软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006437]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Elsa：一种面向跨区域架构的无协调分布式键值存储系统*

崔玉龙[1], 付 国[1], 张岩峰[1,2], 于 戈[1]

[1](东北大学 计算机科学与工程学院,辽宁 沈阳 110169)

[2](医学影像智能计算教育部重点实验室,辽宁 沈阳 110819)

通讯作者: 张岩峰, E-mail: zhangyf@mail.neu.edu.cn

摘 要: 作为具备高性能和高可伸缩性的分布式存储解决方案,键值存储系统近年来被广泛采用,例如 Redis、MongoDB、Cassandra 等.分布式存储系统中广泛使用的多副本机制一方面提高了系统吞吐量和可靠性,但同时也增加了系统协调和副本一致性的额外开销.对于跨域分布式系统来说,远距离的副本协调开销甚至可能成为系统的性能瓶颈,降低系统的可用性和吞吐量.本文提出的分布式键值存储系统 Elsa,是一种面向跨区域架构的无协调键值存储系统.Elsa 在保证高性能和高可拓展性的基础上,采用无冲突备份数据结构（CRDT）技术来无协调的保证副本间的强最终一致性,降低了系统节点间的协调开销.本文在阿里云上构建了跨 4 数据中心 8 节点的跨区域分布式环境,进行了大规模分布式性能对比实验,实验结果表明：在跨域的分布式环境下,对于高并发争用的负载,Elsa 系统的性能具备明显的优势,最高达到 MongoDB 集群的 7.37 倍,Cassandra 集群的 1.62 倍.

关键词: 跨区域架构;键值存储系统;无冲突备份数据结构;副本一致性;强最终一致性

中图法分类号: TP311

Elsa：a Coordination-free distributed KVS for the cross-region architecture

CUI Yu-Long[1], FU Guo[1], ZHANG Yan-Feng[1,2], YU Ge[1]

[1](School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China)

[2](Key Laboratory of Intelligent Computing in Medical Image,Ministry of Education, Shenyang 110819, China)

Abstract: As a distributed storage solution with high performance and high scalability, key-value storage systems have been widely adopted in recent years, such as Redis, MongoDB, Cassandra, etc. On the one hand,the multi-replication mechanism widely used in distributed storage system improves system throughput and reliability, but also increases the extra overhead of system coordination and replication consistency. For the cross-region distributed system, the long-distance replication coordination overhead may even become the performance bottleneck of the system, reducing system availability and throughput. The distributed key-value storage system called Elsa, which proposed in the article is a coordination-free multi-master key-value storage system that designed for cross-region architecture. On the basis of ensuring high performance and high scalability, Elsa adopts the conflict-free replicated data types (CRDT) technology to ensure strong eventual consistency between replications without coordination,reducing the coordination overhead between system nodes. In this paper, we set up a cross-region distributed environment spanning 4 datacenters and 8 nodes on aliyun platform and make a large-scale distributed performance comparison experiment.The experimental results show that under the cross-region distributed environment, the throughput of Elsa has obvious advantages for high concurrent contention loads, reaching up to 7.37 times of the MongoDB cluster and 1.62 times of the Cassandra cluster.

Key words: cross-region architecture; key value store; CRDT; replication consistency; strongly eventual consistency

键值对存储(key value store, KVS)[1]是一种非关系型的 NoSQL 数据库,用于存储键值对而非关系数据库中的结构化数据.其中的键和值可以是由基础数据类型(如数字、字符串等)组成的复杂数据类型(如列表、集合等).

* 基金项目: 国家自然科学基金(62072082,61672141);CCF-华为数据库创新研究计划(CCF-HuaweiDBIR2020009B);辽宁省重点研发计划(2020JH2/10100037)

收稿时间: 2021-04-02; 修改时间: 2021-06-23, 2021-07-29; 采用时间: 2021-08-23; jos 在线出版时间: 2021-10-20

由于没有关系型数据库中表的概念,键值数据库中键值对的形式十分灵活,不再受到表结构的限制.所以,理论上来说键值数据库是高度可分区的并且允许任意规模的水平拓展,这是键值数据库的优势所在.除此之外,其还具有高性能、操作简单和高可靠性等特点.

随着现代社会数据量的快速增长,单机数据库不再能够满足业务需求,分布式数据库随之出现.分布式系统解决了单机模式下数据存储和系统性能不足的缺点,理论上只要增加物理机器,整个系统的性能就能随之提升.但是,CAP 定理[2-4]却表明,分布式系统在满足分区容错(P)的前提下,不能同时满足强一致性(C)和高可用性(A).CAP 定理指明了分布式系统的设计原则,所以在设计分布式系统时,一般会根据应用场景和特定的业务需求选择优先满足 CAP 三个特性中的两个.

以 MongoDB 为代表的 CP 类系统更加看重副本的一致性,所以一般采用主从或是主备的架构,用户的请求全部由主节点进行处理,并且需要将处理结果同步给集群中的所有节点之后才能够返回给用户处理结果,即副本间满足强一致性.保证强一致性带来的代价则是系统的吞吐量降低和延迟的增加,尤其是在集群节点数较多或是节点跨域的情况下更加明显.而以 Dynamo[5]为代表的 AP 类系统则正好相反,系统的可用性被优先考虑并且只保证副本间的弱一致性.这种架构下,集群间的节点是对等的关系,即任意节点都可以接受并处理用户发来的请求,只需要定期同步数据即可.

然而,在集群节点跨域分布的情况下,网络故障和数据同步的协调开销变得十分明显,甚至可能会抵消掉多副本机制带来的性能提升,这不符合 AP 类系统的设计初衷.这个问题引起了我们的注意,这也是 Elsa 的设计初衷之一.另一方面,为了优先满足系统的吞吐量和可拓展性,AP 类分布式键值存储系统一般不支持事务或仅支持事务的部分特性,如 Dynamo 不支持事务、Redis 采用批处理来替代事务机制、Cassandra 仅支持读已提交的弱隔离级别.然而,这对于需要使用到事务机制来保证数据一致性和完整性的业务逻辑来说是十分不便的.

综合考虑系统性能和对事务特性的支持,我们提出了一种支持多种事务隔离级别的无协调一致性模型.这样可以在保证系统性能和数据一致性的前提下,很好的支持那些对事务隔离级别要求没那么高的应用,这也是 Elsa 的另一个设计目标.

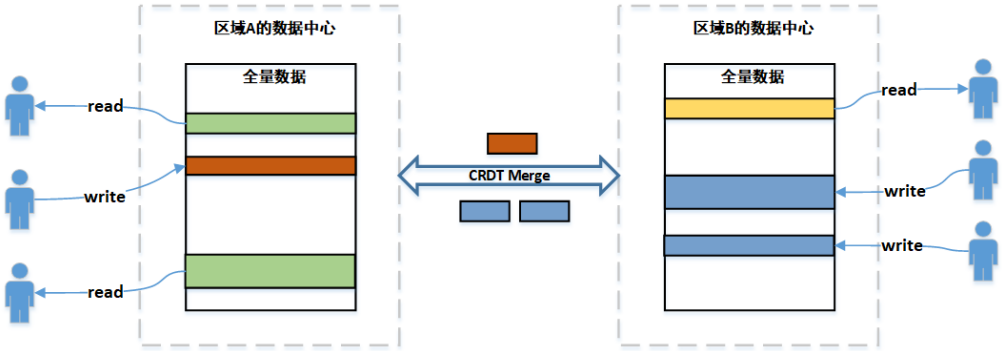


图 1 系统简要架构图

为了实现上述的两个设计目标,我们总结概括了两个主要的设计需求.首先,为了确保系统的高性能和高可用性,我们决定采取域间数据全复制的多副本多活架构,如图 1 所示.集群中的每个节点(域)都拥有全量数据的一份副本,节点间(域间)可以并行的处理各自负责区域内的用户发来的读写事务请求(多点写入),并通过异步地广播消息来保证副本间的强最终一致性,尽可能降低因一致性问题而带来的协调开销.这种跨地域全复制的分布式数据库架构,近些年来被广泛关注和研究,类似的系统有 Aria[6],PSI[7]等.

其次,在不违背第一个设计目标的基础上,综合考虑系统的性能与开销,我们采用轻量级无协调的并发控制算法,而不是其他的基于锁机制的算法或共识协议,来提供一种支持多种事务隔离级别的事务模型.

本文的主要工作包含以下 3 点.1) 面向跨域架构下的弱一致性分布式系统,文章提出了一种基于 CRDT 的

事务一致性模型,提供域内的多种隔离级别的事务保证以及域间副本的强最终一致性.2) 基于上述的事务一致性模型,编码实现了原型系统 Elsa.3) 在大规模的跨域环境下,做了大量的性能对比实验,验证模型的正确性和系统的有效性.

文章其余部分的内容如下.第 1 节介绍了相关的背景知识和国内外相关工作,第 2 节介绍支持多种隔离级别的无协调一致性模型,这是本文工作的核心部分.第 3 节介绍了系统的总体结构,第 4 节介绍了系统的具体实现细节.最后在第 5 节对实验结果做了分析和阐述,第 6 节则是对本文的总结与对未来工作的展望.

1 相关工作

1.1 无冲突备份数据结构

冲突是一个在分布式系统中经常出现的问题,如并发冲突、副本一致性冲突等.并发冲突是指多个线程对同一计算机资源如内存等的竞争所导致的问题,事务型数据库中经常出现的并发写写冲突实际上就是对同一块数据的写竞争.而对于分布式系统中经常采用的多副本机制,一方面多副本机制提升了整个系统的吞吐量和安全性,但同时也造成了副本数据的不一致问题.在没有主次副本的情况下,即各个副本间是对等的关系时,谁的数据才是正确的是一个关键的问题.冲突的产生促使了协调机制的提出与发展,协调机制的作用是消解冲突,促使系统按照一定的规则平稳运行.关系型数据库中常见的并发写写冲突,锁机制是其中比较经典的协调机制.而对于分布式副本一致性冲突问题,基于共识的 Paxos[8-9]或 Raft[10]协议则被广泛采用.协调机制解决了系统中存在的冲突问题,但同时也增加了相应的协调开销.

无冲突备份数据结构(conflict-free replicated data types,CRDT)[11]是各种基础数据结构最终一致算法的理论总结,2011 年由 Marc Shapiro 等人提出,其能根据自定义的规则进行合并,无协调的解决副本间的冲突,并达到强最终一致的效果.“强最终一致性”作为一种分布式弱一致性级别,在论文[12]中被首次提出.文章指出,当且仅当多个副本在交换更新后能够同时达到一致状态即满足强收敛性(Strong Convergence),则称副本间满足强最终一致性.而“强最终一致性”与同为弱一致性级别的“最终一致性”的区别在于,前者要求在副本数据交换后则达到一致性状态,而后者则对副本达到一致性状态的时间点没有保证.

如图 2 是 CRDT 的逻辑结构图,一个 CRDT 结构的实例称为一个 Object.Object 内部存储副本数据,不能直接被用户或其他 Object 直接访问,而是通过对外提供的 Query、Update、Merge 三个接口进行交互.Query 接口用于查询副本数据值,Update 操作允许用户修改副本数据,Merge 操作用于接受远端 CRDT Object 发来的数据并进行同步合并.CRDT 是一种高度可自定义的数据结构,开发者通过设计以上三个接口的业务逻辑和副本数据格式,可以自定义不同的 CRDT 结构,从而实现特定的一致性模型.

一个数据结构是否符合 CRDT 的条件是 Update 和 Merge 操作需要满足可结合性(associativity)、可交换性(commutativity)和幂等性(idempotence),即 ACI 特性.如果 Update 操作本身满足以上 ACI 三律,那么 Merge 操作只需要对远端 CRDT Object 的 update 操作进行本地重放即可,这种形式称为 op-based CRDT 即基于操作的 CRDT,一个典型的例子是跨数据中心的购物车模型.

如图 3 所示,A 和 B 分别是位于两个数据中心的购物车模型,用于记录用户的购物数据,多个用户可以向购物车添加商品.在某段时间内,多个用户分别向 A 中的购物车添加“milk”商品,向 B 中的购物车添加“bread”商品.此时,两个购物车中的商品数据是不同的.由于集合的添加元素操作是满足 ACI 三条特性的,所以在进行副本同步时,A 和 B 只需将本地的添加商品操作发送给对方并进行回放即可,如图中①和②所示.最终,A 和 B 中的购物车数据达到了全局一致.

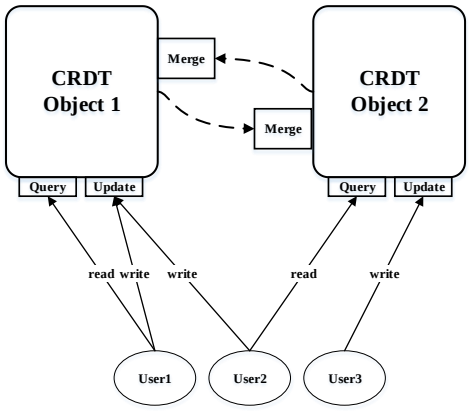


图 2 CRDT 逻辑结构图

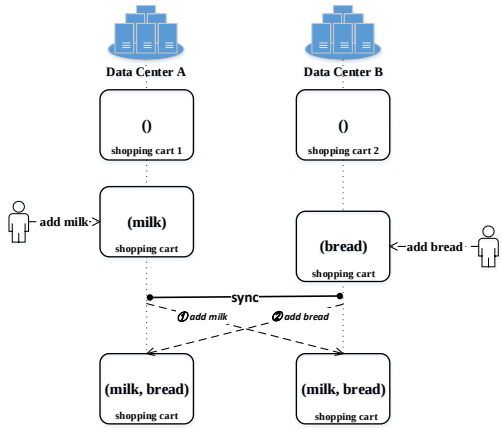


图 3 基于操作的 CRDT 例子：购物车

而对于 Update 操作不能满足 ACI 三条特性的情况,需要为 CRDT 结构附加额外的元信息来辅助 Update 和 Merge 操作具备 ACI 三律.这种形式称为 state-based CRDT,也被称为具备格性质的 CRDT 格结构.典型的例子是遵循“因果一致性”的留言板机制.如图 4 所示,Alice 和 Bob 分别在留言板给对方留言.为了保证留言的因果性,对副本附加了向量时钟的额外元信息.当进行同步时,和基于操作的 CRDT 不同,需要将附加向量时钟的本地数据发给对方进行合并而非直接发送操作.对于满足因果一致性的合并过程来说,向量时钟较大的数据覆盖较小的即可,如图中①所示.而对于向量时钟不能比较的情况,则需要针对具体的业务逻辑设计合并逻辑,如图中②和③所示.所以,对于 state-based CRDT 来说,元信息和合并逻辑的设计是至关重要的.

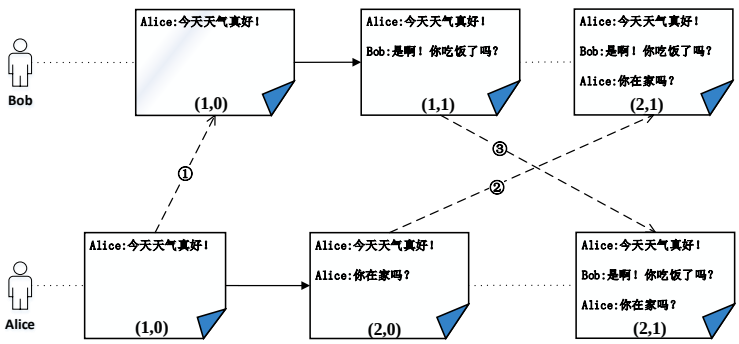


图 4 基于状态的 CRDT 例子：留言板

需要说明的是,虽然 CRDT 结构具备无协调、无冲突的分布式特性,但是它并非适用于所有的分布式事务场景.例如,银行转账、金融股权交易等需要保证分布式强一致性的业务场景,仅支持强最终一致性的 CRDT 结构无法满足.

1.2 无协调一致性技术的最新进展

随着云计算技术的提出与发展,分布式系统的应用变得越来越广泛.由于无协调一致性技术能够有效的消除或减少分布式节点间的协调开销,从而提高系统的性能和可拓展性,近几年成为国内外学者的研究趋势和热点.

加州大学伯克利分校的 RISE 实验室在无协调一致性研究方向先后发表了多篇论文和重要成果.1) 发布了一套名为 Bloom[13]的分布式编程语言,帮助编程人员快速开发无协调的分布式程序.2) 提出了基于单调逻辑的 CALM[14]理论,并证明当且仅当分布式程序满足单调性时,程序才具有一致性和无协调的实现. 3) 提出了

Blazes [15]分析框架,用于分析流计算程序是否符合单调性从而可以无协调的执行.4) 提出了分布式键值存储系统 Anna[16],其设计目标是具备高可扩展性和高性能,适用于从共享内存到分布式消息传递的多种系统架构,并采用了一种基于格结构的分布式数据结构来保证分布式副本的最终一致性.

基于 CRDT 技术的相关研究也在国内外广泛的开展.主要包含：1) 对于复杂数据类型,如集合(set)、列表(list)、树(tree)等 CRDT 结构的实现和改进.例如,论文[17]提出了一种新的面向集合的 CRDT 类型 CLSet, 通过仅附加一个自然数类型的元信息来保证集合数据的一致性.论文[18]提出了一种支持元素移动语义的面向列表类型的 CRDT 结构.论文[19]将基于状态的 CRDT 技术与 Merkle 树结合起来,提出了一种名为默克尔搜索树(MST)的数据结构,并将其应用于分布式存储中.2) 面向 CRDT 的描述和验证框架[20].文章旨在采用模型检验技术验证一系列 CRDT 协议的正确性,构建了一个可复用的 CRDT 协议描述与验证框架.

2 支持多种隔离级别的无协调一致性模型

数据库事务是关系型数据库的重要概念,是由多条命令组成的最小程序执行单元.在传统的关系型数据库中,事务需要满足 ACID 四个特性,是数据恢复和并发控制的基本单位.隔离性是对并发事务的并发程度的定义,隔离性越强并发事务的并发度越低,出现的并发异常问题也会更少.

在 ANSI SQL 标准下定义了一系列的事务隔离级别,包括读已提交、可重复读、快照隔离和可序列化等.由于可序列化隔离级别要求并发事务以近似串行的方式进行执行,所以一般用于对吞吐量要求不高但对一致性要求较高的业务中,如银行转账等操作.而对于大多数的业务逻辑来说,前三种隔离级别已经完全足够.为了叙述更加清楚,在本文其余部分,我们将前三种隔离级别统称为弱隔离级别.

而随着数据量的指数级增长和对于业务数据容灾备份的需求越来越大,跨地区、跨国家等跨域的分布式系统近些年来得到了广泛应用.在分布式环境下,一般需要同时考虑域内事务的并发控制和域间副本的一致性问题,例如采用多版本并发控制(MVCC) + Paxos 协议.然而,对于一些面向跨域架构的弱一致性应用,如 Twitter、Facebook 等面向全球用户的社交平台,MVCC + Paxos 的解决方案显然不太适合,这会使得系统变得更加复杂并且显著地增加了域间的协调开销.而 CRDT 技术做为一种分布式数据结构,能够无协调的解决副本间的冲突,使得所有副本上的数据满足强最终一致性.同时,CRDT 结构中包含的时间戳等元信息,也能够用来做事务并发控制.因此,结合论文[21]中提到的基于缓存的并发读写控制机制和 CRDT 技术,我们提出了一种支持多种隔离级别的无协调一致性模型,从而在保证系统性能和可拓展性的基础上,无协调地实现事务并发控制和解决副本一致性问题.

下面,我们将按照在弱隔离级别定义下,需要解决的异常并发问题和副本一致性问题的先后顺序来阐述相应的解决方案.需要注意的是,由于 Elsa 是一种存储形式为松散的键值对的数据存储,而非传统的关系表,所以我们没有考虑读偏斜异常问题.为了描述的更加清晰和直观,我们将涉及到的操作用符号进行定义.***BEGIN_TRANSACTION*** 用来显式的声明一个事务的开始,***COMMIT*** 和 ***ABORT*** 分别表示事务的提交和废弃.***READ(x)***表示对数据项 x 的读操作,***WRITE(y = 100)***代表将数据项 y 写为 100 的写操作.

2.1 脏读异常的解决方案

脏读是一种读写异常问题,是指对于两个并发事务 A 和 B,A 事务读到了 B 事务还未提交的写操作结果.在 Elsa 中,我们通过将事务还未提交的写操作临时存储在一块称为写缓存的缓冲区中,直至接收到事务发来的 **COMMIT** 命令后才将缓存的所有写操作打包发送给数据库进行原子性提交,从而避免了脏读的问题.所以,当系统启用写缓存机制时,Elsa 系统支持事务的**读已提交**隔离级别.

算法 1 描述了写缓存算法的核心流程,**writeBuffer** 是一块以二级哈希表形式组织的全局可动态拓展的缓冲区.这块缓冲区位于代理端节点,当接收到用户发来的请求指令后,首先获取该请求所属的事务 tid 并根据请求的类型进行判断.如果是 **BEGIN_TRANSACTION** 操作,则为该事务分配一块内存区域(通过 tid 进行索引)用于缓存之后的写请求（3-4 行）.如果是 **WRITE** 操作,则将写请求要写入的数据项 key 和 value 值等元信息缓存在对应的内存区域（6-7 行）.如果是 **COMMIT** 请求,则将缓存中的所有写请求打包发给服务器节点执行,

最后将该事务对应的内存区域回收（10-14 行）.如果是 **ABORT** 请求,则只需要回收对应的内存区域即可（8-9 行）.

算法 1．解决脏读异常的写缓存算法

输入：用户发来的请求 *request*,写缓存 *writeBuffer*

1. *tid* ← *request*.tid
2. *type* ← *request*.type
3. if *type* = ***BEGIN_TRANSACTION*** then
4. *startTs* ← now() // 获取当前时间戳并赋值给 startTs 变量
5. *writeBuffer*[*tid*] ← new HashMap() // 为当前事务在写缓存中申请内存空间
6. else if *type* = ***WRITE***(*key* = *value*) then
7. *writeBuffer*[*tid*][*key*] ← *value* // 缓存并记录下该写请求
8. else if *type* = ***ABORT*** then
9. clear(*writeBuffer*[*tid*]) // 释放申请的写缓存内存空间
10. else if *type* = ***COMMIT*** then
11. *commitTs* ← now() // 获取当前时间戳并赋值给 commitTs 变量
12. commit(*tid*, *startTs*, *commitTs*) // 向服务端提交事务, 具体逻辑见算法 3
13. clear(*writeBuffer*[*tid*]) // 提交结束,释放申请的内存空间
14. endif

2.2 不可重复读异常的解决方案

不可重复读是指对于两个并发事务 A 和 B,事务 A 的先后两次读操作的结果由于事务 B 的写操作而导致的不一致问题.在快照隔离的定义下,事务的读操作可以读取当前时间戳之前的历史数据即快照.利用快照技术为每个事务保留相同的快照,这样对于同一事务中对相同数据项的读操作结果即是一致的.在 Elsa 中,我们通过维护一个和写缓存 **writeBuffer** 类似的读缓存 **readBuffer** 来解决不可重复读的问题.所以,当系统同时开启读缓存和写缓存机制时,Elsa 系统支持**可重复读**的隔离级别.

如算法 2 所示,当代理端接收到用户发来的 **BEGIN_TRANSACTION** 请求,为该事务分配一块由事务 tid 索引的缓存区域用于缓存读操作结果即读操作的快照（3-4 行）.当接收到 ***READ*** 请求后,会首先在写缓存 **writeBuffer** 中进行搜索是否存在隶属于当前事务的对同一个 key 的写操作,如果存在则将写操作的值返回给用户（6-7 行）.否则,会在 **readBuffer** 中进行搜索是否存在该读操作的快照,如果有直接将结果返回给用户（8-9 行）.如果缓存中不存在数据,则需要将该读请求转发给服务器节点进行查询,接收到服务器的查询结果后将其缓存在 **readBuffer** 中,之后将结果返回给用户（10-14 行）.当接收到用户发来的 ***ABORT*** 和 ***COMMIT*** 请求时,这标志着该事务的结束,此时只需要回收 **readBuffer** 中对应的缓存区域即可.

一方面 **readBuffer** 通过缓存读操作的快照避免了不可重复读的问题,另一方面直接读取代理端缓存的数据也减少了一次服务端查询的过程,提高了系统的吞吐量.需要注意的是,对于读已提交和可重复读隔离级别,在满足其基本定义的基础上,还需要考虑读取数据的时效性问题.受到乐观并发控制(OCC)技术的启发,在事务提交时可以增加一步验证本地读集的过程(如算法 2 中 18-22 行所示).如果读集中已读取数据的版本(版本信息可由 CRDT 结构中的时间戳元信息充当)与当前数据库存储中的最新数据版本不同,则证明存在另一个并发事务在本事务执行期间修改了相应的数据,即存在读写冲突.对于没有通过读集验证阶段的事务,可以直接废弃或重新执行,反之则直接提交即可,这样能够保证用户不会读到旧数据.

算法 2. 解决不可重复读异常的读缓存算法

输入：用户发来的请求 *request*,写缓存 *writeBuffer*,读缓存 *readBuffer*

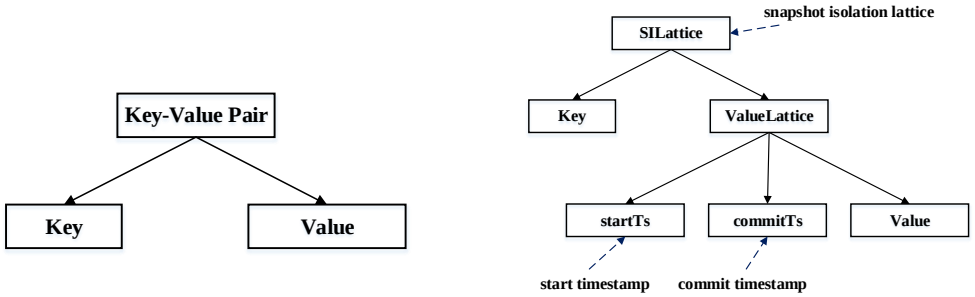
```
1.  tid ← request.tid
2.  type ← request.type
3.  if type = BEGIN_TRANSACTION then
4.      readBuffer[tid] ← new HashMap()   // 为当前事务在读缓存中申请内存空间
5.  else if type = READ(key) then
6.      if key in writeBuffer[tid] then   // 要读取的 key 在写缓存中,直接返回修改后的值即可
7.          sendToUser(writeBuffer[tid][key])
8.      else if key in readBuffer[tid] then   // 要读取的 key 在读缓存中,直接将结果返回给用户
9.          sendToUser(readBuffer[tid][key])
10.     else then
11.         value ← getDataFromServer(key)   // key 不在任何缓存中,发送读请求到服务端
12.         readBuffer[tid][key] ← value   // 将读取到的值写入读缓存并返回给用户
13.         sendToUser(value)
14.     endif
15. else if type = ABORT or COMMIT then
16.     clear(readBuffer[tid])   // 释放申请的读缓存内存空间
17. endif
18. function ValidateReadSet(tid, readSet) //  读集验证过程,保证读取数据的时效性
19.     for record in readSet then
20.         if record.version != DB[record.key].version then   //本地读集中数据版本与数据存储中的不同
21.             return ABORT   //验证失败
22.         endif
23.     endfor
24. return SUCCESS    //验证成功
```

2.3 丢失更新异常的解决方案

在论文[22]中首次提出了快照隔离概念,并提出了一种并发的写写冲突异常问题,称之为丢失更新异常.丢失更新异常可以通过事务对应的时间戳区间来进行判定.时间戳区间的起始时间点是事务的第一次读操作之前的任意时间,记做 **startTs**.结束时间点是事务提交时的当前系统最大的时间戳,记为 **commitTs**.当两个事务的时间戳区间存在交集并且写操作涉及的数据项存在交集时,证明两个并发事务存在丢失更新异常问题.

传统的快照隔离实现对于存在的丢失更新异常是通过施加悲观锁来解决的,即在提交之前锁住相应的行数据,然后遍历自己的写操作集合,检查是否存在某条行数据的最近一次提交时间戳落在了自己的**[startTs, commitTs]**时间戳区间内,如果存在则需要废弃本事务,否则提交本事务并释放悲观锁.通过施加悲观锁的方法,避免了“丢失更新”冲突问题的发生,但同时由悲观锁机制所带来的加锁、释放锁等协调开销和死锁等问题却也降低了系统的执行效率.

为了优先满足系统的高性能和高可用性的设计目标,我们结合章节 1.1 中所介绍的 state-based CRDT（格结构）技术,提出了一种无协调地解决丢失更新异常的解决方案.



a) b)

图 5 支持快照隔离的 CRDT 格结构

首先,我们对服务端的数据存储格式做了修改,以一种格结构的形式进行存储.如图 5 所示,传统的键值存储系统的数据以图 5 a)中的键值对形式进行组织,不包含任何额外的元信息.而如前面介绍的一样,并发事务是否存在丢失异常是通过事务的开始时间戳 **startTs** 和提交时间戳 **commitTs** 来进行判定的,所以我们将这两项时间戳元信息直接添加到了数据存储中,形成了如图 5 b)所示的支持快照隔离的 CRDT 格结构.那么,当事务进行提交时,直接比较本事务的时间戳信息与数据存储中的时间戳信息是否存在冲突,即可以判断当前事务是否可以提交.

事务的具体提交逻辑如算法 3 所示,分为预提交和写入数据库两个阶段.在预提交阶段,对于每一条写记录,需要判断其时间戳区间是否与数据存储中的格结构时间戳区间存在冲突,如果任意一条写记录存在冲突,则事务预提交失败（3-7 行）.否则,事务预提交成功,进入写入数据库阶段,将写操作持久化到数据库中（8-10 行）.结合算法 1、2 的读写缓存算法和基于格结构的事务提交算法,Elsa 系统支持**快照隔离**的事务隔离级别.值得注意的是,为了优先保证系统的可用性和可拓展性,域内事务在提交时并不会强制对其他域间节点上的数据进行 Merge 操作.也就是说,不同域间节点上的事务可以无协调地提交,冲突消解和数据一致性由 CRDT 结构来异步地保证.所以,Elsa 系统所支持的快照隔离级别,在其标准定义的基础上做了宽松处理,即域内事务保证快照隔离级别,域间数据满足强最终一致性.

算法 3. 基于格结构的事务提交算法

输入：要提交的事务 *tid*, 事务开始时间戳 *startTs*, 事务提交时间戳 *commitTs*

```
1.  writeSet ← writeBuffer[tid]   // 获取事务需要提交的写操作集合
2.  for record in writeSet then   // prepare 阶段
3.      result ← preCommit(record.key, startTs, commitTs)
4.      if result = FALSE then    // 预提交失败,整个事务被废弃
5.          return FAILED
6.      endif
7.  endfor
8.  for record in writeSet then   // 预提交成功,正式提交事务到服务端并持久化写入结果
9.      persist(record.key, record.value, startTs, commitTs)
10. endfor
11. return SUCCESS
```

```
// 预提交阶段
1.  function preCommit(key, startTs, commitTs)
2.      oldLattice ← DB[key]   // 读取数据存储中的格结构
3.      if startTs > oldLattice.commitTs then // 格结构不存在时间戳区间冲突,预提交成功
4.          return TRUE
5.      endif
6.      return FALSE   // 预提交失败
// 写入数据库阶段,将结果写入服务端存储
1.  function persist(key, value, startTs, commitTs)
2.      oldLattice ← DB[key]    // 读取数据存储中的格结构
3.      oldLattice.value ← value
4.      oldLattice.startTs ← startTs
5.      oldLattice.commitTs ← commits
```

2.4 副本一致性的解决方案

为了提高 Elsa 系统的吞吐量和可拓展性,Elsa 系统被设计为一种支持跨域多活的去中心化的架构,不同的数据中心（服务端节点）可以同时接受用户发来的读写请求,各节点间是对等而非主从的关系.同时,综合考虑系统的性能和副本间一致性的权衡关系,Elsa 采用 state-based CRDT 结构来保证副本间的强最终一致性.

由于服务器节点间是对等的关系,所以服务器节点需要定期将本地的数据修改集（以格结构的形式组织）发送给集群中除自己之外的所有其他节点,同时自己接受所有其他节点发来的修改集并与本地的数据进行合并以达到全局副本的一致性状态.服务端节点间的数据合并过程如算法 4 所示,对于远程节点发来的修改集中的每个格结构,与本地存储中的对应格结构数据进行合并即可,合并的规则正如章节 1.1 中所说,可以根据业务需要进行自定义并保证满足 ACI 三条性质即可.算法 4 给出了基于**后提交胜利**(last-commit-win)规则的副本合并过程.

算法 4. 服务端节点间的数据合并算法

输入：远端服务端节点发来的数据修改集 *remote_changeset*

```
1.  for lattice in remote_changeset then
2.      key ← lattice.key
3.      if lattice.commitTs > DB[key].commitTs then   // 后提交胜利的合并规则
4.          DB[key].value ← lattice.value
5.          DB[key].startTs ← lattice.startTs
6.          DB[key].commitTs ← lattice.commitTs
7.      endif
8.  endfor
```

2.5 正确性证明

state-based CRDT 结构在论文[12]中首次被提出,并指出对于具备单调半格性质即满足**幂等性**、**可交换性**和

可结合性的数据类型,每个副本只需要定期将数据广播给其他副本并进行合并,即可以保证副本间的强最终一致性.

对于本文提出的支持快照隔离的格结构,我们假设两个格结构对象 $\mathbf{SIL_1}$ 和 $\mathbf{SIL_2}$ 进行合并,当 $\mathbf{SIL_1 = SIL_2}$ 时,显然:

$$\mathbf{SIL_1 \cup SIL_2 = SIL_1 = SIL_2}$$

则幂等性得证.需要说明的是,当前副本间需要进行广播的数据是在本阶段内被修改(update)的数据,即发送方广播部分写集,接收方进行合并(数据赋值)即可,能够保证幂等性.这种基于状态(state-based)的同步相较于基于操作(op-based)的同步方法的好处在于,对于一些同时包含读写的操作(例如自增操作),也能够保证幂等性.但是,这可能会导致部分操作被废弃,所以在不考虑通信开销的前提下,可以通过传递读集(read set)来解决.而当 $\mathbf{SIL_1}$ 和 $\mathbf{SIL_2}$ 不相等时,由于格结构的合并逻辑是提交时间戳(**commitTs**)大的格结构写入胜利,那么不妨假设 $\mathbf{SIL_1.commitTs < SIL_2.commitTs}$.可知格结构间的合并结果与运算顺序无关(满足单调性),而只与格结构的 **commitTs** 属性相关,可得:

$$\mathbf{SIL_1 \cup SIL_2 = SIL_2 \cup SIL_1 = SIL_2}$$

可交换性得证.同理,对于三个格结构 $\mathbf{SIL_1}$、$\mathbf{SIL_2}$ 和 $\mathbf{SIL_3}$ 进行两两合并时,不妨假设 $\mathbf{SIL_1.commitTs < SIL_2.commitTs < SIL_3.commitTs}$,那么根据后提交胜利的合并规则,可知:

$$\mathbf{(SIL_1 \cup SIL_2) \cup SIL_3 = SIL_2 \cup SIL_3 = SIL_3}$$
$$\mathbf{SIL_1 \cup (SIL_2 \cup SIL_3) = SIL_1 \cup SIL_3 = SIL_3}$$

可结合性得证.

2.6 无协调一致性模型的总结

模型中关于“无协调”的表述较为抽象,为了便于理解,我们将其总结概括为以下 3 点.1) 采用代理端缓存来解决脏读等读写异常问题,代理端节点与服务端节点通过消息传递进行通信,读写并发控制机制与数据存储分离,便于系统扩展.2) 采用 CRDT 格结构来检测和消解并发写写冲突,相较于悲观锁机制,协调开销大大降低.3) 基于异步消息的副本合并模型,结合 CRDT 的 ACI 特性,消除数据传输过程的乱序、重发等传输故障问题,保证全局副本的强最终一致性.

可以看到,基于 CRDT 的无协调一致性模型和传统的悲观锁、共识协议等机制都能够解决并发事务下的读写冲突、副本一致性问题,但它们之间是存在差别的.

悲观锁机制是一种协调事务并发冲突的机制,其制定的规则就是获得锁的事务可以执行而其他事务则需要等待.这是一种行之有效的方法,但是这期间产生了一些例如加锁、释放锁、线程等待等代价,这无疑增加了系统的额外协调开销.同时,加锁、释放锁等过程需要额外的第三方锁管理机制来进行统筹和管理.而 Paxos 等共识协议虽然能够保证分布式副本的强一致性,然而在一些跨域的分布式环境或是对副本一致性要求没有那么高的场景下,共识协议则不太适用.

而对于 CRDT 来说,其巧妙地将冲突消解的规则(merge 函数)封装在数据结构中,即把数据存储和计算逻辑组成了一个有机的整体,从而不再需要第三方协调机制,仅需要定期进行通信交换即可,大大降低了分布式系统中的协调开销,在跨域等远距离场景下尤其明显.

3 系统架构

系统的总体架构如图 6 所示,主要分为客户端(user client)、代理端(proxy)和服务端(server)三个模块.下面,我们将首先结合第 2 章提出的模型来对系统的事务处理流程和总体架构进行说明.

客户端模块提供给用户访问 Elsa 系统的多种接口,用户请求以事务为粒度转发给代理端模块.代理端模块

是连接客户端和服务端的中间件,主要负责负载均衡、读写并发控制等功能.代理端的快照管理器(snapshot manager)是域内事务读写并发控制的重要组件,主要包含读缓存和写缓存两个子模块,其对应的处理逻辑如章节 2 中的算法 1 和 2 所示.域内事务的提交过程由代理端发起,将通过验证的事务写入到服务端存储中.而对于域间的服务端副本,需要通过 Gossip 协议来定期广播增量 CRDT 数据来异步的保证强最终一致性.下面我们将对这 3 个重要模块的详细设计分别进行阐述.

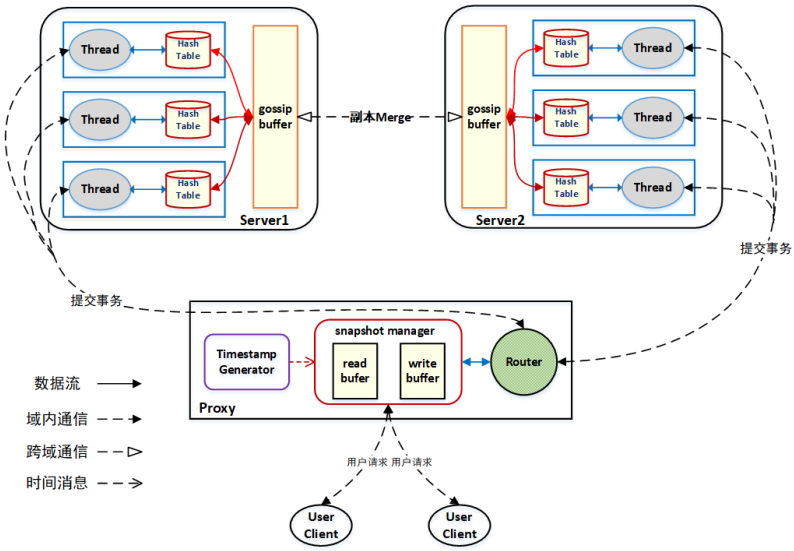


图 6 Elsa 系统架构图

3.1 客户端

客户端用于用户和代理端节点的交互过程.客户端和代理端采用请求-应答的通信模型来进行通信,保证了用户请求的及时响应.

3.2 代理端

代理端是客户端和服务端交互的中间组件,主要负责事务的并发控制和集群管理.

1) 快照管理器(snapshot manager): 负责事务快照的生成和管理,包含读缓存和写缓存两部分,解决脏读和不可重复读的并发读写异常问题.为了提高系统的可拓展性,我们将事务的并发控制模块和数据存储模块进行了剥离,即事务的并发控制在代理端,数据存储和操作执行都在服务端.

2) 时间戳生成器(timestamp generator): 用于生成全局唯一单调递增的时间戳,为格结构中的 **StartTS** 和 **CommitTS** 属性赋值.

3) 路由组件(router): 用于转发用户发来的读写请求到对应的服务端进行执行,并提供负载均衡服务.同时为了支持更好的动态可拓展性和高可用性,代理端还负责集群节点的管理,包括新节点的动态加入、服务端节点状态监控、宕机节点移除等.

3.3 服务端

服务端是系统的计算和存储节点,主要负责数据存储和操作执行.

1) 索引: Elsa 支持基于内存的键值存储,由于当前仅支持插入,更新和删除等基本的操作,所以索引结构选择了查询速度更快[23]实现更简单的哈希表而非更加复杂的 B+树[24-25]等树型索引.

2) 计算与执行模型: 为了充分利用多副本的加速请求处理的特点和实现更好的可拓展性,我们没有采用传统的共享内存的计算模型而是支持分布式高并发的 Actor 模型[26-27].Actor 是系统中最小的计算和存储单元,

由一条计算线程和一块私有的内存区域构成.Actor 之间通过消息传递进行通信,而不能直接访问对方的私有内存,从而解决了共享内存模型在高并发情况下由于线程争用而导致的性能下降问题.

同时为了满足无协调的设计需求,对于接收到的读写请求,我们规定仅需要在单个 Actor 上执行即可返回结果给用户,而不是需要将执行结果同步给所有的副本(无等待执行,wait-free).之后在一定的时间区间后通过 Gossip[28]协议来异步地广播更新过的数据以达到副本间的强最终一致性.Gossip buffer 就是用于存储本地更新结果的一块内存区域,在广播阶段结束后会被清空.

3) 分片与副本：为了加速系统的执行效率,对于一个 server 内的数据是以 Actor 为单位进行分片放置的,而多个 server 间的数据是全备份的,以保证系统的高可用性.对于涉及多个分片的事务,采用轻量级两阶段协议来保证事务的原子性.

4) 持久化日志：为了保证事务的持久化特性,Elsa 系统支持内存中数据的定期写入日志.不同于计算模型的无等待执行特性,持久化只会发生在域内和域间数据达到全局一致状态后,保证已落盘数据的确定性和可恢复性.

需要说明的是,我们将集群节点间相邻的两次数据同步过程间的时间间隔,定义为一个数据合并周期(merge epoch).由于 Elsa 的无等待执行的特点,所以在一个合并周期内对于相同数据的不同副本的事务有可能存在冲突并且同时提交成功,即在合并周期内的多副本间的数据是不一致的. 为了解决这个问题, 我们通过将合并周期尽可能的缩小的方法, 在保证系统吞吐量的前提下来减小数据不一致的窗口期, 从而保证多副本并行事务的正确性. 如章节 5.7 中的实验结果所示,在跨域的分布式环境下,合并周期最小可以设置为 100ms.用户可以根据业务逻辑对事务隔离性和副本一致性的需求程度来调整合并周期参数,从而满足特定的业务需求.

4 系统实现

Elsa 的系统实现参考了 Anna 系统的开源代码,代码总行数在 4000 行左右.系统的通信模块是在 ZeroMQ[29]消息队列库的基础上构建的,同时为了加快消息传输的速度和压缩消息传输量,我们采用了 Protobuf[30]序列化库作为传输协议.下面将分别从客户端,代理端和服务端三个最重要的部分结合具体的实现细节逐条说明.

4.1 客户端

用户通过键入命令来与系统进行交互,客户端会首先对用户键入的命令进行语法合法性检查,如果不合法则提醒用户否则将请求打包为 Protobuf 报文发送给代理端进行处理.

Elsa 当前支持 ***BEGIN***、***PUT***、***GET***、***DELETE***、***COMMIT*** 和 ***ABORT*** 这 6 种主要的操作.***PUT*** 操作用来对数据库的指定键值进行写入,语法格式为 ***PUT*** *key value*,数据存储中存在 *key* 时则为更新操作否则为插入操作.用户通过 ***GET*** 操作来读取键对应的值,其语法格式为 ***GET*** *key*.***DELETE*** 操作支持对键值对的删除,语法格式为 ***DELETE*** *key*,具体实现是将键对应的值设为空而不是直接将键从数据库中物理删除.Elsa 同时支持显式事务和隐式事务,被 ***BEGIN*** 和 ***COMMIT*** 或是 ***ABORT*** 命令包围的事务即为显式事务,其中可以包含多条 ***PUT*** 和 ***GET*** 命令.而对于单独出现的 ***PUT*** 和 ***GET*** 命令,系统将其视为由一条命令组成的隐式事务,并默认自动提交.

由于代理端的读写缓存是以事务为单位进行管理的,所以客户端需要保证本地事务号的全局唯一性.在同一台物理机器下可能同时存在多个线程并行的提交事务,所以我们设置了由三级标识符组成的全局 **tid = *ip_thread-id_counter***,由 IP 地址、线程号和线程私有的自增计数值组成的字符串.

4.2 代理端

代理端的读写缓存采用哈希表进行组织,具体实现时采用了 C++ STL 标准库中的 unordered_map 结构.对于时间戳生成器(timestamp generator),我们采用了类似于 Percolator[31]的 TSO 全局发号器的集中式实现.相比较于 Google Spanner[32]提出的 True Time 和混合逻辑时钟 HLC[33],这种方案具备实现简单、严格定序、实现成本低和性能较好的优点,符合 Elsa 的设计目标.

代理端的路由模块是系统性能和负载均衡的重要依赖,理想的路由模块应该能够使得整个系统的负载达

到动态平衡从而最大化利用系统的硬件资源.Dynamo 是亚马逊提出的键值存储平台,为了提高系统的可拓展性和均衡负载,使用了一致性哈希算法(consistent hashing)[34]作为路由协议.一致性哈希算法不同于传统的哈希算法,一方面其将 key 值路由到距其顺时针方向最近的服务器节点,这种方法能够保证在服务器节点增加或减少时多数 key 值哈希结果的不变性,增强了服务器的动态可拓展性.另一方面,通过增加虚拟节点来均衡系统的负载,理论上在虚拟节点足够大时能够保证负载的均匀分布.Elsa 将虚拟节点数量设置为 1024.

在 Elsa 中的代理端路由组件中,用户对某个 key 值的请求会首先通过一致性哈希获取 key 值所属的所有副本,之后通过随机哈希算法来选取其中的一个副本发送请求并执行.一方面带有虚拟节点机制的一致性哈希保证了副本均匀地被分配给服务器节点,另一方面通过将请求随机的分配给任一副本也保证了请求负载的相对均衡.

4.3 服务端

CRDT 本质上是对数据和冲突消解算法在逻辑上的封装,是副本间同步策略的一种面向数据结构的描述,冲突消解算法如何实现完全由编程人员决定,所以具备很强的自定义性和灵活性.CRDT 内部保存副本数据,只提供 Query、Update 和 Merge 接口与外界进行交互.为了不破坏 CRDT 的可拓展性和灵活性,我们采用了 C++的模板和多态技术,定义了 CRDT 结构的统一接口,遵循接口规范可以自定义多种 CRDT 结构.

在 Elsa 的设计目标下,服务端 Actor 线程多数时间都会被用来为用户提供服务,Actor 之间无协调的向前执行.为了实现该目标,服务端采用了基于 epoll 的 I/O 多路复用技术,监控 socket 事件并触发请求处理流程.在高并发即存在大量的 socket 连接(但活跃的 socket 较少)的情况下,epoll 相较于 select/poll 机制有明显的性能提升.

Gossip 缓冲区用于本地副本的 CRDT 更新结果的存储,直到本轮次的更新阶段结束后进入广播阶段才通过 Gossip 协议将更新结果广播给其他副本.Gossip 协议保证了消息最终会异步地被接收方收到,以达到全局副本的强最终一致性.系统进程间和跨节点的通信全部使用 ZeroMQ 消息队列库实现,使用 Protobuf 库对消息进行序列化和反序列化以减少通信量,降低通信开销.

5 实验与结果

为了评估 Elsa 的性能,本文将 Elsa 与流行的关系型数据库 MySQL[35]、openGauss MOT[36]内存引擎、文档型数据库 MongoDB[37]和键值存储 Cassandra[38]在单机环境下进行了吞吐量对比,并在跨域的分布式环境下与 MongoDB 集群、Cassandra 集群进行了性能对比.除此之外,本文还对 Elsa 的性能随着工作线程、副本数等系统参数变化的吞吐量变化趋势在分布式环境下做了相应的实验.

5.1 实验准备

5.1.1 跨区域分布式实验环境

跨区域分布式实验环境由 **4 个区域的 8 台阿里云 ecs.c5.2xlarge 实例**组成,分别位于北京(华北),杭州(华东),深圳(华南),成都(西南),每个区域各 2 个实例.每个 ecs.c5.2xlarge 实例配有 8 核 2.5GHz 的 intel 处理器、16GB 运行内存、40GB SSD 以及 100Mbs 以太网卡.操作系统均采用 Ubuntu 16.04 LTS.

5.1.2 单机实验环境

单机实验采用了一台计算机,配有 AMD Ryzen 5 2600X 3.60GHz 6Core 处理器,100Mbs 以太网卡,16G 运行内存,256G SSD,操作系统采用 Ubuntu 16.04 LTS.

5.1.3 对比系统

为了充分测试 Elsa 系统提出的无协调一致性模型相较于悲观锁、乐观并发控制和 MVCC 等机制的性能优势,我们与 MySQL、openGauss MOT 和 MongoDB 等采用不同技术以支持快照隔离事务的数据库系统进行了性能对比实验.同时,为了测试 Elsa 系统在跨域分布式环境下的系统吞吐量和可拓展性,我们与 MongoDB 集群、Cassandra 集群进行了大规模的对比实验.下面,我们对这些对比系统进行简单介绍.

a) MySQL 是一款流行的关系型数据库,广泛的应用于各大公司和企业,采用 MVCC 技术实现了读已提交

和可重复读的隔离级别,在实验中我们将 MySQL 设置为可重复读的隔离级别.

b) openGauss 是华为开源的关系型数据库,支持行列存储和内存等多种存储引擎.实验采用了基于 OCC 技术来保证事务并发控制的 openGauss MOT 内存引擎.

c) MongoDB 是一款基于文档的 NoSQL 数据库,支持海量数据的读写与存储,实验中采用了其基于 MVCC 技术实现的快照隔离级别.

d) Cassandra 是一款开源的 AP 类分布式 NoSQL 数据库,具有良好的拓展性并保证副本的最终一致性.

5.1.4 负载设置

为了充分测试 Elsa 在不同并发争用负载下的系统吞吐量,我们在 **YCSB** 测试框架的基础上,对其进行了事务测试的扩展改造.负载粒度由单个读写请求变为由多个读写请求组成的事务,并包含一些只读事务、超长事务等特殊负载,能够更全面地测试系统的事务处理能力,称之为 **YCSB-T**.对于 MySQL、openGauss MOT 关系型数据库,我们将 ***GET*** 和 ***PUT*** 操作转化为数据表的相应 **SELECT** 和 **UPDATE** 操作.为了保证实验的公平性,我们将 Elsa 系统的事务隔离级别设置为快照隔离,并对所有系统都创建主键索引以最大化系统性能.

5.2 高并发争用下单机吞吐量对比

本节首先测试在高并发争用下 (zipf coefficient = 4)的系统吞吐量变化,每个并发线程的负载由 10000 个读/写请求组成.图 7 显示了 Elsa (replica = 3),Elsa(replica = full),openGauss MOT,MySQL,MongoDB 以及 Cassandra 的吞吐量随着并发请求线程数量的增加的变化,replica 代表 Elsa 的副本数(full 代表全备份,本实验中值为 16),每种并发线程条件下的吞吐量最大值在图中标出.实验结果显示,在并发线请求程数小于 14 时,Elsa (replica = 3)和 Elsa (replica = full)的吞吐量相差不大且优于其他系统,而在并发请求线程数大于等于 14 时,Elsa (replica = full)的吞吐量则远远高于其他系统.

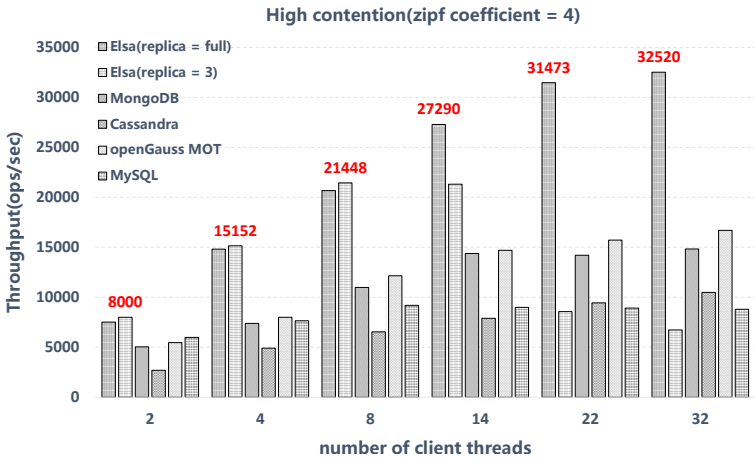


图 7 高并发争用下的单机吞吐量对比

在高并发情况下,MySQL、MongoDB 等采用锁机制来解决写写冲突的系统会出现死锁检查、加解锁等额外的协调开销,从而导致了吞吐量的下降.openGauss MOT 采用的 OCC 机制相较于锁机制,没有过多的协调开销,随着并发线程数的增加吞吐量呈现较缓慢的上升趋势.Elsa 由于采用了 Actor 编程模型和基于 CRDT 的一致性模型,系统的协调开销大大降低,在高并发争用的情况下性能很好.

5.3 低并发争用下单机吞吐量对比

在低并发争用下(zipf coefficient = 1)系统的吞吐量实验结果如图 8 所示,可以看到,在并发请求线程数小于 14 时,MySQL 吞吐量高于 Elsa (replica = full)、Elsa(replica = 3),但差距不大.这是由于此时事务的并发度相对较低,MySQL 所采用的锁机制较为高效,没有产生大量的协调开销从而降低系统的吞吐量.

而随着并发线程数的增加,MySQL 的吞吐量有所下降,Elsa (replica = 3)相较于 Elsa(replica = full),由于副本数较少从而减少了大量副本的通信开销,所以吞吐量相对较高.在低并发的情况下,并发请求所访问的数据项分布较为均匀,所以相较于高并发请求的情况下,Elsa 的通信开销会相应的增加,从而导致在并发请求线程数小于 14 时,Elsa 的性能略低于 MySQL.

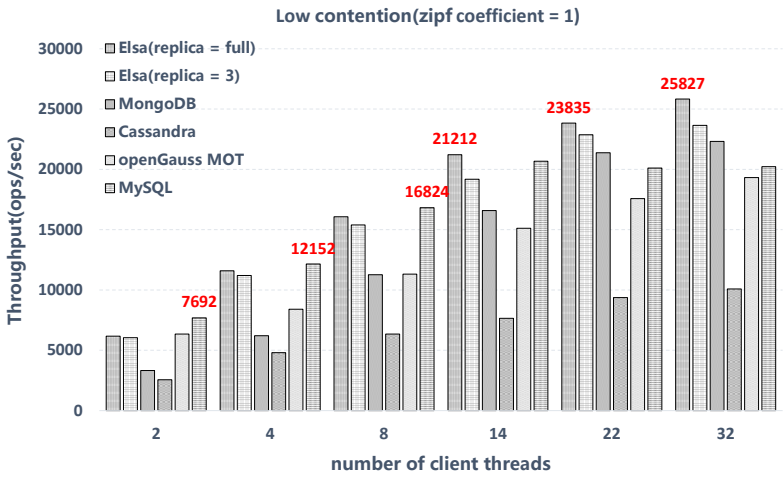


图 8 低并发争用下的单机吞吐量对比

5.4 跨域分布式集群的吞吐量对比

如图 9 和 10 所示,分别是在跨域的分布式环境下,Elsa 与 MongoDB 集群、Cassandra 集群在不同争用程度的负载下的吞吐量对比图.为了保证实验的公平性,我们将副本数全部设置为 8.

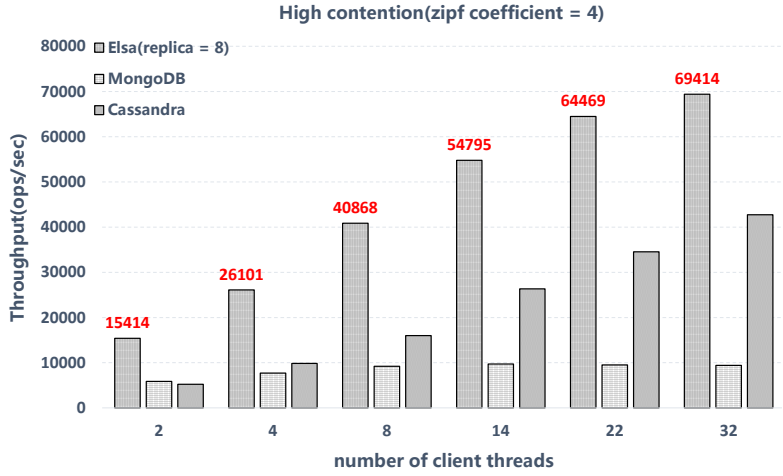


图 9 跨域分布式环境下高并发争用负载的吞吐量对比

在高并发争用负载下,如图 9 所示,可以看到 Elsa 系统的吞吐量随着并发线程数的增加,吞吐量呈现明显的上升趋势,并且远远优于 MongoDB 和 Cassandra 集群.由于 MongoDB 集群仅支持主备架构,只能在主副本执行写操作并同步给从副本,所以大量的并发争用会产生大量的加锁、解锁等开销,使得系统的吞吐量大大降低.Cassandra 集群中的副本是允许同时读写的,所以吞吐量也是呈现上升的趋势.

而在低并发争用负载下,如图 10 所示,可以看到相较于高争用负载,在低争用负载下,MongoDB 的吞吐量有

所提升,但随着并发线程数量的增加,吞吐量的增长逐渐放缓并最终持平,这也是主备架构的数据库集群的缺点,可拓展性并不好.而基于多主架构的 Elsa 和 Cassandra 的可扩展性则相对较好.

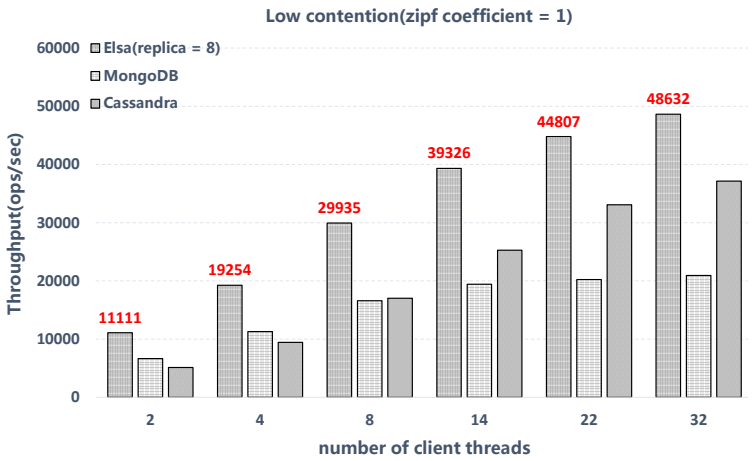


图 10 跨域分布式环境下低并发争用负载的吞吐量对比

5.5 不同隔离级别下系统的吞吐量对比

本文提出的事务一致性模型支持读已提交、可重复读和快照隔离的 3 种弱一致性隔离级别.在本节实验中,我们将负载设置为 32 个并发请求线程,副本数固定为 8,测试在不同争用率负载下,Elsa 系统在不同隔离级别下的吞吐量.

实验结果如图 11 所示,可以看到在高争用(zipf = 4)以及低争用(zipf = 1)负载条件下,随着隔离级别的提高,吞吐量都呈现轻微下降的趋势.相较于读已提交,由于可重复读级别需要执行读集验证,快照隔离级别需要解决丢失更新异常问题,所以会使得部分事务被废弃,从而导致吞吐量轻微下降.

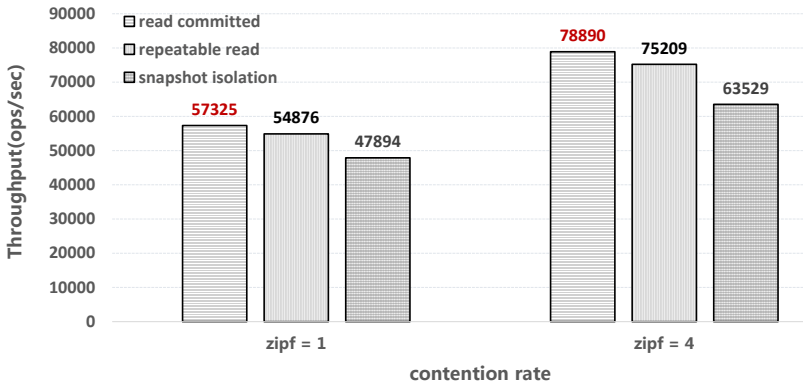


图 11 不同隔离级别下的吞吐量

5.6 工作线程数对性能的影响

Elsa 中的工作线程是最小的工作单元,由一条 CPU 线程和一块私有内存区域组成,负责接收并处理用户发来的读写请求.在本节实验中,我们将负载固定为 32 个并发请求线程,每个线程由 20000 个读写请求组成,通过增加工作线程(集群节点数量)来测试系统在分布式环境下的可拓展性.

从图 12 的实验结果图可以看到,随着分布式集群中节点数量的增加,系统的吞吐量变化呈现近似线性的趋

势.这是 Actor 模型相较于共享内存模型的优势,对于共享内存模型来说,工作线程数的增加虽然会加快执行效率但也增加了对共享内存的争用开销,所以导致在共享内存模型下工作线程的增加并没有带来系统吞吐量的线性增加. 同时, 由于系统的无等待执行和副本间强最终一致性的特点, 使得集群中节点的大量时间和资源都花在响应用户请求而非通信过程, 从而保证了系统的可扩展性.

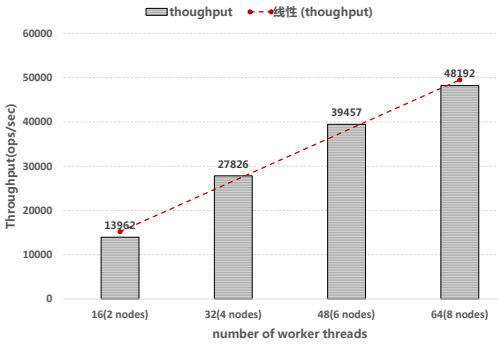


图 12 不同工作线程下的吞吐量

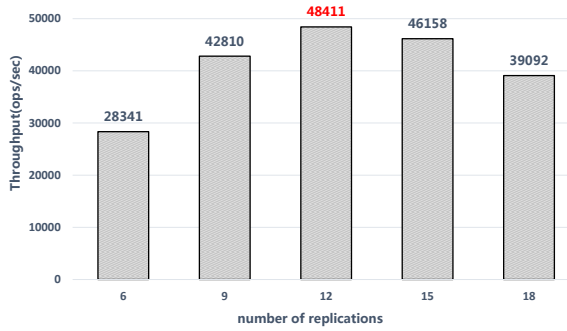


图 13 不同副本数下的吞吐量

5.7 副本数对性能的影响

Elsa 的设计目标是多副本同时提供读写服务的多主架构,所以随着副本数的增加理论上会带来系统吞吐量的提升.在本节实验中,我们将负载固定为 32 个并发请求线程,每个线程由 20000 个读写请求组成,通过调整系统的副本数量来测试系统的吞吐量变化.

如图 13 所示,随着副本数的增加,系统的吞吐量有着明显的提升.而当副本数由 12 增长为 15 时,我们发现系统的吞吐量出现了下降趋势.经过分析后,我们认为这是由于副本数大于线程并发度以及副本间的通信开销所带来的性能下降.具体来说,当对同一数据项的并发请求数小于副本数的时候,多余的副本处于非活跃状态即没有为用户提供读写服务,然而在副本间进行同步时,这些非活跃的副本也是会参与同步过程的,带来了相应的开销,起了“反作用”.所以,副本数是与负载紧密相关的,准确的选择副本数对于系统的性能提升十分重要.

5.8 副本同步周期对性能的影响

副本同步的时间周期是指各 server 节点上的副本定期进行数据同步的时间间隔大小.一般来说,时间周期越小,则节点间的同步过程越频繁,副本数据不一致时间就越短,系统的同步开销就越大.在本节实验中,通过改变 Elsa 的副本同步周期大小来测试相同负载下的系统运行时间变化结果.系统负载仍然由 32 个并发线程组成,每个线程模拟发送 20000 个读写请求.为了使实验结果更加准确和广泛,我们还设置了不同的并发争用率(zipf)条件来测试系统性能的变化.

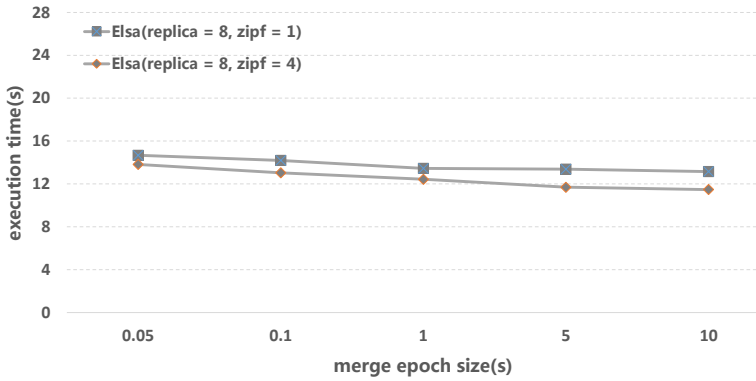


图 14 不同副本同步周期下的运行时间

实验结果如图 14 所示,可以看到在不同实验条件下,随着副本同步周期的减小,系统的运行时间没有明显的变化,呈现出轻微的下降趋势.一方面是由于同步数据副本的 Gossip buffer 中存储的是该周期内发生变化的副本数据而非全部数据,即节点间的副本同步是以增量而非全量的形式进行的.所以,当同步周期减小时,数据同步过程中产生的通信数据量并没有增加,而只是通信次数增加而单次通信的数据量却在降低.另一方面,由于 CRDT 格结构满足可结合和可交换性,所以在发送副本数据之前对于同一数据项的更新操作,可以在本地进行预先合并,这样能够大大减少通信数据量,在高争用的负载上尤其明显.实验结果显示,在该实验环境下,副本同步周期取值为 100ms ~ 1s 较为合适.

5.9 跨域同步协调开销对性能的影响

在跨域架构下的分布式系统中,跨域副本间的同步协调开销会对系统的吞吐量产生巨大影响甚至可能成为性能瓶颈.在本节实验中,我们将 Elsa 系统的跨域同步机制改为跨域两阶段提交(2PC),以此来验证基于 CRDT 的一致性事务模型的有效性.系统负载仍然由 32 个并发线程组成,每个线程模拟发送 20000 个读写请求,副本个数设置为 8.

实验结果如图 15 所示,在高并发和低并发争用负载下,Elsa(CRDT)的吞吐量都要明显优于 Elsa(2PC),这正是 CRDT 结构的优势所在.可以看到,随着并发争用率的提高,Elsa(CRDT)的吞吐量没有下降而是上升.一方面是由于无等待执行的多主架构带来的系统并发量的增加,另一方面,是由于 CRDT 的 ACI 特性允许副本更新在本地优先合并,从而带来了通信开销的降低.然而对于 Elsa(2PC)来说,随着争用率的提升,2PC 机制所带来的协调同步和通信开销却使得系统的吞吐量显著下降.

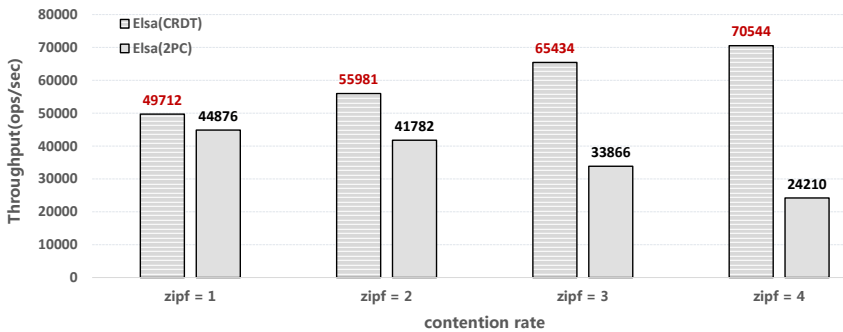


图 15 不同跨域同步机制下的吞吐量

6 总结与未来工作

本文提出了一种面向跨区域架构的无协调键值存储系统 Elsa.为了优先满足系统的性能和可拓展性,降低额外的系统协调开销,采用了一种基于 CRDT 技术的无协调一致性模型而非读写锁、共识协议等机制来解决并发冲突和副本一致性问题.同时,Elsa 采用了 Actor 的编程模型,线程间无协调的处理用户请求,并保证副本间的强最终一致性,保证系统将多数时间花费在处理用户请求上,从而提升系统的吞吐量.

实验结果显示,在跨域的分布式环境下,Elsa 的吞吐量明显高于流行的 Cassandra、MongoDB 等分布式数据存储系统.在高并发争用的负载下,Elsa 的无协调机制优势明显,吞吐量最高达到 MongoDB 的 7.37 倍、Cassandra 的 1.62 倍.同时,Elsa 也存在着一些需要进行优化和还未完成的工作,例如降低采用 Gossip 协议的广播过程的通信开销、副本数动态调整以最大化系统性能等.期待能在未来的工作中进行优化与实现.

References:

[1] SEEGER M. Key-value stores: A pracitical overview. Computer Science and Media. Ultra-Large-Sites, 2009, 2009, 9: 1-21.

[2] Brewer E. A certain freedom: thoughts on the CAP theorem. Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. 2010: 335-335.

[3] Brewer E. CAP twelve years later: How the" rules" have changed. Computer, 2012, 45(2): 23-29.

[4] Brewer E A. Towards robust distributed systems. PODC. 2000, 7(10.1145): 343477.343502.

[5] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store. ACM SIGOPS operating systems review, 2007, 41(6): 205-220.

[6] Lu Y, Yu X, Cao L, et al. Aria: a fast and practical deterministic OLTP database. Proceedings of the VLDB Endowment, 2020, 13(12): 2047-2060.

[7] Sovran Y, Power R, Aguilera M K, et al. Transactional storage for geo-replicated systems. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. 2011: 385-400.

[8] Lamport L. Paxos made simple. ACM Sigact News, 2001, 32(4): 18-25.

[9] Wang Jiang, Zhang Mingxing, Wu Yongwei, Chen Kang, Zheng Weimin. Paxos-like Consensus Algorithms: A Review. Journal of Computer Research and Development, 2019, 56(4): 692-707 (in chinese).

[10] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. 2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14). 2014: 305-319.

[11] Preguiça N, Baquero C, Shapiro M. Conflict-free replicated data types (CRDTs). arXiv preprint arXiv:1805.06358, 2018.

[12] Shapiro M, Preguiça N, Baquero C, et al. Conflict-free replicated data types. Symposium on Self-Stabilizing Systems. Springer, Berlin, Heidelberg, 2011: 386-400.

[13] Alvaro P, Conway N, Hellerstein J M, et al. Consistency Analysis in Bloom: a CALM and Collected Approach. CIDR. 2011: 249-260.

[14] Hellerstein J M, Alvaro P. Keeping CALM: when distributed consistency is easy. arXiv preprint arXiv:1901.01930, 2019.

[15] Alvaro P, Conway N, Hellerstein J M, et al. Blazes: Coordination analysis for distributed programs. 2014 IEEE 30th International Conference on Data Engineering. IEEE, 2014: 52-63.

[16] Wu C, Faleiro J, Lin Y, et al. Anna: A KVS for Any Scale. 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018: 401-412.

[17] Yu W, Rostad S. A low-cost set CRDT based on causal lengths. Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data. 2020: 1-6.

[18] Kleppmann M. Moving elements in list CRDTs. Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data. 2020: 1-6.

[19] Auvolat A, Taïani F. Merkle search trees: Efficient state-based crdts in open networks. 2019 38th Symposium on Reliable Distributed Systems (SRDS). IEEE, 2019: 221-22109.

[20] Ji Y, Wei HF, Huang Y, Lü J. Specifying and verifying CRDT protocols using TLA+. Ruan Jian Xue Bao/Journal of Software, 2020,31(5):1332−1352 (in Chinese).

[21] Bailis P, Davidson A, Fekete A, et al. Highly available transactions: Virtues and limitations. Proceedings of the VLDB Endowment, 2013, 7(3): 181-192.

[22] Berenson H, Bernstein P, Gray J, et al. A critique of ANSI SQL isolation levels. ACM SIGMOD Record, 1995, 24(2): 1-10.

[23] Ji, Y., Chai, Y., Zhou, X. et al. Smart Intra-query Fault Tolerance for Massive Parallel Processing Databases. Data Science and Engineering, 2020, 5(1): 65–79.

[24] Mao Y, Kohler E, Morris R T. Cache craftiness for fast multicore key-value storage. Proceedings of the 7th ACM european conference on Computer Systems. 2012: 183-196.

[25] Lehman P L, Yao S B. Efficient locking for concurrent operations on B-trees. ACM Transactions on Database Systems (TODS), 1981, 6(4): 650-670.

[26] Hewitt C. Actor model of computation: scalable robust information systems. arXiv preprint arXiv:1008.1459, 2010.

[27] Vernon V. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, 2015.

[28] Demers A, Greene D, Hauser C, et al. Epidemic algorithms for replicated database maintenance. Proceedings of the sixth annual ACM Symposium on Principles of distributed computing. 1987: 1-12.

[29] ZeroMQ. https://github.com/zeromq/zeromq4-x.

[30] Feng J, Li J. Google protocol buffers research and application in online game. IEEE conference anthology. IEEE, 2013: 1-4.

[31] Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. 2010.

[32] Corbett J C, Dean J, Epstein M, et al. Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 1-22.

[33] Kulkarni S, Demirbas M, Madeppa D, et al. Logical physical clocks and consistent snapshots in globally distributed databases. The 18th International Conference on Principles of Distributed Systems. 2014.

[34] Karger D, Sherman A, Berkheimer A, et al. Web caching with consistent hashing. Computer Networks, 1999, 31(11-16): 1203-1213.

[35] MySQL. https://www.mysql.com/cn/.

[36] Avni H, Aliev A, Amor O, et al. Industrial-strength OLTP using main memory and many cores. Proceedings of the VLDB Endowment, 2020, 13(12): 3099-3111.

[37] MongoDB. https://www.mongodb.com/zh-cn.

[38] Cassandra. https://cassandra.apache.org/_/index.html.

附中文参考文献:

[9] 王江,章明星,武永卫,陈康,郑纬民.类 Paxos 共识算法研究进展.计算机研究与发展,2019,56(4): 692-707.

[20] 纪业,魏恒峰,黄宇,吕建.CRDT 协议的 TLA+描述与验证.软件学报,2020,31(5):1332−1352.