软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(10):3203−3218 [doi: 10.13328/j.cnki.jos.006023]

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

构建新型高性能与高可用的键值数据库系统*

朱阅岸[1,2], 简怀兵[2], 龙永超[2], 李 彬[2], 王 树[2], 吴喜亮[2], 钟治初[1], 张延松[3]

[1](嘉应学院 计算机学院,广东 梅州 514015)

[2](广州欢聚时代信息科技有限公司,广东 广州 511442)

[3](中国人民大学 中国调查与数据中心,北京 100872)

通讯作者: 朱阅岸, E-mail: iwillgoon@126.com

摘 要: 近年来,写密集型应用程序越来越普遍.如何有效地处理这种工作负载,是数据库系统领域深入研究的方向之一.写操作开销主要由以下两个方面的因素构成:(1) 硬件级别,即写操作引起的 I/O,目前无法在短时间内消除这种开销;(2) 软件开销,即修改内存数据拷贝以及构造日志记录造成的多次写操作.日志即数据(log-as-database,称其为单拷贝系统)的架构能够减少写操作引起的 I/O,同时降低软件方面的开销.目前,业界对单拷贝系统展现出浓厚的兴趣.现有的单拷贝系统大部分建立在特殊的基础设施之上,例如 infiniband 或 NVRam(非易失性随机存取存储器),这种基础设施尚未达到广泛可用或者是依托他系统(例如 Dynamo)构建,这种方法缺乏灵活性与普适性.在商用机器环境中,自底向上构建了一个称为 LogStore 的键值数据库系统,采用 log-as-database 设计理念,以充分利用单拷贝系统的优点,在提升写操作性能的同时,有效缩短主备数据之间的差距.在系统中内嵌复制协议达到高可用性而不是依赖其他系统,使得系统灵活可控.系统新颖的查询执行模型将执行线程与特定分片绑定,结合多版本并发控制技术,以无锁的方式消除读写冲突、写写冲突以及上下文切换开销.用 YCSB 对系统性能进行了详细的评估,对比主流的键值系统 HBase 以及单拷贝系统实现 LogBase,LogStore 在写密集型工作负载上性能要优 4 倍左右.在崩溃恢复方面,LogStore 可在 1 分钟之内完成 TB 级别数据规模的恢复,比 LogBase 要快 1 个数量级以上.

关键词: 单拷贝系统;复制协议;写优化;恢复;细粒度缓冲区管理

中图法分类号: TP311

中文引用格式: 朱阅岸,简怀兵,龙永超,李彬,王树,吴喜亮,钟治初,张延松.构建新型高性能与高可用的键值数据库系统.软件学报,2021,32(10):3203−3218. http://www.jos.org.cn/1000-9825/6023.htm

英文引用格式: Zhu YA, Jian HB, Long YC, Li B, Wang S, Wu XL, Zhong ZC, Zhang YS. Building new key-value store with high performance and high availability. Ruan Jian Xue Bao/Journal of Software, 2021,32(10):3203−3218 (in Chinese). http://www.jos.org.cn/1000-9825/6023.htm

Building New Key-value Store with High Performance and High Availability

ZHU Yue-An[1,2], JIAN Huai-Bing[2], LONG Yong-Chao[2], LI Bin[2], WANG Shu[2], WU Xi-Liang[2], ZHONG Zhi-Chu[2], ZHANG Yan-Song[3]

[1](School of Computer Science, Jiaying University, Meizhou 514051, China)

[2](Guangzhou YY Inc., Guangzhou 511442, China)

[3](National Survey Research Center at Renmin University of China, Beijing 100872, China)

Abstract: In recent year, the write-heavy applications are more and more prevalent. How to efficiently handle this sort of workload is one of intensive research direction in the field of database system. The overhead caused by write operation is mainly issued by two factors. One is the hardware level, i.e., the IO cost caused by write operation. This cost cannot be removed in short period. The other is dual-copy

* 基金项目: 国家自然科学基金(61772533)
Foundation item: National Natural Science Foundation of China (61772533)
收稿时间: 2019-08-06; 修改时间: 2019-11-23; 采用时间: 2020-02-14

software architecture, i.e., multiple writes caused by modifying in-memory data copy and formulating log records. The log-as-database architecture (the following refers it as single-copy system) can reduce the IOs and software cost caused by write as well. But existing systems treating log-as-database either are built on top of special infrastructure such as infiniband or NVRam (non-volatile random access memory) which is far from widely available or is constructed with the help of other system such as Dynamo, which is lack of flexibility and generality. This study builds from scratch a single copy system called LogStore oriented for commodity environment, which adopts log-as-database design philosophy to fully utilize its advantages that can boost the write performance and minimize the gap between primary and secondary. Embedding consensus module into system other than dependent on auxiliary systems makes it more flexible and controllable. The novel execution model binding thread to certain partition plus multi-version concurrency control technique eliminates read-write, write-write conflict, and context switch overhead in lock-free style. The YCSB benchmark is used to assess system performance thoroughly. Compared to prevalent key-value store HBase and its single-copy implementation LogBase, the proposed system can achieve about 4x better. In term of crash recovery, LogStore can finish recovery within one minute for TB scale data volume, which is one order of magnitude recovery time less than LogBase.

Key words: single-copy system; replication protocol; write optimized; recovery; fine-grain buffer management

正如雅虎所经历的,在 Web 2.0 时代,越来越多的应用程序从读密集型转变为写密集型[1].在 2010 年~2012 年期间,其处理的写请求比率从大约 10%提高到 50%.在某种意义上,这种现象是伴随着社交网络工作场景而产生的,例如 Facebook、Twitter 和微信遇到的,其中,不同的用户发出许多更新,稍后他们通过单个读取操作来检索所有新帖子.除社交网络场景外,摄取流数据(如用户点击和传感器读数)的基础架构也面临大量写入密集型工作负载.它们最重要的功能是能够应对所有输入.由于缺乏强大的写入能力,系统不得不丢弃部分数据或阻塞用户请求,因而损害到用户的体验.此外,某些应用也极大地影响着我们的日常生活,例如金融交易或电子商务促销,这些应用本质上也是写作密集型负载.因此,开发具有高写入吞吐量、低延迟、容错和低存储使用率的存储系统,同时提供即时数据恢复能力,是很有现实意义的.

存储系统广泛采用预写日志[2]的方法来实现最大写入吞吐量,同时提供持久性保证.在该体系架构中,随机写操作转换为对日志文件顺序写入.这种操作被称为非强制刷盘策略.在该策略下,当操作完成时,不需要将数据修改写回磁盘,只需写入增量日志.系统崩溃的时候,日志可用于恢复系统.该体系结构称为双拷贝存储(data+log).双拷贝存储有 3 个缺点.

- 首先,虽然它可以将修改后的数据推迟写回,但当达到时间或容量阈值时,即在超过检查点间隔或脏页面百分比超过高水位线之后,所有数据都必须刷新到持久存储中,成为写密集型应用程序的瓶颈;
- 其次,双拷贝存储需要在日志和数据存储之间进行细致的同步(确定检查点,依次回放日志记录),增加了软件逻辑的复杂度,并且不可避免地影响内存驻留工作负载的性能.此外,复杂的软件需要更多的代码维护和测试成本.实际上,正如 stonebraker 所指出的,DBMS 实际上由两个 DBMS 组成:一个系统管理数据,另一个管理日志[3];
- 最后,在双拷贝存储下,数据库和日志之间存在着显著差距.因此,当遇到系统故障时,需要针对数据库回放日志以将其转换到最新状态,从而导致明显的停服时间.

双存储系统还会在高可用性场景下遇到麻烦,其中,热备服务器在主机崩溃时刻接管到达系统的请求.由于备机数据通常落后于主机,因此备机必须重放已接收的日志,该情况与发生单机故障的情况相同.

为了满足 Web 应用程序对高性能和高可用的底层设施的需求,我们遵循日志即数据的设计理念,开发了一个称为 LogStore 的键值(key-value)数据库系统.我们验证 LogStore 可以达到读/写高吞吐量,并具有良好的高可用性规范.高可用性规范定义为以下两个相关因素:(1) 当主服务器崩溃时,备机能够以多快的速度成为主服务器,并对外提供服务;(2) 如果备机提供只读请求,它可以提供何种程度的数据可见性.虽然已有 DBMS 尝试使用日志作为唯一的数据存储(即单拷贝数据库系统,log as database),但它们构建在 Amazon S3 或特殊硬件(例如非易失性存储器,Infiniband)之上.我们在商用环境中从零开始构建 LogStore,仔细评估了单拷贝数据库系统的特点.本文的贡献如下:

(1) 自底向上构建了一个称为 LogStore 高性能键值的数据库系统.LogStore 采用日志即数据的思想,最大

化写入吞吐量,同时提高备份的数据可见性.LogStore 在商用机器环境中从零开始构建,内嵌复制模块,在这种情况下,我们可以更透彻地审视单拷贝系统带来的好处;

(2) 新颖的查询执行模型将线程绑定到特定数据分片,结合多版本并发控制技术,以无锁的方式消除并发控制开销和上下文切换.实验结果表明:这种方法可以充分利用底层硬件资源,系统具有线性扩展性;

(3) 我们对单拷贝系统的特点进行了仔细的评估,测试了系统的读写性能、高可用特性,特别是对单拷贝系统的索引访问开销进行了评估.

本文第 1 节是相关工作,对日志即数据的 DBMS 和键值数据库系统进行阐述和分析.第 2 节介绍我们的系统设计,重点论述 LogStore 读写流程以及并发控制机制.第 3 节对系统进行实验评估,验证单拷贝系统对性能和高可用性规范所带来的提升.第 4 节是总结.

1 相关工作

LogStore 涉及若干研究领域,如键值数据库系统、单拷贝系统实现、热备和数据缓存等.本节阐述这些领域的最新研究进展.

- 键值数据库系统

过去 10 年涌现出许多新型的可扩展数据管理系统,这类统称为 NoSQL 的系统废弃了关系型数据库的诸多限制,特别是牺牲一致性以达到可扩展需求.键值系统是 NoSQL 系统的典型代表.HBase[4]是最早、也是应用最广泛之一的可扩展分布式键值系统,其完全遵照谷歌公司的 BigTable[5]模型进行设计开发.HBase 可将数据存储在一张拥有数亿条数据、数百万列的大表中,通过唯一的 *key* 与 *column family* 定位唯一的数据单元,即{*key*, *column family*,*version*}→*cell*.RocksDB[6]是从 Google 的 LevelDB[7,8]派生出来的,由 Facebook 开发维护.它被设计成一个嵌入式、高性能、持久的键值存储系统.Facebook 的开发人员在数据层级大小和压缩策略方面优化了 RocksDB.除了 Facebook 之外,雅虎、领英和 MongoDB 等许多其他公司都将 RocksDB 作为存储的基础设施.但是 RocksDB 使用 WAL+data 双拷贝架构.目前的研究结论是,这一存储模型对于灾难恢复和高可用性规范不友好.微软公司开发了 Faster 高性能键值存储系统[9].无闩锁数据结构、二级存储的无缝集成和原地更新,使 Faster 获得了优越的性能.Faster 采用类似二次机会置换的缓冲区管理算法.当数据记录从磁盘中读入内存时,它被保留在称为混合日志的热区域中,其中数据可以就地更新.随着时间的推移,记录被移动到只读区域,最后被置换回磁盘.LogStore 使用类似的缓冲区管理算法,随机选择数据项移动到冷却区域.如果在一个时间窗口内,该数据量未被再次访问,那么系统会将该数据项置换出磁盘.与 Faster 不同,LogStore 不仅关注单机性能,还将系统高可用纳入考察范围.Bitcask[10]是一个只支持追加操作的键值存储系统,并使用日志作为唯一的数据存储,但它必须确保内存足够大,以容纳关键字空间.

- 日志即数据架构(log as database,也被称为单拷贝架构)

亚马逊 Aurora 开发团队认为,数据处理的瓶颈已从存储层转向网络层[11].因此,Aurora 只通过网络层传输 redo 日志到 Amazon S3 存储,以减少网络 I/O,同时将记录日志操作与恢复功能委托给存储节点.通过这种方式,Aurora 能够针对 SLA(service level agreement)提供更好的保障以及在计算层上对并发控制模块、锁模块等进行深度优化.但是,Aurora 运行在 Amazon 网络服务生态系统之上,外界开发人员无法对日志即数据架构带来的好处进行细致考察.我们自底向上构建 LogStore,并且验证日志即数据设计理念能够在普通商用计算环境中获得良好性能.LogBase[12]是较早的一个单拷贝系统,由新加坡国立大学数据库团队研发.LogBase 在 HBase 的基础上发展而来,不同于 HBase,它只写入 redo 日志到 HDFS 以优化写性能.LogBase 的另一个贡献是开发了一个多版本的内存索引,提高了对日志的访问效率.LogBase 需要定期做合并操作,以更好地支持范围查询.合并操作对性能会产生较大的影响.目前,LogStore 也是采用类似的操作以支持范围查询.不久的将来,我们计划利用历史查询来指导合并操作,以降低开销.Hyder[13]将事务处理模块从存储层中独立出来,开发了一系列针对 SSD 的算法,目的是为了在 SSD 共享数据集群中更好地扩展数据库.LogStore 的目标是面向一般的商用机器.与我们的工作另外一个类似的系统是多伦多大学的 Query Fresh[14],该系统采用日志即数据的单拷贝架构,不仅考虑单机的性

能,同时也将备机的情况考虑在内,以获得良好的高可用行为.Query Fresh 对待重放操作只需将 redo 日志存储到相应的内存数组即可,因此重放操作在 Query Fresh 中的开销可以忽略不计.此外,针对典型 DBMS 中重放操作的串行化行为,Query Fresh 中不同的线程可以并行实施重放操作,最大限度地减少系统的不可用间隔.Query Fresh 最大的不足是它对硬件配置要求很高,需要在 Infiniband 以及非易失性内存的高端机器环境中运行.

总的来说,log-as-database 系统实现上存在两种实践.

1) 一种方式是定制底层文件系统,其向上层的执行引擎提供读写接口,实现简单的文件操作功能;同时,针对数据库管理系统的要求,实现高可用、时间点恢复等功能.国内互联网公司,例如腾讯、阿里巴巴等都采用这种技术路线;
2) 第 2 种方式是在已有的文件系统,例如 HDFS、S3 等上实现单拷贝系统.NUS 与 Amazon 采用的就是这种方法.

对比两种不同的技术路线或者量化分析底层文件系统(例如 HDFS)给系统带来的开销是很有意义的,这可以作为未来的一个研究方向.

- 复制

复制是分布式环境中最复杂的一个技术.复制策略可以被划分为两大类:主动复制[15]和被动复制[16,17].主动复制技术实现起来相对较为简单,因为它规定了事务提交操作的网络传输代价.主动复制的缺点在于它要求事务的操作序列必须是确定性的,这与现实环境的工作负载存在较大的冲突.因此,大多数工程实践采用被动复制策略,LogStore 采用类似 raft[18]的同步复制协议.

2 系统架构

LogStore 是一个多线程、无锁化、高性能的键值数据库系统,同时将高可用规范纳入系统设计准则.LogStore 将分片指派到特定线程,同时利用多版本并发控制,以无锁的方式消除读写冲突、写写冲突、上下文切换开销.我们充分利用日志即数据的设计理念优化 LogStore,以达到满意的效果.此外,为了方便读操作,LogStore 集成了轻量级缓冲区.系统定期执行合并操作以支持范围查询.因此,在 LogStore,数据分为两部分:排序、无序的两部分.图 1 所示为系统总体架构示意图.每个分片都配备一个队列以接收到达的写请求;利用 ART 树(adaptive radix tree)[19]对外存数据进行索引,以便快速定位顺序写入的数据;LogStore 内嵌了轻量级缓冲区以及复制模块,以高效地支持读操作以及系统高可用.

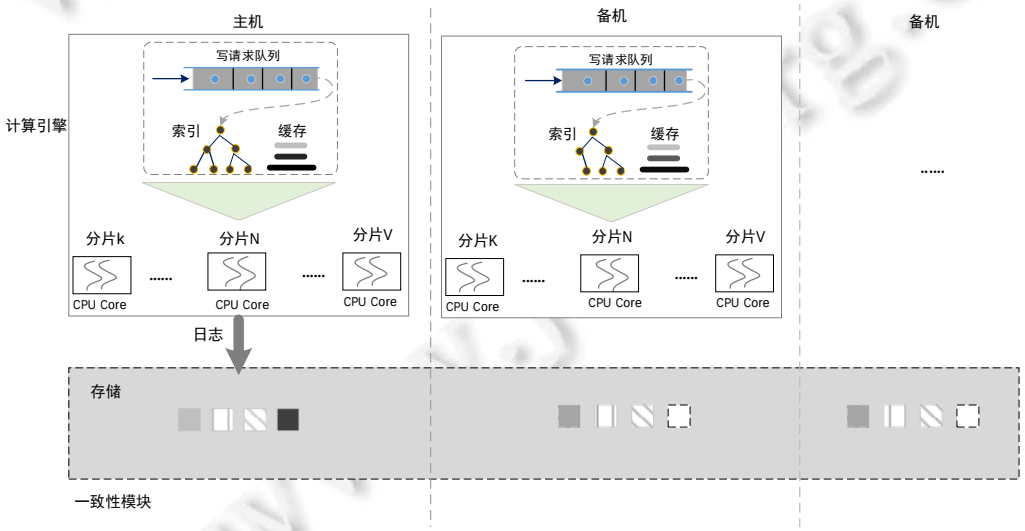


Fig.1 System overview

图 1 系统总体架构

- 队列:与 VoltDB[20,21]类似,系统采用 single-thread-per-partition 的执行模型消除写写冲突并发控制开销.当写请求到达时,它首先被放入分片对应的队列中等待调度;
- 索引:数据以顺序 I/O 的方式写入磁盘.我们利用 ART 树索引日志,能够以最多一次 I/O 的代价检索(点查询)请求的数据.ART 树是一个高效的内存索引,并且利用可变节点大小节省内存空间使用.最重要的是:如文献[22]所证明的,ART 树的性能大幅度优于典型的读优化的索引,例如高速缓存优化的 CSB+–tree[23];并且,ART 树支持更新操作.ART 树的一个不足之处是,对范围扫描支持不够友好.优化 ART 树的范围扫描特性,是我们未来工作的一部分;
- 存储:系统分为计算引擎与存储引擎,通过这种划分,我们可以单独优化不同的模块.类似于 Aurora,LogStore 只写日志到存储节点.这种设计有利于节省磁盘/网络带宽;而且我们还可以在这种架构上进行许多优化,例如:存储引擎可以将合并操作委托给不同的机器,计算引擎可以用较小的代价对索引、缓冲区进行单独的优化.

2.1 执行模型

多线程执行以及对同步进行精细的设计,一直被开发人员认为是高性能系统设计的不二法则.传统的 DBMS 将这个准则利用到极致.在这个设计理念的背后,底层的硬件设施起着决定性的作用.在 RDBMS 出现时期,计算机的内存与外设的访问速度差距达到 6 个数量级左右,而且内存的容量以 KB 来衡量.存储系统的性能瓶颈集中在 I/O 上.因此,高性能系统的一个要求是运行尽可能多的任务,期望多线程的复用可以隐藏 I/O 的延迟.存储系统的永恒主题是设计高效的缓冲区管理器、细粒度的锁机制以及高性能的闩锁和日志子系统.但是,现在的情况已经发生了巨大的变化.服务器通常配备数 GB 大小的内存以及多个处理核心的 CPU.很多工作负载,特别是 OLTP,都可以常驻内存.对于这类工作负载,尤其是 KV 操作,通常是搜索以及更新若干记录.这些操作本身并不耗时,但是日志模块、缓冲区、闩锁等占用了线程大部分执行时间.正如 Harizopoulos 等人[24]所验证的:在典型的 OLTP 工作负载中,线程花费在实际有效的工作时间只占执行总时间的 12%,其余时间都耗费在加锁、缓冲区管理等上面.基于这一个研究结论,LogStore 转而使用 single-thread-per-partition 的执行模型结合多版本并发控制,消除线程同步开销、读写冲突以及写写冲突.同时将线程绑定到特定的分片,以减少上下文切换开销.当读写请求到来时,该写请求被放入相应的分片的请求队列等待调度.任务执行期间以独占的方式执行,不能被抢占.

2.2 索 引

在 LogStore 中,日志文件被划分成两部分:一部分是有序的、不活跃段;另一部是无序的、活跃段.系统将日志记录顺序写入活跃文件段.无序的子文件会被定期合并到有序的、不活跃段以清除无效数据,同时为范围查询提供便利.ART 树索引建立在日志文件之上,以快速响应查询请求,如图 2 所示.

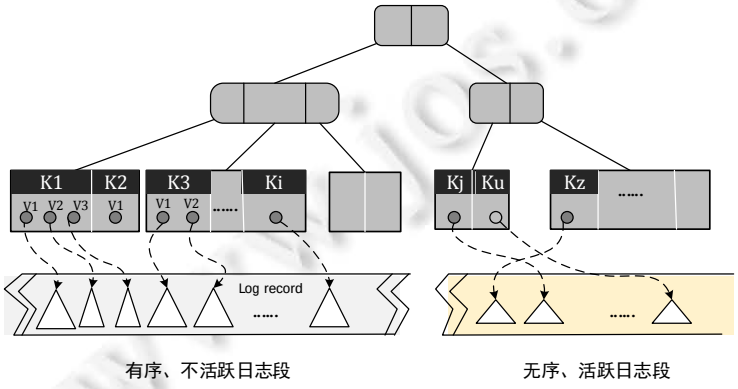


Fig.2 Index on log file

图 2 日志文件索引

叶子节点由形如下面的多元组构成:$\langle k_i,Version_1,Version_2,\ldots\rangle$.其中,*Version* 指向相同 *key* 的不同版本的 *value*.除此之外,*Version* 还维护另外两个字段实现 MVCC 的功能.在数据合并期间,这些过期版本的数据将会被清理.LogStore 中,对于每一个叶子节点的 *version* 域都包含 3 个字段,分别是指向相应的数据记录的 *offset* 以及用于判断版本的可见性的 *X*min 和 *X*max:

1. *X*min:在创建(***PUT*** 操作)键值对(key-value 对)时,记录该值为插入键值对的事务 ID;
2. *X*max:默认值为 0.在删除或者更新键值时记录此值.

LogStore 不会直接在旧 *value* 上修改,如何实现对这些并发操作进行隔离的呢?*X*max 字段具有两层含义.

- 其一是充当锁的作用,如果将该值设置为特殊的值,系统可以控制并发的更改操作.由于 LogStore 的单线程执行模型,这个特性没有发挥作用;
- 其二是限定可见的时间区间,读操作只会读取特定版本的数据,实现读写隔离.

简而言之,LogStore 判断 *value* 是否对一个 transaction id 为 *t_id* 的事务可见,遵循如下规则.

1. 对于 *version*@*X*min>*t_id* 的数据记录不可见;
2. 对于 *version*@*X*min<*t_id* & (*version*@*X*max=0 //*t_id*<*version*@*Xmax*)的键值并且已经提交的事务可见(已删除键值对除外).

系统的读写流程如下.

- *PUT* 请求

当一个创建请求到达时,系统将该请求放入相应分片的队列,同时放入对应的复制通道.执行线程不断地从队列中取出任务,然后执行操作.首先,系统获取插入时间戳 *ts*,将数据转换成相应的格式,令 *version*@*Xmin*:=*ts*.日志记录一旦写入文件,系统立刻返回该日志记录在活跃日志文件的偏移,以更新对应分片的内存索引(赋值 *version* 域的 *offset* 字段).后面的查询可以利用索引快速检索到该记录.如果系统中已经存在相应的 *key*,则为该 *key* 对应的 *value* 创建一个新版本.为减少系统与外设交互的频率,LogStore 也会将该日志记录加入到缓冲区.因此,读操作大部分可以从缓冲区获得相应数据而很少从磁盘检索数据.类似于 Aurora,LogStore 无需将缓冲区的数据写回存储层,而仅仅是利用缓冲区提高读性能.

- *GET* 操作

LogStore 支持 ***GET***(*key*)语义,其中,*key* 由用户指定.为了快速应答读请求,系统首先在缓冲区中查找是否存在对应的 *key*:如果存在,则读取对应的数据版本的 *value*,然后返回.具体而言,可见性规则如下:

$$X\text{min}\leqslant t\_id<X\text{max}\ \text{或者}\ t\_id\geqslant X\text{min}\ \&\ X\text{max}=0.$$

如果数据不在缓冲区内,否则需要访问 ART 树进行一次 I/O 操作读取相应的数据.缓冲区在慢 I/O 设备上对读操作的性能有着较大的影响,因此,一个轻量级、高效的缓冲区就显得比较重要.由于日志索引的存在,LogStore 可以在 SSD 上较好地应对长尾查询.LogStore 的缓冲区采用类似二次置换算法,但是系统无需维护数据的访问信息,从而将开销降至最低.大部分读请求能够在缓冲区中获得相应的数据.即使遇到不命中的情况,系统也能够以至多一次 I/O 的代价从外存读取相应的数据以响应点查询请求.

- *DELETE*

与插入操作相同,删除操作也需要进入对应分片的队列.为了删除一条记录,缓冲区与日志文件的数据都需要被删除.首先,系统会从缓冲区中移除该数据;然后写入一条记录,表示该 *key* 对应的数据已经被删除.同时,在该 *key* 在 ART 树中的索引条目对应地创建一个新版本的数据写入删除标记,置 *X*max 为–1.在数据合并时期,该记录所占用的物理空间才会被释放.

2.3 复制模块

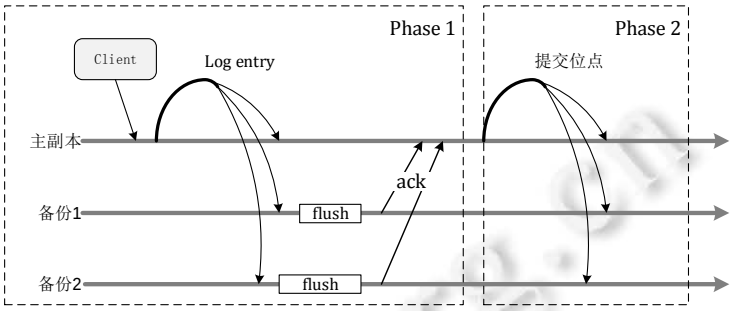
2.3.1 副本数据可见性

传统的 data+WAL 的双拷贝架构会引发额外的复制开销.下面是具体的例子,演示在传统的复制协议下双拷贝系统会面临的问题.以 raft 算法为例,复制流程分为两个阶段,如图 3 所示.

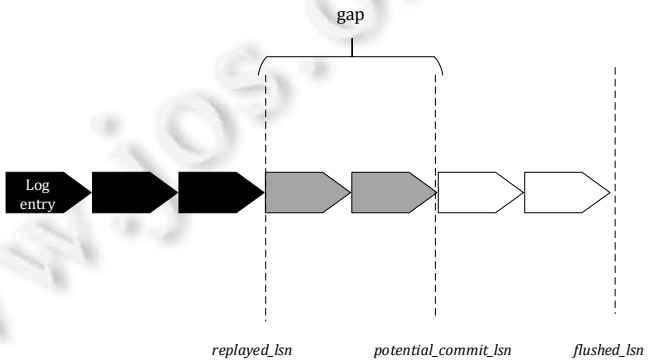
- 阶段 1:主机负责接收客户端请求,然后调用 *RPC* 接口 *AppendEntries*,将日志记录批量发送到备机.当主

机确认日志记录已被成功地在多数派备机上执行时,它即可以安全地提交这批事务,然后回复客户端;

- 阶段 2:主机首先更新本地的提交位点(commit point).在下一轮,主机附带上该提交位点,通知备机将提交位点推进该位置.备机需要再次从磁盘读入已经达成共识的日志记录,然后回放(一个优化措施是将最近的日志记录放入内存缓存,但是仍然需要多一次的内存拷贝).这意味着这一轮接收与刷盘的日志需要在下一轮再次被读入,回放.主从差距产生的根源即在于此流程.



(a) 数据复制流程



(b) 主从差距示意图

Fig.3 Data replication flow and illustrates the gap between primary and secondary

图 3 数据复制流程和主从差距的示意图

尽管主机可以使用批量提交以及流水线技术最小化 I/O 代价,但是备机面临的问题却是双拷贝架构无法避免的.备机将本轮接收的日志记录追加到文件末尾,但是这些新到来的日志记录即使在参与者之间达成共识,备机也无法马上进行回放操作,而是需要等待主机的提交位点.当只读请求被路由到备机时,它读到的数据很大可能是较早前的快照.如图 3(b)所示,备机存在 3 种状态的日志记录:第 1 种情况是,已经回放完成的日志记录,其描述的操作已经反映到数据存储;第 2 种情况是,已经确定在主机提交的日志记录但是提交位点尚未同步到备机,我们将这些日志记录的状态称为潜在提交;最后一种是已经刷盘但是尚未达成共识的日志记录.本文采用 3 个变量区分这 3 种状态的日志记录.

1) *replayed_lsn*:LSN 小于该变量的日志记录都属于已经回放完成的记录;
2) *potential_commit_lsn*:LSN 大于 *replayed_lsn* 但是小于等于该变量的日志记录都已经达成共识,但是尚未在备机回放;
3) *flushed_lsn*:LSN 大于 *potential_commit_lsn* 小于该变量的日志记录都尚未达成共识.

因此,当时间戳为 l 的读请求到达备机时,面临 3 种情况.

1) $l \leqslant$ *replayed_lsn*:这种情况下,读请求可以读到其所需的数据,然后马上返回;

2) *replayed_lsn*<*l*≤*potential_commit_lsn*:如果读请求的时间戳落入此区间,那么该请求需要被阻塞直到备机收到主机推送的提交位点,然后回放完成;

3) *l*>*potential_commit_lsn* 或 *l*≥*flushed_lsn*:这种情况下,该请求需要被丢弃.

在单拷贝系统下,可以对第 2 种情况进行优化.系统只需将 *replayed_lsn* 推进到 *potential_commit_lsn*,表示此区间日志已经回放完成,并且可以为查询提供服务,这几乎完全消除了图 3(b)所示的主从差距.当读取请求从元服务器获取最新的读取视图位点 *l* 后向备机请求数据,它可以将 *potential_commit_lsn* 与 *l* 进行比较.若 *l* 不大于 *potential_commit_lsn*,则读取请求可以利用内存中的索引来检索数据,而无需等待回放操作,提高了备机的可见性与数据库的响应速度.

2.3.2 复制技术

LogStore 复制算法类似于 raft 同步复制协议.我们在工程实现上优化了 raft 协议.最大的不同如图 4 所示.

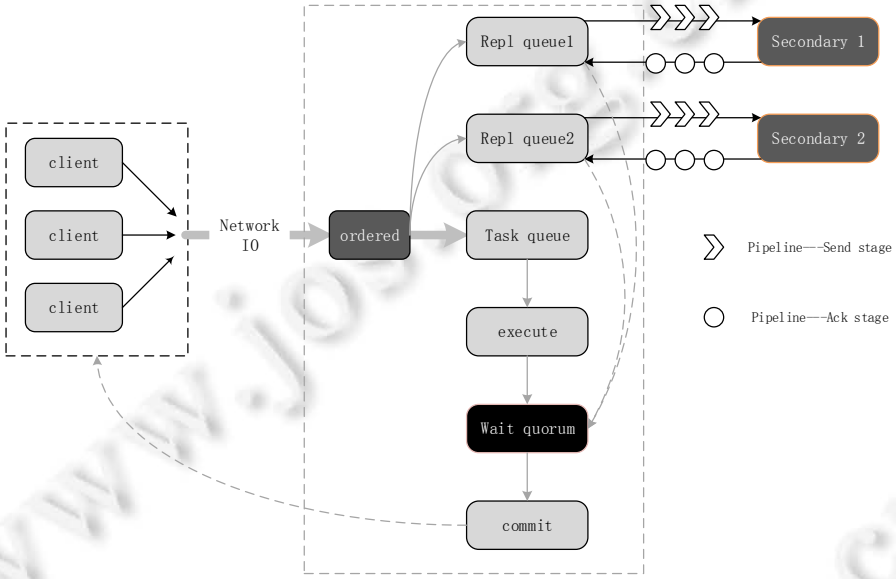


Fig.4 Replication architecture

图 4 复制算法总体架构

系统使用流水线技术,将复制分为请求发送阶段与 *ack* 接收阶段;同时,为了减少请求开销,主机将多个请求打包到一个任务.系统的数据复制流程如下.

(1) 主机负责将请求定序,执行本地提交并等待来自备机的确认.一旦收到客户端的请求,LogStore 就会将请求复制到备机.这一步骤与本地提交并行执行,以避免串行化等待.需要注意的是,主机和备机的存储引擎可以与计算引擎在不同的机器上部署;

(2) 复制的两个阶段,即数据发送和 *ack* 接收,可以同时进行;

(3) 当收到的 *ack* 达到多数派,主机立即将结果返回给客户端.

此外,我们从工程的角度优化 LogStore 复制模块,例如合并 I/O 请求减少系统调用、使用无锁数据结构等.当提供只读请求时,LogStore 的单拷贝架构可以使得副本能够提供所需的数据可见性(这主要得益于 LogStore 无需重放日志来产生最新的快照).复制的具体算法见下面各个子算法的相关描述.下面给出算法使用到的符号以及含义.

- $\mathcal{M}_p$:第 *p* 个分片维护的修改请求集合;
- u_p^i:针对第 *p* 个分片的第 *i* 个修改操作;

- $\mathcal{R}_p$:第 p 个分片的复制集合;
- $\mathcal{C}_p$、$\mathcal{C}_{p\_repl}$、$\mathcal{C}_{p\_ack}$、$\mathcal{C}_{p\_r}$:分别是第 p 个分片已完成的修改操作集合、复制集合、*ack* 等待集合与回复集合;
- $\mathcal{U}_p$:第 p 个分片的批量修改操作;
- $state_p^{i-k+1} \leftarrow applicator(u, state_p^{i-k})$:在重放函数 *applicator* 的作用下,第 p 个分片的状态从 $state_p^{i-k}$ 迁移到 $state_p^{i-k+1}$;
- $\rho(\mathcal{U}_p)$:将请求打包,然后使用一个系统调用将其发送到其他节点;
- $\mathcal{A}_p^u$ 针对第 p 个分片的修改操作 u 的 *ack* 集合;
- $\mathbb{A}_p$:第 p 个分片一批修改操作的 *ack* 集合;
- r_p:第 p 个分片所有修改操作的待回复集合;
- τ_p^u:等待回复的第 p 个分片的修改操作 u;
- τ_p:等待回复的第 p 个分片上某一批修改操作 u.

Replication Algorithm. Replicate log record to other storage nodes.

Input: File descriptor *fd*;

1. **Procedure *RequestDispatch***(*fd*) /*dispatch a task to certain channel*/
2. **while** *LogStore.IsRunning* **do**
3. $\mathcal{M}_p \leftarrow \mathcal{M}_p \cup \{u_p^i\}$ /*receive *i*th request for *p*th partition*/
4. $\mathcal{R}_p \leftarrow \mathcal{R}_p \cup \{u_p^i\}$ /*put *i*th request into *p*th channel*/
5. increment LSN of replication channel

Input: Set of *p*th partition requests $\mathcal{M}_p$;

1. **Procedure *ProcessTask***($\mathcal{M}_p$) /*process one batch of task*/
2. **while** $\mathcal{M}_p - \mathcal{C}_p \neq \varnothing$ **do**
3. $\mathcal{U}_p \leftarrow \varnothing$
4. $\mathcal{U}_p \leftarrow$pick a batch of requests from $\mathcal{M}_p - \mathcal{C}_p$
5. $\mathcal{C}_p \leftarrow \mathcal{C}_p \cup \mathcal{U}_p$
6. **for each** u in $\mathcal{U}_p$ **do** /*processing a batch size *k*/
7. $state_p^{i-k+1} \leftarrow applicator(u, state_p^{i-k})$
8. $r_p \leftarrow r_p \cup \{\tau_p^u\}$ /*add *u*'s reply obj τ_p^u into waiting reply set r_p*/
9. add ack into *u*'s ack set $\mathcal{A}_p^u$ in *p*th replication channel /*local ack*/

Input: Set of *p*th replication requests $\mathcal{R}_p$;

1. **Procedure *replicateSendStage***($\mathcal{R}_p$)
2. **while** $\mathcal{R}_p - \mathcal{C}_{p\_repl} \neq \varnothing$ **do**
3. $\mathcal{U}_p \leftarrow$pick a batch of requests from $\{\mathcal{R}_p - \mathcal{C}_{p\_repl}\}$
4. $\mathcal{C}_{p\_repl} \leftarrow \mathcal{C}_{p\_repl} \cup \mathcal{U}_p$
5. send $\rho(\mathcal{U}_p)$ /*pack modification messages and send by one sys call*/
6. **foreach** u in $\mathcal{U}_p$ **do**
7. $\mathbb{A}_p \leftarrow \mathbb{A}_p \cup \mathcal{A}_p^u$ /*add *u*'s ack set $\mathcal{A}_p^u$ into waiting ack set $\mathbb{A}_p$*/

Input: Set of *p*th partition acknowledge $\mathbb{A}_p$;

1. **Procedure *ReplicationRecvAndAckStage***($\mathbb{A}_p$)

2. **while** $\mathbb{A}_p - \mathcal{C}_{p\_ack} \neq \varnothing$ **do**
3. $\mathbb{A}_p^i \leftarrow$ pick a batch of waiting ack from $\{\mathbb{A}_p - \mathcal{C}_{p-ack}\}$
4. $\mathcal{C}_{p\_ack} \leftarrow \mathcal{C}_{p\_ack} \cup \mathbb{A}_p^i$
5. **for each** $\mathcal{A}_p^u$ in $\mathbb{A}_p^i$ **do**
6. add ack into $\mathcal{A}_p^u$ /*receive an ack for *u*'s ack set $\mathcal{A}_p^u$ */

Input: Set of *p*th partition reply r_p;

1. **Procedure *ReplyProcess***(r_p)
2. **while** $r_p - \mathcal{C}_{p\_r} \neq \varnothing$ **do**
3. $\tau_p \leftarrow$pick a batch of waiting reply from $r_p - \mathcal{C}_{p\_r}$
4. $\mathcal{C}_{p\_r} \leftarrow \tau_p \cup \mathcal{C}_{p\_r}$
5. **for each** u's reply obj τ_p^u in τ_p in **do**
6. fetch *u*'s ack set $\mathcal{A}_p^u$ by τ_p^u
7. **if** $|\mathcal{A}_p^u| \geqslant N/2+1$ **do**
8. send reply to client

LogStore 由如下 4 个子服务组成:*tcpServer*、*replicationServer*、*replyServer*、*stateMachServer*.*tcpServer* 负责接收客户端请求并对其进行参数检查:如果请求合法,且该请求是修改操作,则将其放入对应分片的任务集合 M_p 以及对应的复制集 R_p(*RequestDispatch*,#3-#4).*stateMachServer* 从队列获取一批任务进行处理(*ProcessTask*,#4),然后利用 *applicator* 改变系统的状态(*ProcessTask*,#7).当任务成功地在多数派备机上执行时,则主机会将该状态持久化.为了追踪该任务可否回复给客户端,*stateMachServer* 针对每一个修改操作 *u* 都维护一个对象 τ_p^u,收集来自其他节点的 *ack*.接着,将该对象加入第 *p* 个分片的等待回复集合 r_p.函数 *ProcessTask* 的最后一个步骤是将操作 *u* 的本地 *ack* 加入到第 *p* 个分片复制通道.在 LogStore 中,复制任务(*ReplicateSendStage*)与收集回复(*ReplicateRecvAndAckStage*)这两个函数运行在不同的通道,互不干扰.*ReplicationServer* 从复制任务队列获取一批任务 R_p,发送给不同的备机(*ReplicateSendStage*,#3-#4).然后将每一个修改操作 *u* 的 *ack* 集合 $\mathcal{A}_p^u$ 加入对应分片 *ack* 集合 $\mathbb{A}_p^i$. 同时,*ReplicationRecvAndAckStage* 收集来自备机的回复,并将其加入对应操作 *u* 的 *ack* 集合 $\mathcal{A}_p^u$(*ReplicationRecvAndAckStage*,#6).*replyServer* 不停地检查函数 *processTask* 产生的回复对象(*replyProcess*,#5),如果对应的 *ack* 集合的基数满足大于多数派要求,那么该操作可以成功回复客户端(*replyProcess*,#7-#8).

3 实 验

3.1 实验配置

实验在阿里云集群上进行,配置如下:3 台机器,每台机器有 16 个核心处理器,32GB 物理内存,600GB SSD 容量,默认通过千兆以太网连接.但是由于阿里云允许的带宽,我们只能使用 1/3 网络带宽.从 github 下载的 LogBase(由 NUS 数据库团队开发)的源代码无法正常运行,我们修复了这些错误.问题如下.

1) LogBase 与 HDFS 以及客户端不匹配.在 NUS 发布的版本中,LogBase 使用 HDFS(v 0.20.2),这个配置与客户端不兼容.尝试后,我们用版本 0.20.205 替换客户端和 HDFS;
2) *CreateFileNum* 函数的错误实现,代码片段 Int ret=(int) *System.currentTimeMillis*(·)/100 会导致溢出并出现逻辑错误;
3) 每个操作都要咨询 *meta*,导致性能问题.我们将缓存添加到客户端以绕过此问题;
4) ***GET***(byte [·] *key*)API 将遍历大于 *key* 的所有键值,不符合语义;
5) LogBase 将其日志写入 HDFS,并再次写入 HLog(WAL for HBase),即写入两次日志.我们修复了这个问

题,使得 LogBase 只写一次日志.

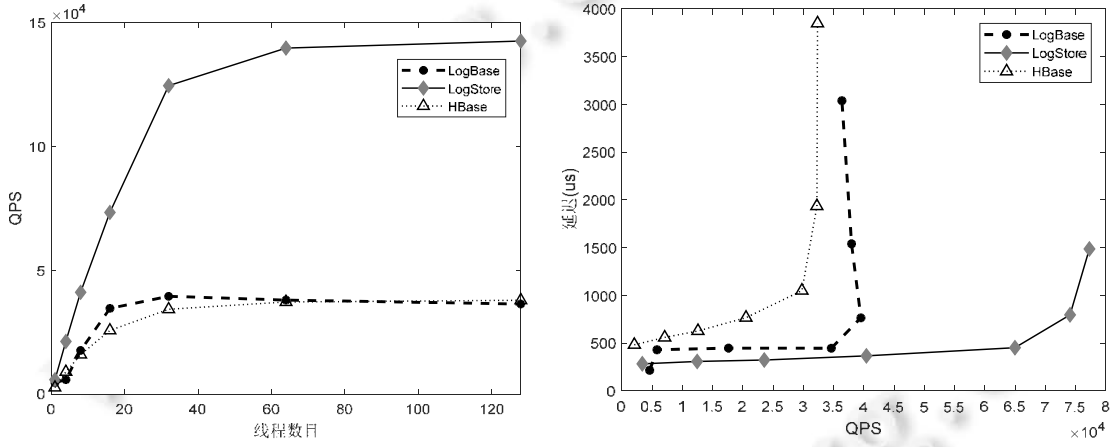
我们将修改后的 LogBase 发布在 github,https://github.com/logkv/logbase_changed 上.

LogStore 采用 C++实现,大约 1 万行代码.对于内存索引,系统使用 ART Tree.所有读操作将首先检查缓冲区.如果没有找到,则通过索引确定数据在文件中的位置.数据复制功能嵌入到 LogStore 中,因此,我们可以细致地评估日志即数据的设计理念带来的好处.在实验中,使用 HBase(版本 2.1.0)和 HDFS(版本 3.1.1).HBase 配置的内存缓存为 8GB.对于 HDFS,所有设置都保持默认,特别是块大小为 64MB,复制因子为 3.实验中使用的数据由 YCSB 生成.数据记录的大小约 1KB,*key* 的范围是$(0,2\times10^9)$.我们在每次实验前预热系统.

3.2 基准测试

3.2.1 写性能

写性能测试实验每次插入 100 万条数据记录.图 5(a)比较了 LogBase、LogStore 和 HBase 在单机模式下多线程的性能.这种设置使我们可以排除不相关的因素来研究 LogStore.线程数设置为 1、4、8、16、32、64、128.在 128 个线程的并发下,LogStore 的性能优于 LogBase 和 HBase 有 4 倍以上.另外,如图 5(a)所示,LogStore 在低于 32 个并发线程时可以达到线性扩展(大于 32 个线程以后未能达到线性扩展的原因,主要是受限于 CPU 核数).这主要得益于并发控制模型,单线程执行加多版本并发控制,消除了同步开销.图 5(b)给出了复制打开时 3 个系统的性能,以考察并发执行如何影响(线程的增加方式参照图 5(a))系统 QPS 与延迟之间的关系.



(a) 不同线程下的写性能 (b) 不同 QPS 下系统的延迟

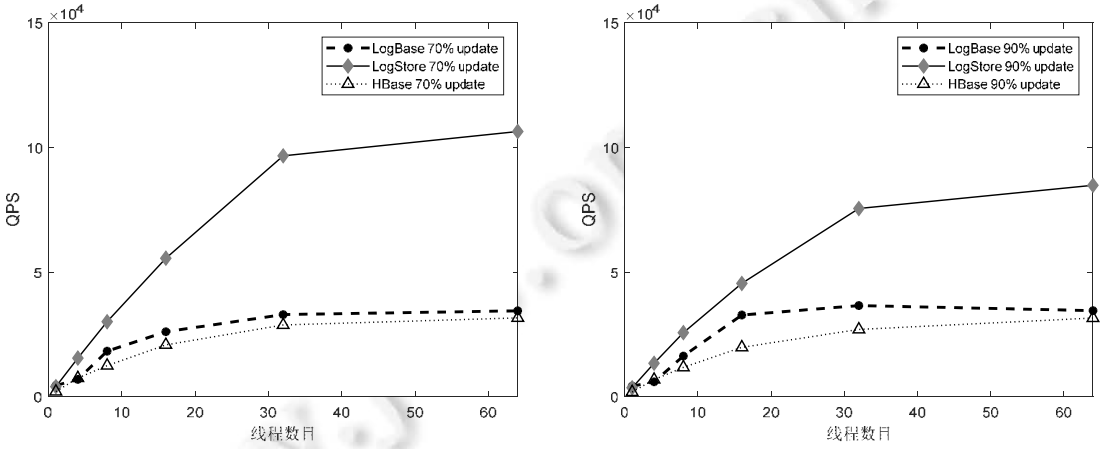
Fig.5 Write performance under different threas and relationship about latency and QPS

图 5 不同线程下的写性能和不同 QPS 下系统的延迟

实验观察到:当并发线程没有达到物理 CPU 资源限制时,QPS 随着工作线程的增加而增加,线程的增加对系统的延迟影响较小;当物理 CPU 资源达到上限时再增加线程对系统的吞吐量影响不大,但是延迟会急剧增加.这主要是并发线程对系统资源的激烈竞争,例如锁、CPU 而导致的.这个现象对于 HBase、LogBase 以及 LogStore 都是类似的.但是,LogBase 的锁对于并发处理不友好,因此延迟容易抖动.实际上,在图 5(b)所示的实验中:在单线程执行的情形下,LogBase 的 QPS 为 4 691,延迟仅为 211μs;当 4 线程并发执行时,此刻系统的 QPS 为 6 501,延迟跃升到 430μs,在图 5(b)观察到明显的抖动;当 8 线程并发执行时,系统的 QPS 为 17 833,延迟为 441μs;当 16 线程并发执行时,此刻系统 QPS 为 34 618,延迟为 436μs.此时已经达到了 CPU 处理核数的上限,再增加并发线程会急剧增加查询延迟.因此,32 线程并行执行时,系统 QPS 增加不明显,仅为 39 513,但是延迟却成倍飙升到 764μs.由于 LogStore 的无锁化设计以及单线程执行模型,当并发执行数为 64 以上时才会对延迟造成较大影响,此时系统 QPS 为 64 973.LogStore、LogBase 以及 HBase 在复制打开的情况下,QPS 峰值分别约为 80K、37K

和 32K.数据复制操作对 LogBase 的性能几乎没有影响.这主要是因为 LogBase 采用异步提交,即:在将数据安全地复制到其他存储节点上之前,LogBase 可以将结果返回给客户端.另一方面,LogStore 和 HBase 的吞吐量有不同程度的下降:HBase 的吞吐量下降约 12%;复制打开时,LogStore 需要将数据复制到 3 个不同的节点上,达到多数派的回复以后才可以提交,此时,系统吞吐量是单机模式下的 70%左右.实际上,在复制情况下,HBase 的延迟是 LogStore 的 2 倍.LogStore 的复制算法在延迟和吞吐量之间取得了较好的平衡.

此外,我们测试了混合工作负载下 3 个系统的性能.如图 6(a)和图 6(b)所示,LogBase 在读操作比重增加的情况下性能略有下降.这主要是由于 LogBase 牺牲安全性的异步复制以及内存索引对于读操作的支持不够高效(下面的读操作实验验证了这一点).LogStore 以及 HBase 在读写比例为 1:9 以及 3:7 这两个混合工作负载下,约有 10%的性能提升.



(a) 读写比例为 3:7 (b) 读写比例为 1:9

Fig.6 System performance under read-write ratio 3:7 and 1:9

图 6 读写比例为 3:7、1:9 时,不同系统的性能

我们还对双拷贝架构如何影响写入性能进行了测试.如图 7 所示,HBase 的 *memstore* 分别设置为 32MB、64MB 和 128MB.当 *memstore* 写满时,HBase 必须将 *memstore* 写到持久存储中,这会产生更多的写入开销,影响系统性能.这个双拷贝架构使 HBase 的写入性能受到很大影响.在使用 32MB *memstore* 的情况下,HBase 的写入性能仅为使用 128MB *memstore* 的 50%.LogStore 和 LogBase 只写一份数据,因此它们的写性能不会受内存大小的影响.

Fig.7 How size of MemStore affects system perfomance

图 7 MemStore 大小对系统性能的影响

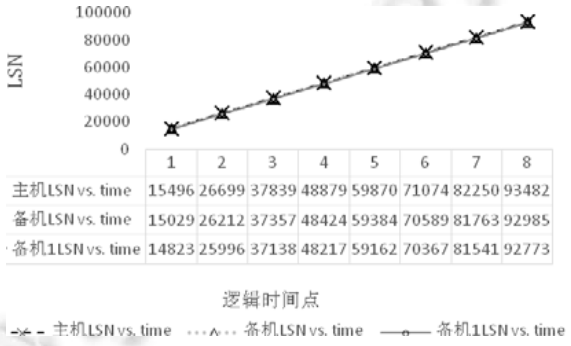
3.2.2 系统恢复

- 数据可见度

本节介绍复制功能.我们专注于主从差距的探讨.备份数据可见度可以通过如文献[14]中给出的公式来测量,其定义如下:

备份数据可见度=bLSN/pLSN,

其中,bLSN 指备份上的 LSN,pLSN 是指主机上的 LSN.文献[14]同步每个节点上的时钟,并且,每 20ms 读取 LSN.例如,在某个时间 *t*,主机的 LSN 为 1 000,备机的 LSN 为 800,备份数据可见度为 0.8.同样,我们探测主从机器上的(time,LSN)对.我们从主机以及备机上各选取 8 个点,然后在坐标轴上绘制相应的数据可见度曲线.由于 LogBase 以及 HBase 在单集群下并没有备机这个概念,因此也就没有主备日志滞后的概念.系统行为表现的度量标准是读写延迟.WAL 日志与文件不断地写入 HDFS,做 3 副本存储.系统周期性地删除过期的 WAL 日志.类似关系数据库中的 checkpoint.只有系统崩溃了才需要读取日志进行恢复操作;而搭建 HBase 主备集群又只能做到最终一致性.在这个系统拓扑下,双拷贝架构给系统带来的开销已经不是占主导的因素.因此,我们没有对 LogBase 和 HBase 两个系统进行主备数据可见性实验.图 8 给出了 LogStore 主备之间的数据延迟.由于单拷贝的设计使得备机无需重放日志,日志刷盘位点 bLSN 就可以度量数据可见度.如图 8 所示,备机数据可见度约为 95%.如果主机崩溃,备机几乎可以立刻接管主机,然后对外提供服务.因此,我们认为 LogStore 具有良好的高可用规范.



	1	2	3	4	5	6	7	8
主机LSN vs. time	15496	26699	37839	48879	59870	71074	82250	93482
备机LSN vs. time	15029	26212	37357	48424	59384	70589	81763	92985
备机1LSN vs. time	14823	25996	37138	48217	59162	70367	81541	92773

Fig.8 Gap between the primary and secondaries

图 8 主备之间的差距

- 恢复时间

现在我们研究 LogBase 和 LogStore 这两种单拷贝系统的恢复表现.利用生成快照成本(将内存中索引写入持久存储)和恢复时间两个维度度量系统恢复功能.对于单拷贝系统架构而言,产生快照成本与恢复时间成正比(系统无需重放操作),因此我们只需测试恢复时间.对于 LogBase 和 LogStore,恢复成本与要读取的日志记录量密切相关,因为它们不需要应用日志记录来生成最新的数据文件.我们利用 *fork* 系统调用为 LogStore 生成快照.如图 9 所示,LogStore 在系统恢复方面的性能令人满意.LogBase 在重新加载快照和恢复时间上的成本几乎是 LogStore 的 10 倍.因此,LogBase 创建检查点成本比 LogStore 高 10 倍.这主要是因为 LogStore 的内存索引结构相当节省空间,因此可以大幅度地减少生成快照时间.在数据集大小为 4GB 时,内存索引只有 16MB 大小.可变大小的树节点以及只在叶子节点存储多版本信息是节省内存的关键.当数据大小超过 64G 时,LogBase 需要 1 小时才能生成快照.因此,我们没有给出 LogBase 在数据大小超过 64GB 后的恢复时间.以上实验在单个分区上进行.可以认为:如果在分区之间并行执行恢复,LogStore 可以在 1 分钟内完成数据大小为 TB 级别的系统恢复.

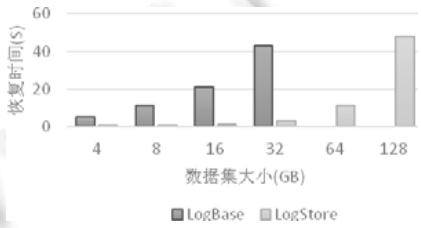


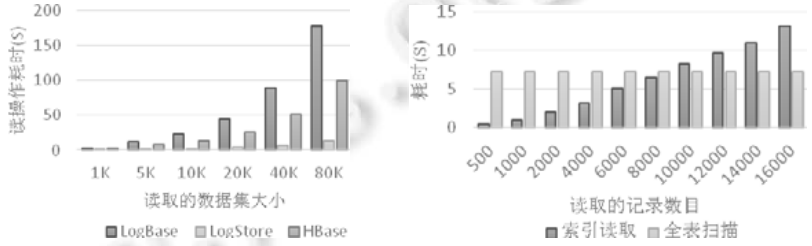
Fig.9 Recovery time under different data volume

图 9 不同数据大小下的系统恢复时间

3.2.3 读性能

本节验证索引设计对系统读性能的影响.实验对比 LogStore、LogBase 和 HBase 这 3 个系统的随机访问的性能.数据集大小为100万.数据读取大小的范围为1K~80K.3个系统的总时间由网络延迟和执行时间组成.一旦随机访问请求完成,YCSB 即将结果返回给客户端,即请求的数据记录总量等于客户端与服务器交互的网络次数.LogStore 和 LogBase 可以利用内存索引来精确定位记录的位置.另一方面,HBase 需要使用全表扫描的方法来检索数据(布隆过滤器可以减少需要扫描的数据块).从图 10(a)所示结果可以看出:HBase 的性能要优于 LogBase,因为布隆过滤器大幅度减少了数据块的扫描.得益于 ART 树的高效以及一次 IO 模型,LogStore 的读性能要比其他两个系统优 10 倍左右.实际上,LogStore 与理想性能更接近.在实验中观察到的 LogStore 的 TPS 约为 5 700,但网络延迟为 156us,因此理想性能约为 6 400.

单拷贝系统中读操作大部分都通过索引进行,因此,调研索引访问的成本相当有必要.这可以为此类单拷贝系统的查询优化提供很好的建议.为了验证索引对系统性能的影响,我们修改了 YCSB 基准,允许它随机获得一批记录,然后再将结果返回给客户端.索引访问的批处理大小范围从 1K~16K.如图 10(b)所示:当获取的数据记录总量小于 8K 时,索引访问更有利.但当数据大小超过 8K 时,顺序访问优于索引访问.我们将这个简单的启发式方法集成到 LogStore 中,当访问数据部分大于 1/10 时,启动表扫描方法.



(a) 读性能测试 (b) 索引访问开销

Fig.10 Read performance and index access cost

图 10 读性能测试和索引访问开销

4 总结与展望

本文主要介绍 LogStore KV 存储系统的设计.该系统遵循日志即数据的设计,其中,所有持久数据仅存储在日志文件中,统一数据与日志的存在方式.这种新颖的存储架构将内存数据结构与其持久表示解耦,消除了传统 WAL 数据系统设计的很多开销.与传统的三阶段恢复设计(即扫描,重做,撤消)相比,单拷贝系统还极大地提高了恢复能力.由于日志索引包含了最新一致状态下的数据项所需的精确信息,因此从系统恢复只需重建索引.此外,与大多数 DBMS 采用的方法不同,该架构不需要设置检查点、写回脏数据以及强制执行 WAL.通过将最先进的数据管理方法结合在一起,诸如快速内存访问、LSM 数据组织和缓冲区管理等,LogStore 在现代内存数据库系统和传统的基于磁盘的系统之间达到性能与成本的一个最优点.在这种设计下,LogStore 具备线性扩展能力,在配置足够的情况下,每秒可达百万次的一致性读/写操作.同时,LogStore 中的读/写延迟限制在 20ms 以内.此外,其高可用性规范令人满意,即主备延迟可忽略不计.另一方面,LogStore 对数据进行分片,当出现数据偏斜的情况时,会造成系统资源使用严重不均衡.系统的单线程执行模型对复杂操作支持不够,因此,LogStore 不适合分析型工作负载.目前,LogStore 也尚不支持跨分片分布式事务.增加负载均衡模块以及支持跨分片事务,留待未来加以研究.

实际上,日志即文件的设计是 LogStore 此类系统的优点.但在某种程度上而言,这也是单拷贝系统不足之处产生的根源.单拷贝系统将数据按照时间先后顺序写入磁盘,没有维护数据的有序性,失去了传统数据库系统聚簇存储的优点,对读操作不友好,特别是范围读取操作.因此,隔一段时间之后,系统不可避免地需要进行合并操作,对 *key* 排序.这个过程相当耗费资源.利用机器学习为数据集动态选择最优的数据组织方式,减少这个过程是

未来的一个研究方向[25].此外,为了弥补单拷贝系统数据的无序性,加速数据查找通常会在日志文件上增加一个索引.这个索引一般不做持久化操作,因此系统崩溃恢复的时候需要重建索引.如何快速地重建索引,是一个不小的挑战.对内存使用要求严格的使用场景(例如嵌入式),索引节点也有可能不被全部放入内存,这会增加额外的 IO 操作.对内存和时延要求严格的场景如何优化 log-as-database 系统,是一个待探索的方向.

References:

[1] Sears R, Ramakrishnan R. BLSM: A general purpose log structured merge tree. In: Candan KS, Chen Y, Snodgrass RT, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 2012). Scottsdale: ACM, 2012. 217−228. [doi: 10.1145/2213836.2213862]

[2] Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. on Database System, 1992,17(1):94−162. [doi: 10.1145/128765.128770]

[3] Stonebraker M. The land sharks are on the squawk box. ACM Communication, 2016,59(2):74−83. [doi: 10.1145/2869958]

[4] HBase. Open source implementation of HBase. http://hadoop.apache.org/hbase/

[5] 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):4:1−4:26. [doi: 10.1145/1365815.1365816]

[6] Facebook. RocksDB. 2016. http://rocksdb.org

[7] http://leveldb.org

[8] https://github.com/google/leveldb

[9] Chandramouli B, Prasaad G, Kossmann D, Levandoski JJ, Hunter J, Barnett M. FASTER: A concurrent key-value store with in-place updates. In: Das G, Jermaine CM, Bernstein PA, eds, Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Houston: ACM, 2018. 275−290. [doi: 10.1145/3183713.3196898]

[10] Sheehy J, Smith D. Bitcask: A log-structured hash table for fast key/value data. White Paper, Basho Technologies, 2010.

[11] Verbitski A, Gupta A, Saha D, Brahmadesam M, Gupta K, Mittal R, Krishnamurthy S, Maurice S, Kharatishvili T, Bao XF. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: Salihoglu S, Zhou W, Chirkova R, Yang J, Suciu D, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1041−1052. [doi: 10.1145/3035918.3056101]

[12] Vo HT, Wang S, Agrawal D, Chen G, Ooi BC. LogBase: A scalable log-structured database system in the cloud. PVLDB, 2012, 5(10):1004−1015. [doi: 10.14778/2336664.2336673]

[13] Bernstein PA, Reid CW, Das S. Hyder—A transactional record manager for shared flash. In: Proc. of the CIDR. Asilomar: www.cidrdb.org, 2011. 9−20.

[14] Wang T, Johnson R, Pandis I. Query fresh: Log shipping on steroids. PVLDB, 2017,11(4):406−417. [doi: 10.1145/3186728.3164137]

[15] Schneider FB. Distributed Systems. 2nd ed., Boston: Addison-Wesley, 1993. 18−41.

[16] Budhiraja N, Marzullo K, Schneider FB, Toueg S. Distributed Systems. 2nd ed., Boston: Addison-Wesley, 1993. 199−216.

[17] Guerraoui R, Schiper A. Software-based replication for fault tolerance. IEEE Computer, 1997,30(4):68−74. [doi: 10.1109/2.585156]

[18] Ongaro D, Ousterhout JK. In search of an understandable consensus algorithm. In: Gibson G, Zeldovich N, eds. Proc. of the USENIX Annual Technical Conf. Philadelphia: USENIX Association, 2014. 305−319.

[19] Leis V, Kemper A, Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases. In: Jensen CS, Jermaine CM, Zhou X, eds, Proc. of the 29th IEEE Int'l Conf. on Data Engineering (ICDE). Brisbane: IEEE Computer Society, 2013. 38−49. [doi: 10.1109/ICDE.2013.6544812]

[20] Kallman R, Kimura H, Natkins J, Pavlo A, Rasin A, Zdonik S, Jones EPC, Madden S, Stonebraker M, Zhang Y, Hugg J, Abadi DJ. H-Store: A high-performance, distributed main memory transaction processing system. PVLDB, 2008,1(2):1496−1499. [doi: 10.14778/1454159.1454211]

[21] VoltDB. http://voltdb.com

[22] Wang Z, Pavlo A, Lim H, Leis V, Zhang H, Kaminsky M, Andersen D. Building a BW-tree takes more than just buzz words. In: Das G, Jermaine CM, Bernstein PA, eds. Proc. of the ACM SIGMOD Int'l Conf. on Managment of Data. Houston: ACM, 2018. 473–488. [doi: 10.1145/3183713.3196895]

[23] Rao J, Ross K. Making B+-trees cache conscious in main memory. In: Chen W, Naughton JF, Bernstein PA, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Texas: ACM, 2000. 475–486. [doi: 10.1145/342009.335449]

[24] Harizopoulos S, Abadi DJ, Madden S, Stonebraker M. OLTP through the looking glass, and what we found there. In: Proc. of the SIGMOD. 2008. 981–992.

[25] Idreos S, Dayan N, Qin W, Akmanalp M, Hilgard S, Ross A, Lennon J, Jain V, Gupta H, Li D, Zhu Z. Design continuums and the path toward self-designing key-value stores that know and learn. In: Proc. of the CIDR. Asilomar: www.cidrdb.org, 2019.

朱阅岸(1983－),男,博士,工程师,主要研究领域为高性能数据库系统,OLTP,新硬件数据库技术.

王树(1979－),男,工程师,主要研究领域为分布式系统.

简怀兵(1980－),男,工程师,主要研究领域为分布式系统.

吴喜亮(1988－),男,硕士,主要研究领域为信号处理,计算机.

龙永超(1982－),男,学士,主要研究领域为高性能数据库系统,分布式系统.

钟治初(1964－),男,副教授,CCF 高级会员,主要研究领域为软件工程,编程语言.

李彬(1988－),男,工程师,主要研究领域为分布式系统.

张延松(1973－),男,博士,副教授,主要研究领域为内存数据库,OLAP,新硬件数据库技术,GPU 数据库.