软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2017,28(3):476−489 [doi: 10.13328/j.cnki.jos.005162]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 集群数据库系统的日志复制和故障恢复*

王嘉豪[1], 蔡 鹏[1,2], 钱卫宁[1], 周傲英[1]

[1](华东师范大学 计算机科学与软件工程学院,上海 200062)

[2](广西可信软件重点实验室(桂林电子科技大学),广西 桂林 541004)

通讯作者: 蔡鹏, E-mail: pcai@sei.ecnu.edu.cn

**摘 要**: 互联网、社交、购物、金融等各类应用直接面临海量用户的高并发访问,传统的单点数据库逐渐成为这些应用系统的瓶颈,而众多互联网应用能够良好运行的主要原因是使用了基于集群环境的数据管理系统作支撑.与传统数据库系统相比,基于集群环境的数据库系统具有更好的扩展性和可用性,而日志复制是保证这些特性的核心组件.传统的主备架构的日志复制在异常情况下对未决事务日志处理不佳,导致数据副本之间存在不一致的风险.另外,分布式系统领域的一致性算法缺乏对事务一致性的处理,而且在选主时存在活锁、多主和频繁选主的问题,无法直接适用于事务日志复制.提出了一种集群环境下的事务日志复制策略和恢复机制,能够有效处理未提交日志,提供了强弱两种读一致性,并且提出一种轻量级的选主算法,可以避免出现以上的选主问题.在开源 OceanBase 分布式数据库系统中实现了上述机制,并使用基准测试工具对系统进行测试,通过一系列实验验证了系统的扩展性和可用性.

**关键词**: 日志复制;故障恢复;一致性;可用性;数据库系统

**中图法分类号**: TP311

中文引用格式: 王嘉豪,蔡鹏,钱卫宁,周傲英.集群数据库系统的日志复制和故障恢复.软件学报,2017,28(3):476−489. http://www.jos.org.cn/1000-9825/5162.htm

英文引用格式: Wang JH, Cai P, Qian WN, Zhou AY. Log replication and recovery in cluster-based database system. Ruan Jian Xue Bao/Journal of Software, 2017,28(3):476−489 (in Chinese). http://www.jos.org.cn/1000-9825/5162.htm

## Log Replication and Recovery in Cluster-Based Database System

WANG Jia-Hao[1], CAI Peng[1,2], QIAN Wei-Ning[1], ZHOU Ao-Ying[1]

[1](School of Computer Science and Software Engineering, East China Normal University, Shanghai 200062, China)

[2](Guangxi Key Laboratory of Trusted Software (Guilin University of Electronic Technology), Guilin 541004, China)

**Abstract**: Many applications such as social networking, online shopping and online finance may receive highly concurrent data access from massive Internet users. In this scenario, traditional single node database systems gradually become the bottleneck of the system, and the main reason for many successful Internet applications is the use of cluster-based data management systems. Compared with traditional database systems, cluster-based distributed database systems have better scalability and availability, and log replication is one of the core components to build these features. Master-slave based log replication cannot handle the uncertain logs while failure occurs, resulting in the risk of inconsistency among different copies. Consensus algorithms cannot be directly applied to the database system due to the lack of transaction consistency model, and they also have issues in leader election with livelock, as well as double master and continuous election problem. This paper introduces a log replication strategy and corresponding recovery technique for cluster environments, which can effectively process the uncertain logs and provide two read consistency options, i.e. strong and weak consistency. A lightweight

* 基金项目: 国家高技术研究发展计划(863)(2015AA015307); 国家自然科学基金(61332006, 61432006, 61672232); 广西可信软件重点实验室研究课题(kx201602)

Foundation item: National High-Tech R&D Program of China (863) (2015AA015307); National Natural Science Foundation of China (61332006, 61432006, 61672232); Guangxi Key Laboratory of Trusted Software (kx201602)

收稿时间: 2016-07-30; 修改时间: 2016-09-14; 采用时间: 2016-11-01; jos 在线出版时间: 2016-11-29

CNKI 网络优先出版: 2016-11-29 13:35:12, http://www.cnki.net/kcms/detail/11.2560.TP.20161129.1335.014.html

master election algorithm is also presented to avoid the master election issues. The algorithms are implemented in the OceanBase distributed database system and tested using benchmark tool. Experiments show that the proposed method can improve the scalability and availability.

**Key words**: log replication; recovery; consistency; availability; database system

随着近年来互联网的发展,互联网应用所处理的数据量增长迅速,数据管理系统需要高效地处理高并发数据访问请求,同时确保系统稳定可靠.在互联网场景下,传统的单点数据库逐渐成为系统的瓶颈,不能很好地满足应用的需求,而众多互联网应用能够良好运行的主要原因是使用了基于集群环境的数据管理系统作为支撑.基于集群环境的数据库系统具有良好的扩展性和可用性,而日志复制是保证这些特性的核心组件.通过日志复制,数据的更新操作可以发送到分布式系统的各个节点上,从而使得数据能够在多个节点上访问,提高了系统的可扩展性.在某些节点发生故障时,系统可以从其他节点上访问到该数据,从而保证了系统的可用性.

传统的数据库系统大多使用集中式的数据存储,不需要日志复制.而在集群数据库系统中,许多节点处于同一个集群,为了避免单点故障,每份数据需要多个副本存放在不同节点上,并通过日志复制,保证数据副本之间的一致性.不同的日志复制策略决定了数据库事务在副本之间采取的不同一致性级别,主要分为强一致性、弱一致性和最终一致性这 3 种[1].强一致性需要保证在事务提交时数据副本之间保持一致;而弱一致性和最终一致性在事务已经提交的情况下,数据副本之间可能有不一致的情况存在.

在基于集群环境的分布式数据库系统中,单个节点发生机器故障是常见现象,在节点异常情况下可能产生未提交事务日志,由于这些日志对应事务的提交状态是不确定的,因此我们将这部分日志称为未决日志,其对应的事务称为未决事务.如何有效处理故障节点的未提交日志是个难题[2].例如,在经典的主备架构(master-slave)中,主节点提供写服务,通过异步的日志复制将修改同步到备节点,假设某一时刻主节点发生故障,主节点故障之前正在进行一部分事务日志的复制,备节点无法确定是否完整地收到了这部分未决日志,只有询问重启后的主节点才能得到确认,而这样做牺牲了可用性;如果备节点直接代替主节点提供服务可能会丢失事务日志,从而导致主备数据库出现不一致的状态.在多主架构(master-master)中,每个节点都可以提供写服务,当某些节点故障时,还可以通过其他节点继续写,并且通过合并不同节点上的同一纪录的事务日志,使该纪录最终能够达成一致.因此,这种方式的一致性称为最终一致性,无法满足某些强一致性需求的业务.

由于主备架构和多主架构在一致性方面表现不佳,越来越多的研究人员开始使用基于 Paxos 算法[3]的日志复制策略.该算法首先进行选主,使得系统中有唯一的主节点提供写服务;然后将事务日志复制到多数节点上再提交事务,使得在多数节点存活的情况下,能够保证节点之间数据一致性.但是 Paxos 算法难以理解,工程实践难度大,许多对 Paxos 算法的实现都对其做了不同程度的修改,如 Chubby[4],Raft[5]等.其中,Chubby 是 Google 开发的一款分布式锁服务,在 Google 的多款产品中(如 Bigtable[6])得到了应用,其底层使用了 Paxos 算法,使用了 5 个独立的机器为一组来提供可靠的服务.为了避免 Paxos 的活锁问题,Chubby 采用了选主机制,并通过租约机制来保证主节点存在时,不会再次选出其他主节点.Raft 算法包含选主和日志复制两部分,从算法逻辑上,Raft 算法比 Paxos 算法逻辑简单,理解和实现相对容易.本文的日志复制算法也是对 Paxos 算法的一种改造,通过选主和租约来避免活锁问题,优化执行的效率.

虽然分布式系统领域的一致性算法(如 Paxos,Raft 算法)能够使得分布式节点之间达成一致,但它们无法直接适用于集群数据库系统的日志复制.例如,Raft 算法就存在以下几个问题:第一,Raft 算法只允许读写主节点,无法将读写负载均衡到集群中的其他节点,可能造成单节点压力过重,影响了系统的扩展性;第二,选举算法至少需要半数以上的节点参与,有时候需要执行多轮才能成功,选举时间会随着节点数量的增加而变长,影响了系统可用性;第三,在网络分区情况下可能导致多个主节点的存在,多主无法满足系统一致性的需求;第四,在网络分区的情况下,还可能造成频繁选举的问题.为了解决上述问题,本文的主要贡献包括:(1) 设计了适用于集群环境的日志同步和故障恢复机制,在主节点提供高可靠写服务的同时,主节点和非主节点均可提供读服务,在此基础上对集群环境下的读一致性做了深入讨论和分析.(2) 设计了一种轻量级的主节点选举算法,避免了 Paxos 和 Raft 等一致性算法所具有的活锁、频繁选举和多主等问题;(3) 在开源分布式数据库系统 OceanBase[2,7]中实现

上述方法,并通过大量实验验证该算法能够有效提高系统的扩展性和可用性.

本文第 1 节介绍日志复制的相关工作.第 2 节举例分析日志复制和选主方面的问题.第 3 节、第 4 节为本文重点,详细介绍集群环境下的日志复制和异常情况下的恢复机制.第 5 节分析本文算法的一致性.第 6 节通过实验验证日志复制策略和恢复机制的正确性和高效性.

## 1 相关工作

日志复制策略通常有 4 种类型.

- 第 1 种是使用经典的主备架构模式,例如 MySQL 数据库.这种方式具有延迟低、吞吐量高的优点,但是其缺点是主备节点之间的数据是弱一致性的;此外,主备节点中有故障发生时,单节点服务的风险很高并可能造成数据丢失,因此可用性很低[8,9];
- 第 2 种是采用两阶段提交协议(2PC)[1]来进行原子性的事务提交或者回滚,例如 Hive[10]等.每次事务更新的数据会采用同步请求的方式复制给所有副本,直到所有副本成功更改才能提交事务.这样做保证了数据副本之间的强一致性,但是由于两阶段提交有多次网络交互和等待,这使得系统吞吐量显著降低、延迟升高;并且如果发生故障,两阶段提交会被阻塞,会严重影响了系统的可用性[11];
- 第 3 种是许多分布式数据库,例如 Dynamo[12],PNUTS[13]采用的多主架构,这种方式允许在不同的节点同时写,通过节点之间的日志复制合并不一致的数据,使得不同节点之间最终能够达成一致.这种方式为了提高性能和保证高可用性,牺牲了一致性,虽然适用于一些对一致性要求不高的互联网应用,但无法满足有强一致性要求的业务,例如银行类业务;
- 第 4 种是如 Cassandra[14],Spanner[15]等基于 Quorum[16]的同步方式,通过写读多数节点实现了强一致性模型,而读某一个节点是最终一致性的.这种策略使得强一致性读写延迟较高,性能较差.

综上所述,这几种策略都不同程度地在可用性和一致性之间做出了权衡,根据 CAP 理论[17],在分布式网络分区的情况下,同时保证可用性和一致性是不可能的,因此,实现一种满足高可用性,并且有多种一致性选择的日志复制策略是十分必要的.

## 2 日志复制及选主问题定义

### 2.1 日志同步相关问题

在主备架构下,当主节点出现异常情况时,主节点可能存有未提交的事务日志,而对这些未提交日志的处理有可能导致系统主备之间数据不一致.

如图 1 中 *A*,*B* 分别代表主备节点,在 $t_1$ 时刻,假设节点 *A* 作为主节点提供服务,节点 *B* 作为备节点提供服务,两个节点最大的日志序号 *A.LSN*,*B.LSN* 均为 2.此时,有一个写操作 $Write_1$ 试图更新数据库,节点 *A* 首先产生一条事务日志 *A*.log3 并写到磁盘后,节点 *A* 的最大日志号 *A.LSN* 变为 3.假设在 $t_2$ 时刻,更新的日志 *A*.log3 还没有复制给节点 *B*,主机 *A* 便出现了故障.在 $t_3$ 时刻,节点 *B* 检测到主节点的故障并将自己的身份变为主节点对外提供服务,紧接着来了第 2 个写操作 $Write_2$,由于节点 *B* 并未收到节点 *A* 生成的 *A*.log3,故节点 *B* 的最大日志序号 *B.LSN* 依然为 2,而 $write_2$ 写操作将生成 *B*.log3,*B.LSN* 变为 3.在 $t_4$ 时刻,节点 *A* 通过故障恢复,重启发现已经有了新的主节点,故作为备节点提供服务,此时,主节点 *B* 试图将刚刚 $write_2$ 操作生成的日志 *B*.log3 复制给节点 *A*,节点 *A* 发现自己有 *A*.log3 与 *B*.log3 内容不一致,拒绝了这个日志,由此产生了在 *A*,*B* 两个节点上,拥有相同日志号的日志但内容不一致的问题.这是由于这两个日志(*A*.log3 和 *B*.log3)分别是由 *A*,*B* 在成为主节点期间生成的,而这两个日志对应的事务是否提交也无从得知,因此,这两个日志往往无法得到正确处理,导致系统产生了数据不一致.此时需要人工介入,检查故障之前这些日志对应的事务提交状态,并删除未提交的事务日志,使得系统继续正常运行.

Raft 算法将事务日志标记为两部分:已提交日志和未提交日志.已提交日志是指已经复制给多数节点,并已

经被主节点提交的日志;未提交日志包括不满足提交条件的日志(该日志未复制给多数节点)和满足提交条件(已经复制给了多数节点)但没有收到主节点确认提交信息日志.针对未提交日志,Raft 算法会保留全部的已提交日志和满足提交条件的未提交日志,而未满足提交条件的未提交日志可能会被删除.通过这种方式,Raft 算法保证了副本之间数据的一致性,但是 Raft 缺乏对事务一致性的讨论,读写操作只能通过主节点,而未对在其他节点进行读写事务的一致性作阐述.本文将 Raft 协议进行扩充,使得读操作可以使用所有节点,从而降低主节点的压力,并且提高可扩展性.

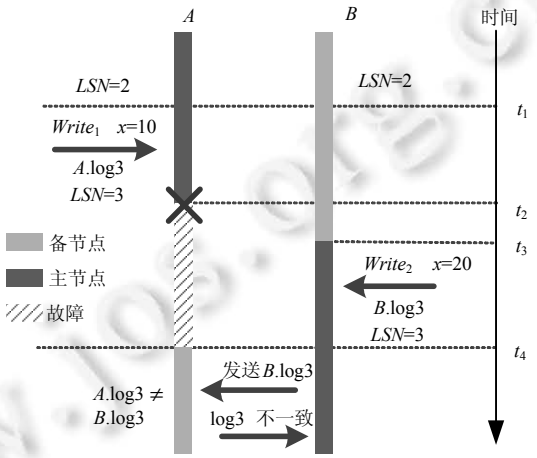


Fig.1 Error examle in master-slave replication

图 1 主备日志同步的异常示例

### 2.2 选主相关问题

活锁问题(livelock)[3,18]出现在 Paxos 算法的投票选举阶段,每个节点同时投票造成的选主连续失败导致选主时间过长,影响了系统可用性.虽然有些 Paxos 的实现算法(例如 Raft)将选主时间点随机化,以此来减少活锁问题发生的概率,但是并没有彻底避免该问题.而且活锁问题随着系统节点的增加会更加严重,制约着系统扩展性和可用性.

网络分区的情况下,可能造成多个主节点同时服务的问题,我们将其称为多主问题,而多主问题直接影响到系统的读写一致性.例如:当系统中的主节点与其他节点发生网络分区,导致该主节点无法与其他节点进行通信,系统会在其他节点中选出新的主节点,此时系统中出现了两个主节点,虽然 Raft 算法中确保两个主节点只有一个能够写成功,但是无法约束从两个主节点的读操作.在这种异常情况下,无论是 Paxos 算法还是 Raft,都无法保证从新旧两个主节点中读取一致的数据.

此外,在网络分区情况下还有可能出现频繁选举的问题.例如图 2 中的 5 个节点,节点 1 与节点 5 之间无法正常通信,但是与其他节点网络正常.假设此时节点 1 是主节点,节点 5 在一段时间内无法收到主节点的消息,便认为集群内没有主节点,因此会发起选主,而节点 2~节点 4 会收到该选主消息并表示同意.节点 1 在同步日志时,发现节点 2~节点 4 已经有了新的主节点,因此放弃了主的身份.但是由于节点 1 也无法收到节点 5 的消息,一段时间后,节点 1 也将发起选主,如此循环下去.尽管 Raft 算法[5]提出了新的主节点并通过写入一条空日志的方法来避免主节点在节点 1 和节点 5 之间频繁更换,但是这种办法只能保证在每次选主时,节点 5 总能成为主节点,而节点 1 一直会处于选举节点,但是仍然无法避免频繁的重新选主.在选主时间段内,系统无法正常服务,因此会降低系统的可用性.

为了解决以上问题,本文采用了一种轻量级的选主方式,依靠一个轻量级的 Root 节点对各个节点进行选主,该 Root 节点仅维护各个节点信息,依靠 Linux-HA[19],Zookeeper 等第三方软件或者硬件等方式维持高可用.这样做将复杂的选主问题交给了 Root 节点,即使在极端的情况下,也不会影响到事务节点的服务.具体内容将在第

4.1 节详细介绍.

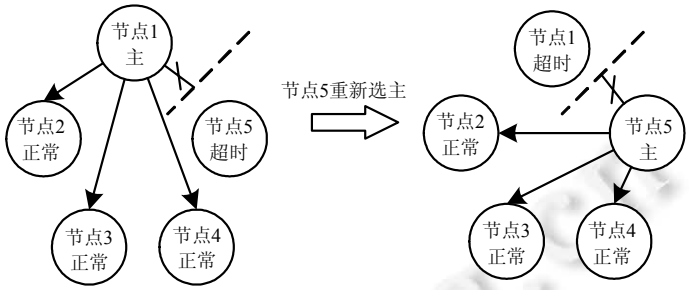


Fig.2 Example of continually election

图 2 频繁选主问题示例

在Root节点选举主事务处理节点时,不需要通过投票选举,直接根据每个节点的日志信息选主即可,从而解决了活锁问题.对于多主问题,本文对选主过程做了两点修改:一是在主节点上添加日志复制租约,当主节点无法正常复制日志(无法成功复制给多数节点)的时间超过了租约时间,主节点将会放弃主,等待重新选主;二是在选主之前,需要先等待旧主节点租约过期.对于频繁选主问题,即使节点之间出现了图 2 所示的问题,此时 Root 节点能够与每个节点正常保持通信,不会发起新的选主,因此避免了频繁选主的问题.

## 3 日志复制的实现

### 3.1 日志文件结构

在内存数据库中,事务数据存储在内存中,为了防止内存掉电丢失数据,需要将事务日志写入到可靠的存储介质(如磁盘)中,这个过程称为日志的持久化.我们在日志项中记录一个事务的修改,日志项由日志内容、日志编号(LSN)和日志版本(term)构成,如图 3 所示.日志文件包含日志项、日志检查点(check_point)、已提交日志点(cmt_seq)和未提交日志点(uncmt_seq).日志检查点是日志恢复的起始点,从日志检查点+1 到已提交日志点表示该节点已经提交的日志,从已提交日志点+1 到未提交日志点是未提交的日志.日志版本是系统主节点成为主那一时刻的时间戳,由于异常情况下主节点可能发生更替,通过日志版本可以比较出该主节点的新旧.已提交日志是大多数节点已经达成一致的日志,最终,所有节点都会接受这个日志,所以这种日志可以直接提交.未提交日志是未确定的日志,表示这部分日志是否已经达成一致是未知的.

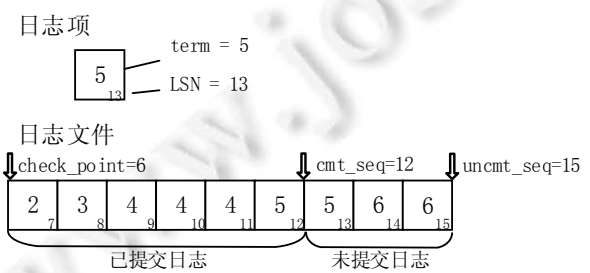


Fig.3 The structure of the redo log file

图 3 重做日志文件结构

### 3.2 日志复制策略

在主节点事务执行完毕、提交该事务之前,需要将该日志异步地发送给所有节点,每个节点收到日志后,将日志写入磁盘,回应主节点.此时,该日志还处于未提交状态.多数节点将日志写磁盘后(包括主节点自己),主节点

提交该日志,同时更新已提交日志点.此外,主节点在每次发送日志时都会携带上主节点已提交日志点,每个节点根据该信息提交自己的日志.如果主节点没有新的日志,则会向每个节点发送空日志(一条特殊的 NOP 日志),用于更新每个节点的已提交日志点.

接下来,我们将以一个具体示例解释上述过程.图 4 展示了客户端的写操作所产生的 3 号日志是如何在节点之间进行同步的.在主节点发送 3 号日志前,未提交日志点 *uncmt_seq* 和已提交日志点 *cmt_seq* 都为 2.主节点将 3 号日志写入磁盘日志文件,更新未提交日志点 *uncmt_seq*=3;然后,主节点将 3 号日志发给每个非主节点,发送时携带当前提交日志号 *cmt_seq*=2 一同发送给非主节点.非主节点收到日志 3 和 *cmt_seq* 后,首先提交日志号小于等于 2 的日志,然后把日志 3 写入磁盘文件,更新未提交日志点 *uncmt_seq*=3,并回应给主节点 $ack_3$,表示已经收到了 3 号日志.若主节点收到了多数节点回应的 $ack_3$,主节点此时提交 3 号日志,更新主节点的已提交日志点 *cmt_seq*=3 并回复客户端 $ret_3$.

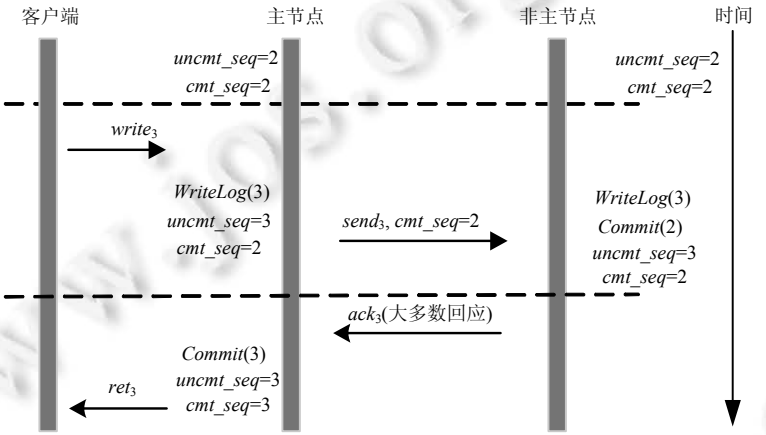


Fig.4 Example of log replication

图 4 日志复制示例

日志复制策略需要考虑到网络通信中存在延迟丢包等不稳定因素,这可能导致主节点无法收到非主节点回应的复制成功的消息,从而不能提交事务,也可能导致非主节点接收到不连续的日志.如果主节点发现发送日志没有收到多数节点的回应,那么非主节点可能收到了该日志也可能没有收到.因此,主节点不能提交相关事务,也不会主动地询问多数节点是否已经持久化了该日志,而是等下一条日志能够提交时,连同本条日志一同提交(如果下一条日志能够收到多数节点的回应,说明这些节点已经持久化了下一条日志之前的所有日志).非主节点会主动查看接收的日志与之前日志之间是否有空缺,如果缺少了之前的日志,则非主节点会向主节点同步请求这些日志,直到前面的日志都补全之后,非主节点才会对主节点进行回应.此时,主节点若收到大多数回应,则主节点会将后面这个日志连同之前的未提交日志一起提交.

图 5 是主节点发送 11 号~20 号日志的例子.主节点发送 11 号、12 号日志是网络处于正常状态,当主节点在发送 13 号~16 号日志期间发生了网络不稳定,导致大部分节点没有收到该日志包,则该日志包中的日志不满足多数回应的条件,因而暂时不能提交.在这之后网络恢复了正常,主机继续发送 17 号~20 号日志,这时候,某些正常收到 13 号~16 号日志的节点能够正常接收日志,而大部分节点会发现自己 13 号~16 号日志缺失,在这种情况下会主动向主节点拉取这部分日志(如图 5 中的*标记所示).等待成功拉取之后,将日志 13 号~20 号一起写入磁盘并回应主机.主节点发现虽然 16 号日志没能收到多数回应,但是 20 号日志收到了多数回应,说明大多数节点已经拉取并补全了缺失的日志,因此主节点顺利提交 20 号之前的日志.如果主节点没有新的事务日志生成,那么主机会生成一条空日志(NOP 日志)发送给非主节点,这条空日志中携带着主机已经提交了 20 号日志的信息,备节点也提交了 20 号之前的日志.这种机制会使备节点最终总有一条日志未提交,而这条日志一定是空日志,

对数据一致性没有影响.

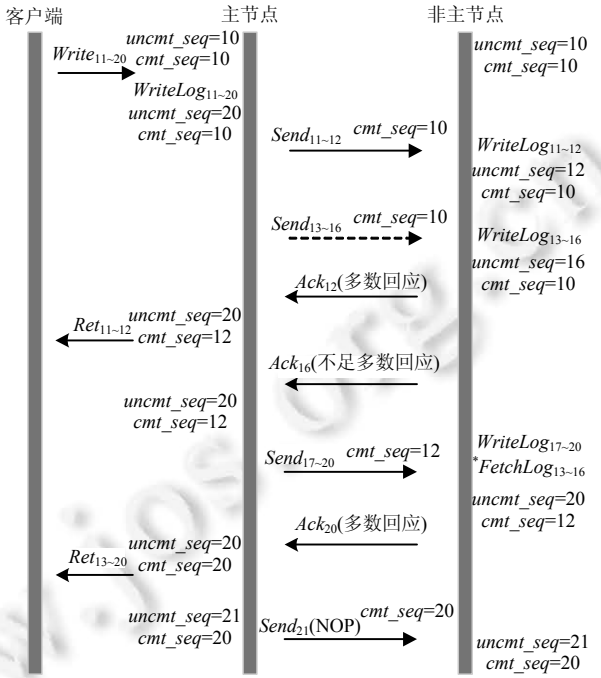


Fig.5 Example of network exception in log replication

图 5 日志复制中的网络异常示例

## 4 故障恢复

一般的机器故障(如磁盘等机器硬件故障)发生时,程序能够检测到,并且程序会自动退出,故障解决后会重启程序.对于内存数据库来说,重启后需要回放日志重新构建内存数据.如果非主节点发生了故障,重启后需要追赶主机的日志.对于主节点故障,首先需要重新选择主节点,之后需要解决不同节点中日志不一致的问题,而这些不一致以主节点日志为准.下面就上述几种情况做详细说明.

### 4.1 选 主

主事务处理节点故障时,其他非主节点可以通过分布式一致性协议进行选主,选出的新主事务处理节点能够继续提供服务.如果在事务处理节点之间使用分布式一致性算法(如 Paxos,Raft 算法)进行选主,选主时间随着事务处理节点数增加而增加,当事节点数量很多时,过长的选主时间会影响系统可用性.而使用一组(例如 3 个或 5 个)轻量级的 Root 节点进行选主则可以很好地避免这个问题.这一组 Root 节点首先进行选择主 Root 节点的工作,主 Root 节点的作用是快速选择主事务处理节点.这一组 Root 节点本身通过 Linux-HA[19],Zookeeper[18]等软件,或者硬件方式维持高可用,在主 Root 节点出现故障时,通过重新选主可以确定新的唯一的主 Root 节点.

Root 节点的选主同样也需要避免多主、频繁选举的问题.Root 节点之间可以通过心跳的方式来维持一定的租约[20],通过租约能够确保任意时刻只有唯一的主 Root 节点.这个结论有两个关键条件:一是当且仅当多数个 Root 节点的租约过期后,选主才能成功;二是当主 Root 节点无法与多数个 Root 节点通信的时间超过了租约时间,主 Root 节点会放弃主身份.对于频繁选举问题,只要在选主之前询问一下其他 Root 节点,在多数节点没有主 Root 节点时才进行选主操作,就可以避免频繁地选主.主 Root 节点选择主事务处理节点的过程首先由主 Root 节点向所有数据库节点获取最后日志号 lsN(last sequence number)和日志版本 term,选择 term 最大的节点成为主节点,当 term 相同时(说明是同一个主节点生成的日志)时,选择 lsN 最大的节点.这样,通过每个节点的信息选

择出哪个节点有资格成为主节点.这个过程与 Raft 算法是一致的,因此能够保证选出的主节点上的日志一定能够被多数节点认可(Raft[5]算法可以证明),选主算法见算法 1.

**算法 1**. 选主算法.

输入参数:总节点数 *total_node_num*;

输入参数:节点列表 *server_list*;

输出参数:主节点编号 *master_idx*.

```
1.  let master_idx=−1;
2.  let max_term=−1, max_lsN=−1;
3.  do
4.    let resp_n=0;   //记录正常通信的节点数量
5.    for ( server_x in server_list) do
6.      发送请求给 server_x,获取 x_term 和 x_lsN;
7.      if 请求成功 then
8.        resp_n++;
9.        if (x_term>max_term||(x_term==max_term && x_lsN>max_lsN)) then
10.         master_idx=server_x.idx;
11.         max_term=x_term, max_lsN=x_lsN;
12.       end if
13.     end if
14.   end for
15. while (master_idx==−1||resp_n≤total_node_num/2);
16. 通知所有节点,选出的主节点的编号为 master_idx
```

选主算法结束后,新主节点上可能有旧主节点的未决日志,需要对其日志进行处理才能够正常服务,这段时间我们将其称为切换时间.因此,从主节点故障到新主节点恢复服务的时间的计算公式为

恢复时间=选主时间+切换时间.

### 4.2 非主节点恢复

非节点故障重启之后,需要回放日志,此时,节点上的日志有已提交日志和未提交日志两种.根据第 3.2 节的日志复制策略,已提交日志一定是一致的,可以直接回放.而未提交日志的处理需要通过主动拉取主节点的日志,按照顺序进行比对:如果未提交日志与主节点一致,则继续向后比较;如果该日志与主节点不一致,说明这些日志是旧主节点不满足提交条件的无效日志,可以直接删除.故将主节点的日志覆盖本地磁盘,并删除后面的日志.在这个恢复的过程中,记录 cfN(confirm number)表示与主节点比对过的最大的日志号,通过这个编号,主节点可以得知其他节点的日志同步状态,进而决定这些未决日志是否可以提交.如果在恢复期间主节点再次故障,则该算法将会重新执行,算法对已提交日志和达成一致的未提交日志一定会被保留.因此,无论算法重复执行多少次,已提交的事务一定可以被恢复,见算法 2.

**算法 2**. 非主节点恢复算法.

```
1.  if 已提交日志未全部回放 then
2.      replay_commit_log()   //回放已提交日志
3.  end if
4.  从日志文件中获取 cmN=已提交的最大日志号
5.  从日志文件中获取 lsN=最大日志号
6.  for (i=cmN+1 to lsN) do
7.      log_i=从主节点取编号为 i 日志
```

```
8.          if log_i 没有在磁盘上 then
9.              将 log_i 写入磁盘
10.         else if log_i 在磁盘上但内容不一致 then
11.             清除磁盘上编号≥i 日志
12              将 log_i 写入本地磁盘
13.             break
14.         else if log_i 在磁盘上且内容一致 then
15.             continue
16.         end if
17.         更新已确认的日志号 cfN=i
18.     end for
```

### 4.3 主节点恢复

如果某个节点被选为了新主节点,则该节点恢复过程与其他节点恢复过程稍有不同.在新主节点上的已提交日志与其他节点上的意义相同,可以直接回放提交.而未提交日志中,一定是旧主节点发来的日志,这些日志可能已经被旧主节点提交了,也可能没有提交,所以为了保险起见,新主机必须等待多数节点确认之后再提交这些日志.因此,新主节点在恢复正常服务之前,先将旧主机的未决事务全部与其他节点进行商榷并提交,保证了旧主已经提交的事务日志不会丢失,具体算法描述见算法 3.

**算法 3**. 主节点恢复算法.

```
1.      if 已提交日志未全部回放 then
2.          replay_commit_log()   //回放已提交日志
3.      end if
4.      从日志文件中获取 cmN=已提交的最大日志号
5.      从日志文件中获取 lsN=最大日志号
6.      for (i=cmN+1 to lsN) do
7.          do
8.              get 所有节点与主机的确认号 cfN
9.          while (多数个节点的 cfN<i)
10.         提交日志号为 i 的日志
11.     end for
```

## 5 一致性分析

表 1 所示的主节点负责事务处理,是系统中唯一的写节点,在主节点上的读写都是强一致性的,但是负载相对较高,吞吐量较低;而非主节点上的日志回放在主节点之后,因此读取非主节点机的数据可能会导致读出旧数据,而最终主节点会将提交信息发给非主节点,最终非主节点会与主节点保持一致,因此读取非主节点上的数据是弱一致性,而非主节点负载相对较低,因此吞吐量可以达到很高.当系统出现异常导致节点数不足多数时,此时没有主节点,所有节点只能提供弱一致性的读服务.由于基于 Paxos 系统中通常会配备 3 个节点或者 5 个节点,所以多数节点故障的几率是比较少见的,即使出现也不会对一致性造成影响.

本文的日志复制实现基于 OceanBase 数据库开源的 0.4.2 版本,该数据库主要包含 4 个部分:RootServer, UpdateServer,MergeServer 和 ChunkServer.其中:RootServer 是集群的管理者;UpdateServer 是内存数据库,维护着系统的增量数据,也是事务的处理中心;ChunkServer 存放基线数据;MergeServer 接受客户端的连接,并将请求分发到 UpdateServer 和 ChunkServer,合并增量数据和基线数据,并提供负载均衡.由于 MergeServer 是专门负责处理 SQL 请求的节点,该节点可以根据不同的应用提供不同的一致性:当需要强一致性读取时,将读请求发给主节

点;当使用弱一致性读时,将读请求发给非主节点.大多数业务对读的一致性要求不高,因此这种机制有很好的负载均衡功能,有助于提高数据库的扩展性,使得主节点不会成为性能瓶颈.

**Table 1** Tradeoff in consistency and throughput

**表 1** 一致性与吞吐量的取舍

| | 服务 | 读主节点 | 读备节点 |
|---|---|---|---|
| 多数节点存活 | 写/读服务 | 强一致性<br>低吞吐量 | 弱一致性<br>高吞吐量 |
| 不足多数节点存活 | 读服务 | –<br>– | 弱一致性<br>高吞吐量 |

## 6 实验评估

本文将日志同步策略和恢复机制实现于 OceanBase 系统中 UpdateServer 内存数据库,实验分为两部分:第 1 部分是在不同的读写比例、网络等情况下,对吞吐量、恢复时间等性能指标的测试分析;第 2 部分是与未使用本文技术的 OceanBase 系统中 UpdateServer 的对比测试.其中,第 1 部分还分为正常情况下的测试和异常情况下的测试.

### 6.1 实验环境

第 1 部分实验基准测试工具 YCSB[21]的 0.7.0 版本进行测试,使用了 OceanBase 的一个集群,集群中有 3 台 UpdateServer,1 台 RootServer,6 台 ChunkServer 和 6 台 MergeServer.其中,3 台 UpdateServer 内存数据库使用了本文提出的日志复制策略.每台机器配置了两颗 Intel(R)Xeon(R)E5606@2.13GHz 型号的 4 核心 CPU 以及 100GB 内存和 100GB 的 SSD 盘.

第 2 部分实验是使用本文提出技术前后的对比测试,在某银行的生产测试环境中进行,使用了基准测试工具 Sysbench 0.5 版本,搭建了原 OceanBase0.4 版本的 1 个集群和本文改造后的 1 个集群.集群中有 3 台 UpdateServer,1 台 RootServer,4 台 ChunkServer 和 4 台 MergeServer.每台机器配置了两颗 Intel(R)Xeon(R)E5-2650 v3@2.30GHz 型号的 10 核心 20 线程的 CPU 以及 100GB 内存和 4T 的机械硬盘.两个集群配置相同,只是其中 UpdateServer 使用的日志复制策略不同,原 OceanBase0.4 版本集群中使用的是主备日志同步策略,本文改造后的集群中使用了本文提出的日志复制策略.

### 6.2 实验方法与结果分析

- 正常情况测试

实验 1:评估不同写负载情况下,强一致性和弱一致性的吞吐量和延迟.其结果如图 6 所示,将写负载所占比例分成 10%~50%这 5 个梯度(剩下的比例为读负载),使用 YCSB 的 200 个线程并发请求.

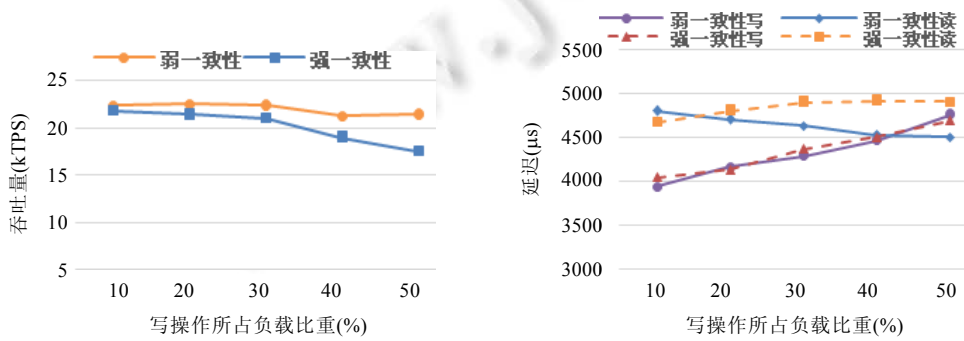


Fig.6 Throughput and latency of two model under different workloads

图 6 两种一致性模在不同写负载比例下的吞吐量和读写延迟

在吞吐量方面,在写比例较少时,强一致性与弱一致性的吞吐量相当;随着写负载比例的增加,强一致性模型的吞吐量会稍有下降,而弱一致性表现比较平稳.在读写延迟方面,写负载的增加将会影响强一致性的读延迟升高,而弱一致性的读延迟随着写比例的上升会下降.这是由于弱一致性读事务分担到了每个节点上,因此性能较好.而在写事务上,由于无论哪个一致性模型都会将写操作作用于主节点,因此写操作的延迟和性能几乎相同.由此可见:选择弱一致性读平衡了读操作的负载,能够提高系统的吞吐量,降低延迟.

实验 2:评估系统的扩展性.对比不同客户端的连接数的情况下,两种一致性的吞吐量,负载使用的是 YCSB 的 workloadb(读 95%写 5%).其结果如图 7 所示:随着客户端连接数量的增加,强一致性的吞吐量逐渐达到瓶颈,而弱一致性吞吐量继续呈增长趋势.这是由于弱一致性读操作将会分担到每个节点上,扩展性相对强一致性读较好.Raft 算法中采用了强一致性的读,而本文采用的是扩展的 Raft,可以使用强弱两种一致性,通过实验看出弱一致性具有较好的扩展性.因此,本文实现的扩展 Raft 与原算法相比有较好的扩展性.

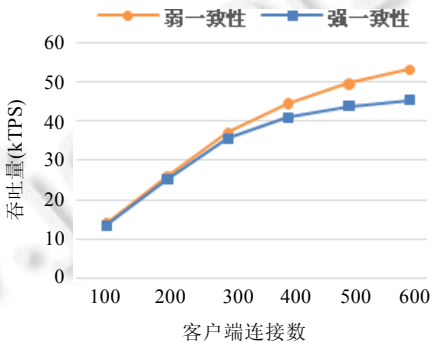


Fig.7 Scalability of two model

图 7 两种一致性模型的扩展性

- 异常情况测试

实验 3:评估系统在不同的负载比例下,将某个非主节点网络中断一段时间之后再恢复,记录该节点的恢复时间.负载使用 6 个梯度,写操作分别所占比例 0~50%,其结果如图 8 所示:写负载越重,网络中断时间越长,恢复时间也越长.这是由于在故障期间,该节点上网络中断,无法收到主节点的日志,而写比例越重的负载,随着时间增加,该节点与主节点日志差距越大,该节点恢复时需要从主节点拉取缺失的日志,因此恢复时间就越长.在写负载为 50%时,网络故障 60s,恢复时间大约为 80s,系统能够较快恢复.

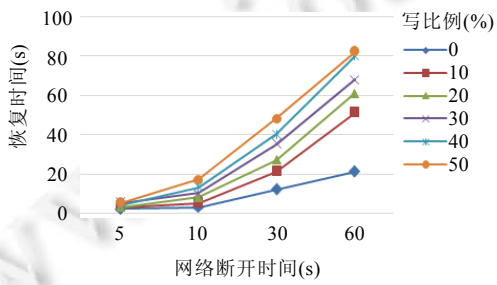


Fig.8 Recovery time in the scenario of network disconnected

图 8 网络故障恢复时间

实验 4:评估主节点故障后,重新选主的效率.在系统服务过程中,将主节点进程杀掉,模拟主节点故障的情况,测试主节点的切换时间和选主时间.其结果如图 9 所示,恢复时间大约在 25ms 左右.其中,主要时间用于切换的开销,约 20ms;选主算法所用时间在 2ms~5ms 之间,节点越少,选主时间越短.与 Raft 算法相比,由于去掉了投

票时间,因此选主用时比 Raft 算法短,而且用时相对稳定.

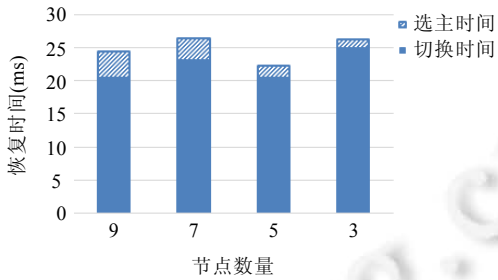


Fig.9 Master recovery time

图 9 主节点恢复时间

实验 5:评估不同的网络延迟和丢包率的情况下,日志复制的效率.通过使用工具人为增加网络延迟和丢包率,在 YCSB 的 workloada 负载(读 50%写 50%)下,测试系统的吞吐量.在网络延迟增加的情况下,日志复制延迟变高,而事务提交需要等待其日志至少复制给一个节点,因此,系统事务处理速度会变慢.而丢包会影响日志复制,有些日志包会丢失需要非主节点主动获取缺失日志,因此也会影响系统的事务提交速度.实验结果如图 10 所示:在网络延迟小于 1ms 和丢包率小于 5%时,对性能影响不大;当网络延迟超过 10ms 或者丢包率超过 10%,对系统有较大影响.因此,本文的日志复制策略能够抵御一定的网络延时和丢包,使得系统吞吐率不会显著下降,提高了系统的可用性.

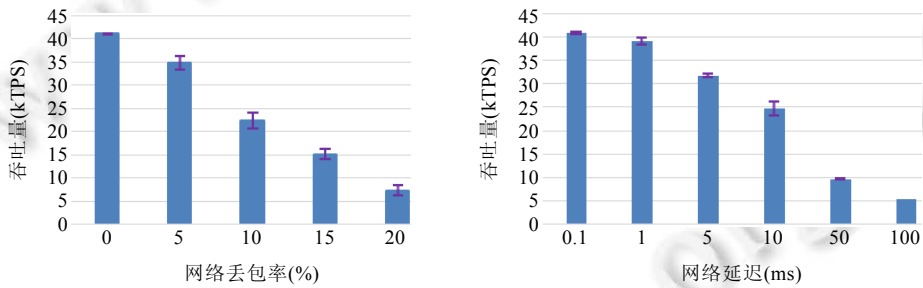


Fig.10 Throughput in the scenario of different network latency and package loss

图 10 不同网络延迟和丢包率的情况下系统吞吐量

实验 6:验证系统在高负载下的可用性.实验使用了 YCSB 的 workloada 负载,如图 11 所示:在系统平稳运行一段时间(0s~50s)后,在某一时刻(50s)杀死主事务处理进程,系统的吞吐量迅速降低为 0,并在随后进行了租约过期检测、重新选主和切换这 3 个步骤;4s 之后(54s),新的主事务处理节点恢复完成,系统重新开始正常运行.由此证明了在较高负载下,本文的恢复机制能够检测到异常并快速恢复,有较高的可用性.

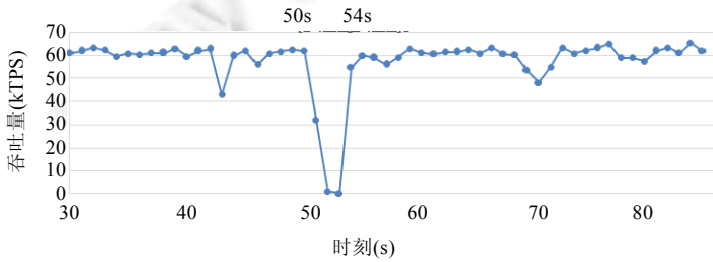


Fig.11 Impact of master transaction processing node failure under high workload

图 11 高负载下主事务处理节点异常对系统的影响

- 对比测试

实验 7:将本文的日志复制策略与原主备日志同步策略的性能进行对比,如图 12 所示.从图中可以看出:在高读负载下,对于相同的读一致性策略,两种日志同步策略性能几乎相同.这是因为日志复制策略对读事务没有影响;而在高写负载下,本文提出的日志复制策略相比原来的同步策略性能有所下降,大约下降了 16%.这是由于相比之下,原主备同步只需要将日志写入本地的主节点,事务便可以提交;而为了提供了可靠的异常恢复机制,本文的日志复制策略需要确保多数节点将日志写入磁盘后,事务才能提交,因此事务执行的时间变长,吞吐量受到了影响.但是这样做带来的好处是保证了提交事务日志不会丢失,提高了系统的可用性和一致性,牺牲了一小部分性能是值得的.

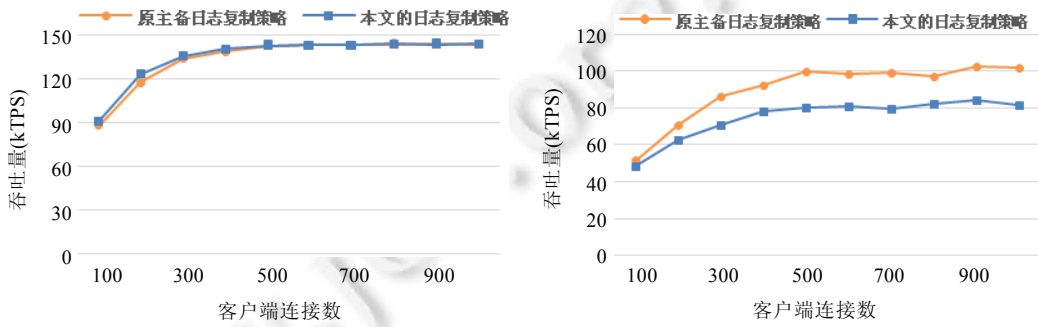


Fig.12 Comparison between our log replication strategy and master-slave strategy

图 12 本文日志复制策略与原主备日志复制策略的性能对比

## 7 结 论

本文提出了一种集群环境下分布式数据库中的日志复制和恢复机制,能够在故障情况下保证事务日志的完整性和一致性,实现了强弱两种一致性读策略,提高了系统的扩展性.此外,本文提出了一种轻量级的选主算法,能够快速稳定地选主,避免了经典一致性算法选主算法中活锁、双主和频繁选主等问题,提高了系统的可用性.最终,通过实验分析各种异常情况下对系统性能的影响,证明了日志复制和恢复的高效率.

本文设计的日志复制和恢复机制已经应用于某银行数据库系统中,日志复制和恢复算法具有很好的通用性,因此也容易应用于其他类似的分布式事务系统.本文只实现了在非主节点弱一致性读的策略,而如何在非主节点实现强一致性读,是今后研究的重点.

**References**:

[1] Özsu MT, Valduriez P. Principles of Distributed Database Systems. 2nd ed., Berlin: Springer-Verlag, 1999.

[2] Yang ZK. The architecture of OceanBase relational database system. Journal of East China Normal University (Natural Science), 2014,9(5):141−148,163 (in Chinese with English abstract).

[3] Lamport L. Paxos made simple. ACM SIGACT News, 2001,32(4):18−25.

[4] Burrows M. The Chubby lock service for loosely-coupled distributed systems. In: Proc. of the Symp. on Operating Systems Design and Implementation. USENIX Association, 2006. 335−350.

[5] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. Stanford University, 2013.

[6] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems, 2008,26(2):205−218.

[7] Yang CH. Large Scale Distributed Storage System. Beijing: China Machine Press, 2013 (in Chinese).

[8] Rao J, Shekita EJ, Tata S. Using Paxos to build a scalable, consistent, and highly available datastore. Computer Science, 2011,4(4): 243−254.

[9] Thalmann L, Ronstrom M. MySQL Cluster Architecture Overview. 2004.

[10] Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Mruthy R. Hive□□A warehousing solution over a map-reduce framework. Proc. of the VLDB Endowment, 2011,2(2):1626−1629.

[11] Chandra T, Griesemer R, Redstone J. Paxos made live: An engineering perspective. In: Proc. of the 26th ACM Symp. on Principles of Distributed Computing (PODC 2007). 2007. 398−407. [doi: 10.1145/1281100.1281103]

[12] Decandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. ACM Sigops Operating Systems Review, 2007,41(6):205−220.

[13] Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen HA, Puz N, Weaver D, Yerneni R, PNUTS: Yahoo!'s hosted data serving platform. Proc. of the VLDB Endowment, 2015,1(2):1277−1288.

[14] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. ACM Sigops Operating Systems Review, 2010, 44(2):35−40.

[15] Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li HY, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D. Spanner: Google's globally-distributed database. ACM Trans. on Computer Systems, 2013, 31(3):251−264.

[16] Skeen D. A quorum-based commit protocol. In: Proc. of the Berkeley Workshop on Distributed Data Management & Computer Networks. 1982. 69−80.

[17] Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM Trans. on ACM Sigact News, 2002,33(2):51−59.

[18] Hunt P, Konar M, Junqueira FP, Reed B. ZooKeeper: Wait-Free Coordination for Internet-scale Systems. 2010. 653−710.

[19] Robertson A, Robertson A. The Evolution of the Linux-HA Project. 2004.

[20] Gray C, Cheriton D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. ACM Sigops Operating Systems Review, 1989,23(5):202−210. [doi: 10.1145/74851.74870]

[21] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the ACM Symp. on Cloud Computing. ACM Press, 2010. 143−154. [doi: 10.1145/1807128.1807152]

## 附中文参考文献:

[2] 阳振坤.OceanBase 关系数据库架构.华东师范大学学报(自然科学版),2014,9(5):141−148,163.

[7] 杨传辉.大规模分布式存储系统.北京:机械工业出版社,2013.

**王嘉豪**(1993－),男,硕士生,主要研究领域为海量数据管理与分析,分布式数据库.

**蔡鹏**(1978－),男,博士,副教授,主要研究领域为可扩展高性能事务处理.

**钱卫宁**(1976－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为互联网环境下的数据管理,大数据管理系统评测基准,社交媒体数据分析,知识图谱构建与应用.

**周傲英**(1965－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为 Web 数据管理,数据密集型计算,内存集群计算,分布事务处理,大数据基准测试和性能优化.