizing hyperledger fabric blockchain platform[C]//2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2018: 264-276.

[24] Blossey G, Eisenhardt J, Hahn G J. Blockchain technology in supply chain management: An application perspective[C]//Proceedings of the 52nd Hawaii International Conference on System Sciences. 2019: 1-9.

[25] Saito Y, Shapiro M. Optimistic replication[J]. ACM Computing Surveys, 2005, 37(1):42-81.

[26] Curino C, Jones E, Zhang Y, et al. Schism: a workload-driven approach to database replication and partitioning[J]. very large data bases, 2010, 3(1):48-57.

[27] Li C, Palanisamy B, Xu R. Scalable and privacy-preserving design of on/off-chain smart contracts[C]//2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW). 2019: 7-12.

[28] Xiao Y, Zhang N, Lou W, et al. Enforcing private data usage control with blockchain and attested off-chain contract execution[J]. arXiv preprint arXiv:1904.07275, 2019.

[29] Kubiatowicz J, Bindel D, Chen Y, et al. Oceanstore: an architecture for global-scale persistent storage[J]. architectural support for programming languages and operating systems, 2000, 35 (11):190-201.

[30] Benet J. Ipfs - content addressed, versioned, p2p file system[J]. arXiv preprint arXiv:1407.3561, 2014.

[31] 袁勇，王飞跃. 区块链技术发展现状与展望[J]. 自动化学报, 2016.

[32] 沈鑫，裴庆祺，刘雪峰. 区块链技术综述[J]. 网络与信息安全学报, 2016.

[33] 蔡维德，郁莲，王荣，刘娜，邓恩艳. 基于区块链的应用系统开发方法研究[J]. 软件学报, 2017.

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历

2013 年 9 月-2017 年 6 月，南开大学计算机与控制工程学院，获得工学学士学位。

2017 年 9 月-2020 年 6 月，中国科学院计算技术研究所，攻读工学硕士学位。

已发表 (或正式接受) 的学术论文:

1. Ning Wang, **Bo Wang**, Taoying Liu, Wei Li and Shuhui Yang, A Middleware Approach to Synchronize Transaction Data to Blockchain, ICCCN 2020, August 3 - August 6, 2020, Honolulu, Hawaii, USA

参加的研究项目及获奖情况:

无

致　谢

在计算所度过的三年对我的人生意义非凡。

首先要感谢我的导师刘淘英老师。刘老师治学严谨、平易近人，在学术、生活、为人处事上对我的指导和帮助将让我受益终身。

感谢课题组李伟老师对我学术上的指导和帮助，许多观点让我深受启发。感谢课题组秘书江岩老师对我生活上的帮助。

感谢课题组一起学习、工作的所有同学：王文娟、岳智磊、陈宜涛、贾绍林、王宁、杨峻峰、郭璐璐、郭文飞、俞娟、姜悦怡、卓凤。

感谢我的家人朋友对我一如既往的支持。感谢王淳熙的陪伴，希望我的未来也能与你一同走过。