文章编号: 1000-5641(2019)05-0143-16

面向日志结构化数据存储的高效数据加载

丁国浩, 徐 辰, 钱卫宁

(华东师范大学 数据科学与工程学院, 上海 200062)

摘要: 近年来, 随着互联网技术的快速发展, 无论是互联网企业还是传统的金融机构, 用户量和业务处理数据量都在快速地增长. 传统的通过增加服务器并采用基于分库分表的方法来解决扩展性问题, 需要大量的人工维护成本和硬件开销. 为降低开销和分库分表带来的各种问题, 业界通常用新型数据库系统替换原有的系统, 其中, 基于日志结构合并树存储的数据库系统(如OceanBase)被广泛采用, 这类系统磁盘上存储数据块呈现全局有序的特征. 在从传统数据库切换到新型数据库过程中, 需要将大量数据加载到新数据库系统中, 长时间加载的过程中可能出现数据库节点宕机. 为了减少总加载时间和故障恢复时间, 提出了一种负载均衡且支持高效容错的数据加载方法; 为了支持负载均衡的数据加载, 与预确定分区划分数据的方法不同, 考虑到目标系统默认存储块大小, 采用通过基于文件大小和目标系统默认存储块大小预计算分区数目, 并利用分库分表的数据导出往往已经排序的特点, 采用选取部分采样块和等间隔选取样本的方式确定分区之间的切分点, 避免了全局采样和随机或头部样本选取方式确定切分点带来的高开销; 为了加快故障恢复速度, 利用日志结构合并树存储系统的多备份减少故障恢复时的数据量, 提出了基本副本局部故障恢复方式, 避免了完全重新加载的故障恢复方式. 实验结果表明, 相比采用预确定分区数目和全局选取采样块的随机或头部选取样本方法, 采用预计算分区数目和部分选取采样块的等间隔选取样本确定切分点的方法, 提高了数据加载的性能, 并且验证了基于副本局部故障恢复方法相比完全重启加载恢复方法的高效性.

关键词: 数据加载; 负载均衡; 容错; 日志结构

中图分类号: TP392 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2019.05.012

Efficient data loading for log-structured data stores

DING Guo-hao, XU Chen, QIAN Wei-ning

(*School of Data Science and Engineering*, *East China Normal University*, *Shanghai* 200062, *China*)

Abstract: With the rapid development of Internet technology in recent years, the number of users and the data processed by Internet companies and traditional financial institutions are growing rapidly. Traditionally, businesses have tackled this scalability problem by adding servers and adopting methods based on database sharding; however,

收稿日期: 2019-07-28

基金项目: 国家重点研发计划(2018YFB1003400); 上海市青年科技英才扬帆计划(19YF1414200)

第一作者: 丁国浩, 男, 硕士研究生, 研究方向为分布式数据管理系统.
E-mail: guohao.ding@outlook.com.

通信作者: 徐 辰, 男, 副教授, 研究方向为大规模分布式数据管理系统.
E-mail: cxu@dase.ecnu.edu.cn.

this can lead to significant manual maintenance expenses and hardware overhead. To reduce overhead and the problems caused by database sharding, businesses commonly replace heritage equipment with new database systems. In this context, new databases based on log-structured merge tree storage (such as OceanBase) are being widely used; the data blocks stored on the disks of such systems exhibit global orderly features. In the process of switching from a traditional database to a new database, a large amount of data must be transferred, and database node downtime may occur when there is extended loading. In order to reduce the total time for loading and failure recovery, we propose a data loading method that supports load balancing and efficient fault tolerance. To support balanced data loading, we pre-calculate the number of partitions based on the file size and the default block size of a target system rather than using a pre-determined number of partitions. In addition, we use the feature that the data exported from the sharding database is usually sorted to determine the split points between partitions by selecting partial sampling blocks and selecting samples in sampling blocks at equal intervals, avoiding the high overhead caused by global sampling and selecting samples randomly or at the head in sampling blocks. To speed up the recovery process, we propose a replica-based partial recovery to avoid restart-based complete reloading; this method uses the multi-replica of an LSM-tree system to reduce the amount of reloaded data. Experimental results show that by pre-calculating the number of partitions and partial sampling blocks and by using equal-interval sampling, we can accelerate data loading relative to pre-determining the number of partitions and global sampling blocks as well as relative to random or head sampling strategies. Hence, we demonstrate the efficiency of replica-based partial failure recovery compared to restart-based complete reloading.

Keywords: data loading; load balance; fault tolerance; log-structured

0 引 言

随着互联网技术的快速发展, 无论是互联网企业还是传统的金融机构, 用户量和业务处理的数据量都在快速地增长. 尤其在金融行业, 通常这些企业需要大型的数据库系统存储和管理海量的数据, 绝大多数的金融行业都采用“IBM 小型机+DB2/Oracle 数据库”来构建其业务的底层存储系统, 利用高端存储阵列和高可靠的数据库软件来保证业务的高可靠性和高可用性. 但随着业务处理的数据量不断增长, 无论是数据量和访问量, 单台的数据库管理系统都无法满足需求. 在原有架构下, 一般根据业务的特点对数据库进行拆分, 将数据水平切分存储在不同的数据库服务器上; 而客户端通过数据库中间层将不同的请求分发到不同的数据库服务器上, 这种方式需要通过不断增加服务器来满足不断增长的业务和数据量增长, 需要大量的人工维护成本和硬件开销. 因此, 很多企业开始把眼光转向具有良好扩展性和容错性的分布式数据库存储系统, 如已经有一些银行系统采用基于日志结构合并(Log-Structured Merge , LSM)树[1]的分布式数据库替换原有的 DB2 或 Oracle. LSM 因为具有良好的写入性能而被 NoSQL 系统和 NewSQL 系统广泛采用. 许多流行的系统, 如 BigTable [2]、LevelDB [3]、Hbase [4]、OceanBase[5]和TiDB[6]等, 都是利用LSM树或其变体来作为存储结构.

本文主要研究在利用基于这种 LSM 树存储的新型系统替换传统数据库系统时涉及的

一个关键问题, 即数据加载. 在这种场景中, 需要将大量的数据从一个系统加载到一个新的系统中. 为了减少切换系统的时间, 并行化数据加载是一种常见的加速数据加载方式, 在并行化加载任务中, 总加载时间取决于执行时间最长的加载任务, 所以需要一种负载均衡的加载方法, 同时在加载过程中能够支持容错处理.

为了实现负载均衡的加载, Silberstein 等提出了一种向 PNUTS[7]系统的范围分区表中批量插入数据的方法[8]. 该方法增加了一个额外计划阶段用于收集加载数据的统计信息, 以确保批量插入数据时尽可能均衡; 但是, 它需要对整个数据集进行全局采样来确定划分分区的边界, 当数据量很大时, 这种采样方法的开销相对较高. 在该工作的扩展[9]中, Hadoop[10]被用于在将数据插入到 PNUTS 前进行排序, 但并没有优化如何采样实现一个负载均衡的加载; 在工作中能够在加载时支持故障处理[9], 但不能处理目标系统存储节点故障. 另外, 一些存储系统也不能很好支持加载过程中容错处理, 如 Cassandra[11], 如果目标系统的存储节点出现故障, 整个加载任务就会失败.

本文工作的目标是为分布式LSM 树存储系统上提供一种负载均衡且支持高效容错的数据加载方法, 从而减少总加载时间和故障恢复时间: ① 本文采用一种基于加载文件大小和目标系统默认存储块大小预计算分区数目, 而不是根据目标存储系统节点数预确定分区数目. 因为预确定分区数目过少可能导致每个分区处理的数据量过大, 在存储系统内部会进行分裂为小的分区, 产生额外的 I/O(Input/Output) 开销, 另外, 如果预确定分区数目过多, 由于每个分区处理任务的初始化开销也并不能带来并行处理性能的提升. ② 本文利用数据集从多个数据库中导出呈现局部有序的特征, 通过基于部分选取采样块和等间隔选取样本的方式确定分区之间的切分点, 而不是采用全局采样和其他选取样本方式, 从而减少了采样开销. 在部分采样方法中, 通过略增大分区数目在采样开销和采样精确度之间进行权衡. ③ 本文利用 LSM 树存储系统多副本的特性, 采用一种基本副本局部故障恢复方式而不是采用基于重启全局故障恢复方式, 来减少从数据源重新加载恢复的数据量, 加快故障恢复速度. 本文的实验结果表明, 采用部分选取采样块和等间隔选取样本的数据加载方法, 相比全局选取采样块和等间隔选取样本的方法[7]总加载时间减少 20% 左右. 本文的主要贡献总结如下.

(1) 提出了一种通过预计算分区数目和基于部分采样确定分区间切分点的负载均衡的加载方法.

(2) 提出了一种基于副本的局部故障恢复的加载方法, 减少了故障恢复的时间.

(3) 基于 Hadoop[10] 和开源数据库 Cedar[12]的实验验证了本文提出的负载均衡和容错数据加载方法的高效性.

本文其余部分安排如下: 第 1 节说明基于分布式LSM树存储系统模型和常用的数据加载方法概述; 第 2 节描述负载均衡的数据加载方法的细节; 第 3 节介绍加载过程中的容错处理; 第 4 节对本文提出的数据加载方法进行实验验证; 第 5 节介绍相关工作; 第 6 节总结全文.

1　概　　述

1.1　系统模型

日志结构合并(LSM)树是一种专门为频繁写入而优化的数据结构. 它主要包括两个部分: 一部分是内存中树形结构; 另一部分是存储在磁盘中的多个不可修改的树形结构.

采用 LSM 树的分布式存储系统的典型实现如图 1 所示. 它由 4 种主要类型的节点组成, 每种节点类型用于执行特定的一组事务并且每种类型节点都是可扩展的: 事务节点

(T-node)存在 1 个主节点, 只有主节点用于接收读(Read)/写(Write)请求; 主控节点(C-node)也存在 1 个主节点, 主节点提供服务; 存储节点(S-node)和查询节点(Q-node)都是一种 share nothing 架构, S-node主要用于存储表在磁盘上的存储结构 SSTable, Q-node主要用于合并 T-node 和 S-node 上的查询结果. 这种设计分离了系统中各个组件关注的点并简化了整个系统的复杂性. 写入 LSM 树等于插入到主 T-node 的内存存储 Memtable 中. 写操作到达客户端时, 它首先将其 redo 日志条目刷到磁盘中, 然后将其修改应用到 Memtable 中; 当内存存储 Memtable 的大小达到设定的阈值时, 其存储的内容将刷新到磁盘; 然后, 新的内存存储 Memtable 接受修改, 并且在经过许多次数据合并之后将生成多个磁盘存储. 系统中的数据表被分区存储到多个存储节点; 每个分区由一系列主键[start key, end key]表示; 每个存储节点将有数百个分区, 通过将各个分区从负载过高的节点移动到负载相对比较低的节点, 进行细粒度的负载平衡. 读操作首先向协调者节点询问数据位置, 然后合并磁盘存储器和内存存储器中的相应数据以进行响应.

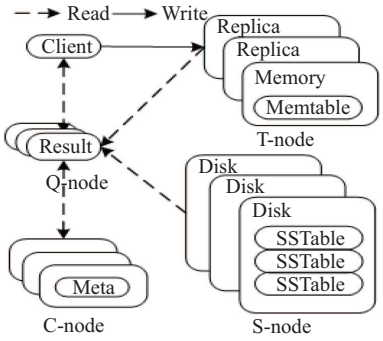


图 1 LSM 树的实现

Fig. 1 Implementation of an LSM-tree

1.2 加载方法概述

1.2.1 基于 SQL 的加载方法

应用程序将数据加载到数据库中最简单方法是使用 Insert 语句一次插入一行. 该技术便于加载少量数据. 传统的商业数据库都支持这种加载数据的方式. 但是, 与更复杂的技术相比, 这种方法比较慢, 并且不适合将大量数据加载到数据库中. 为了进一步提高基于 SQL 的加载的性能, 一些批量加载方法被提出, 即可以使用一条 SQL Insert 语句一次插入多组值, 然后数据库遍历所提供的数据, 并为每组执行一次 SQL Insert 语句. 批量操作比重复单行操作更有效, 因为数据库只解析一次语句, 其加载工具通常使用多线程并发执行来提高数据加载效率. 但是, 执行 SQL Insert 方法会占用大量 CPU 处理时间, 而且高并发数据加载会增加数据库服务器上的负载, 这反过来会导致增加其他应用程序访问数据库的延迟.

1.2.2 基于内存表的加载方法

对于大多数数据库系统, 当使用 Insert 语句将记录插入数据库时, 通常需要几个步骤, 如 SQL 语法解析、生成逻辑和物理计划、查询优化、事务处理、写日志等. 在传统的集中式数据库中, 这些操作在单个节点上执行. 但是, 在分布式数据库中, 这些操作被分成在不同类型的节点上处理, 这些不同类型的节点需要通过网络进行通信, 因此插入语句通常需要经过多次网络传输. 为了减少网络传输并避免 SQL 语句解析, 将数据直接转换为数据库内存表结构 Memtable. Memtable 是用于接受读/写请求的事务节点的内存中的数据结构. 尽管这

种方法在一定程度上加速了数据加载, 但当内存达到临界值时, 仍然需要一段时间才能将内存数据刷新到磁盘.

1.2.3　基于文件的加载方法

另一种方法是将原始数据文件直接转换为存储系统最终磁盘上的存储格式. 将传统数据库中数据迁移到新型基于 LSM 树的存储系统中, 是将传统数据库中导出的局部有序的数据文件转换为新型系统磁盘上存储格式, 即 SSTable files 的集合, 然后将转换后的文件加载到新型 LSM 树系统中. 整个基于文件的加载过程主要分为以下 5 个阶段, 如图 2 所示(数据库集群只画出了加载过程中涉及的相关节点).

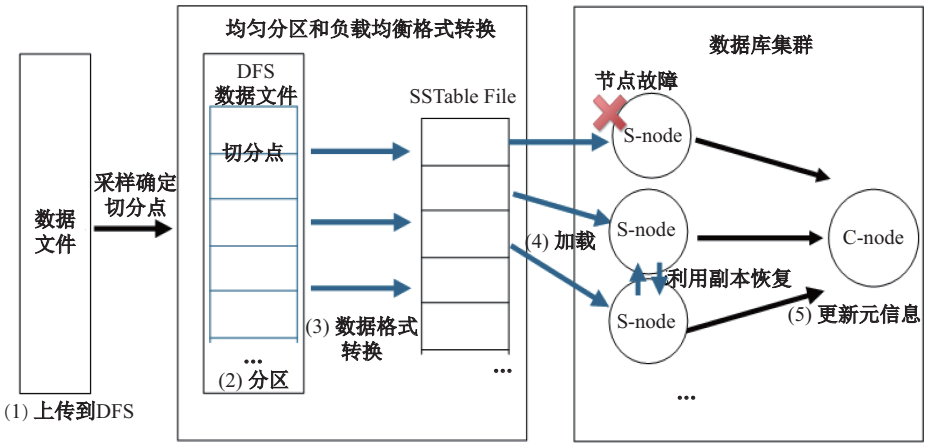


图 2　基于文件的加载过程

Fig. 2　File-based loading process

(1) 上传到分布式文件系统(Distributed File System, DFS): 当数据不在分布式文件系统上时, 需要将数据上传到分布式文件系统中, 假设数据已经在分布式文件系统中.

(2) 分区: 将输入文件拆分为多个分区, 如何确定分区数目和分区之间的切分点是影响负载均衡格式转换的重要因素; 划分分区后可以并行格式化转换并将格式化后分区并行加载到目标存储系统中. 本文第 2 节将讨论如何进行数据分区.

(3) 数据格式转换: 将文本数据转换为数据库的内部存储格式.

(4) 加载: 将转换后的数据文件迁移到数据库的存储节点. 加载阶段可能会出现多种故障类型, 这里主要讨论存储节点故障. 本文第 3 节将详细讨论如何进行故障处理.

(5) 更新元信息: 将已加载的数据报告给协调(主)节点, 然后将这部分数据提供给外部.

2　负载均衡的数据加载

在基于 LSM 树的存储系统中, 要求数据最终是按照全局有序的顺序排列存储在多个存储节点上, 所以基于文件的加载方法首先要对整个数据集进行排序, 并且划分一个个有序的范围, 每个范围对应一个底层存储 SSTable 文件. 为了产生一个全局有序的文件, 简单的方法是将所有数据划分在一个分区中, 在单个分区内对所有数据进行排序; 但该方法在处理大数据量的文件时效率比较低, 因为 1 台机器必须处理所有输出文件, 从而完全丧失了并行性. 还有一种方法是在多台机器上创建一系列排好序的文件, 然后串联这些文件, 最后即为全局排序的文件. 分布式排序是大数据处理系统(如 Hadoop)的典型应用场景. 使用大数据处理系

统进行分布式排序的关键点, 是如何对数据进行分区, 因为分区的不均衡直接影响着接下来数据格式转换和加载阶段的性能. 为了实现均衡的数据加载, 本文主要讨论如何进行数据分区. 在下文中, 首先讨论如何确定分区数目, 然后研究如何确定划分分区的边界点.

2.1 分区数目

分区数目是影响数据加载性能的一个重要因素. 确定分区数目一种方法是使分区数目等于目标系统的存储节点数[13], 分区的数目取决于目标系统存储节点的数量. 也就是说, 当存储节点的数量恒定时, 分区的数目是固定的, 不随加载数据量的变化而变化, 这种方法称为***预确定分区数目***. 但是, 当表中的数据量很大时, 此方法存在一些缺点: 一方面, 因为在基于 LSM 树的存储系统中, 一般系统内部会按照一定的大小将数据划分为一个个有序的存储块, 默认的存储块大小(blockSize)为 256 MB, 当存储块超过默认大小时, 合并阶段会进行分裂, 这会增加系统 I/O 的开销, 而且合并阶段一般无法对外提供服务, 对系统的可用性也造成一定影响; 另一方面, 分区数目决定了数据格式转换阶段和加载阶段的最大并行度, 分区数目少并行度也比较低, 然而, 分区数目也不是越多越好, 因为每个分区数据处理任务的初始化都会消耗一定的时间和资源, 过多的分区数目会导致额外的初始化开销.

本文采用一种自适应文件大小的分区数目设置方法, 即考虑不同数据文件的大小和目标存储系统默认存储块大小, 分区数目用公式

$$\text{numOfPartition} = \left\lceil \frac{\text{fileSize}}{\text{blockSize}} \right\rceil, \tag{1}$$

来近似估算, 本文把这种方法称为***预计算分区数目***. 式(1)中, fileSize 为加载的数据文件总大小, blockSize 为目标存储系统的默认存储块大小. 分区数目近似估算为两者相除向上取整, 以使理想情况下每个分区处理的数据量小于等于blockSize大小. 在这种情况下, 每个分区处理的数据量可以大致相等, 在数据格式转换和加载阶段实现相对均衡的数据处理和加载. 例如, 1 GB 的数据文件, blockSize 为 256 MB, 可以通过预计算分区最终划分 4 个分区, 使每个分区处理的数据量小于等于 256 MB.

确定分区数目后, 需要进一步确定数据划分方法. 数据划分方法决定了数据文件中每个数据记录最终划分到哪个分区处理. 因为分区并行数据处理任务的完成时间取决于并行处理阶段执行最慢的分区任务, 所以分区之间的负载均衡影响着分区并行数据格式转换和加载阶段的性能.

2.2 确定分区边界点

本节主要讨论如何确定划分分区之间的边界点, 以使各个分区处理的数据量相对比较均衡. 一种简单的方法是对整个数据集进行均匀等间隔切分, 比如将[1,1000)范围的数据划分 5 个分区, 则[1,200)在 partitio1, [200,400)在 partition2, [400,600)在 partition3, [600,800)在 partition4, [800,1000)在 partition5. 在数据均匀分布的情况下, 可以使每个分区的数据量大致相等; 但通常数据可能不是均匀分布, 这就会导致分区之间的负载不均衡. 为了获取数据的分布情况, 需要对数据集进行采样, 通过采样确定切分点来将数据集切分为所设置的分区数. 接下来首先讨论如何确定采样的数据块数, 然后讨论在每个采样块中如何选取样本.

2.2.1 采样块数

大数据处理系统一般将数据划分为固定大小的块存储在分布式文件系统中(如 HDFS(Hadoop DFS)), 一种采样方法是对所有的数据块进行采样, 也就是对整个数据集进行采样[7], 本文把这种方法称为***全局采样***. 通过全局采样可以获得比较精确的切分点, 进

而使每个分区处理的数据量都相对比较均匀, 获得比较好的负载均衡. 然而, 全局采样的开销通常比较高, 本文采用一种选部分块进行采样的方法, 本文称这种方法为***部分采样***. 选取部分数据块采样会导致确定的切分点不是很精确, 进而导致每个分区实际处理的数据量不是很均匀, 即大小在 blockSize 左右波动. 在这种情况下, 导致有些分区处理的数据量比较多, 进而使该分区所在的节点执行数据格式转换时需要更长的执行时间, 在加载到存储系统系统中时, 可能还需要额外的 I/O 开销将大分区的数据进行分裂; 而有些分区处理的数据量比较少, 执行完数据格式转换任务需要等待执行时间长的任务完成. 为了使每个分区处理的数据量小于等于blockSize, 本文在第 2.1 节设置分区数目的基础上加上 1 个增量值 ε. 因此, 在部分采样下, 用公式

$$\text{numOfPartition} = \left\lceil \frac{\text{fileSize}}{\text{blockSize}} \right\rceil + \varepsilon \tag{2}$$

来近似估算分区数目. 式(2)中, ε是一个比较小的值, ε值的大小取决于采样的精确度.

采样的块数越多, 采样的精确度越高, 采样开销也越大, 所以采样的块数和采样的精确性之间存在一个权衡. 一般采样的块数越多, 最终确定的切分点越精确, 每个分区处理的数据也越均匀, 因此 ε 的取值就越小; 反之, 采样的块数越少, 确定的切分点相对就不是很精确, 就会导致每个分区处理的数据量不均匀, 出现数据量过大或过小的分区, 则可通过增大 ε 来增加分区的数目, 进而在一定程度来减少过大或过小分区的出现, 使每个分区处理的数据量小于等于 blockSize 大小. 分布式文件系统默认存储块大小为 dfsBlockSize(例如 HDFS 中默认块大小为 64 MB), 这里取近似采样块数, 公式为

$$\text{numOfSampleBlock} \approx \frac{\text{blockSize}}{\text{dfsBlockSize}}. \tag{3}$$

例如, 在图 3 中, 同样 1 GB 的数据文件在分布式文件系统 HDFS 上存储在 16 个不同的块中, 每个块大小(dfsBlockSize)为64 MB, blockSize 为 256 MB. 由于数据集部分有序的特性, 这里在多个存储块中选取 1 个存储块进行采样来估算数据分布, 所以可以选近似 4 个块(图 3 中虚线阴影块)进行采样来估算数据集的分布情况, 使大致 4 个块在 1 个分区中.

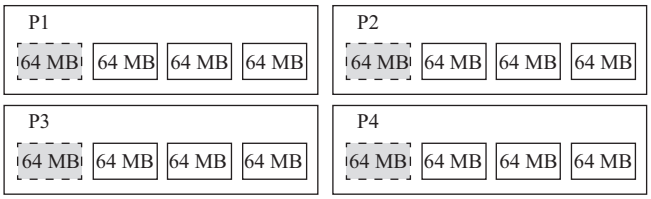


图 3　采样块数

Fig. 3　Number of sampling blocks

2.2.2　样本选取

确定采样块数后, 需要在每个采样块中选取样本, 选取样本的方式有以下 3 种.

(1) 等间隔选取(Interval): 按一定间隔从每个采样块中选择 key 值. 这更适合于局部有序的数据.

(2) 随机选取(Random): 随机从每个采样分块中选取 key 值. 这是一个通用的方法, 但效率通常比较低, 也可能导致负载不均衡.

(3) 头部选取(Split): 选择每个采样块的前 n 条记录作为样本. 这种方式虽然效率比较高但不适合有序的数据.

例如, 同样在上面的例子中, 需要在4个采样块中选取样本. 假设 1 条记录大小为 512 B 左右, 1 个采样块大约有 131 072 条记录, 每个采样块中按照 1% 的采样频率选取样本. 如图 4, 对于等间隔选取样本方法, 在每个采样块中每隔 100 个记录选取 1 个样本, 分布较均匀, 可以近似反应数据分布情况; 对于随机选取样本方法, 在每个采样块中样本选取是随机性的, 分布不均匀; 对于头部选取样本方法, 从头开始依次选择 n 条记录作为样本, 虽然选取样本的效率高, 但并不适合有序数据. 因此, 这里采用等间隔选取样本的方法. 最后, 在完成选取样本后, 需要对各个采样块中选取的所有样本进行排序, 根据分区数目计算切分点, 即生成的切分点数等于分区数目减 1.

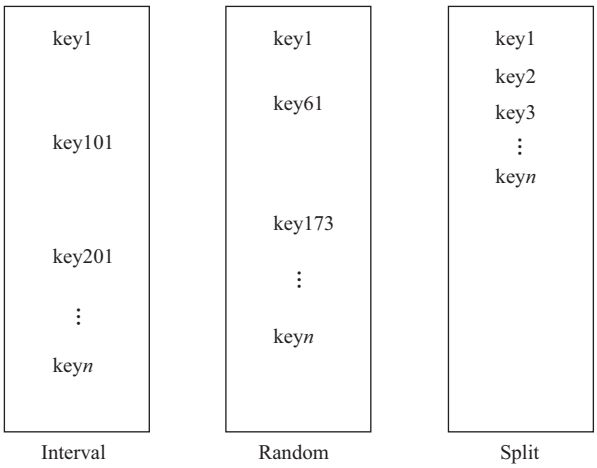


图 4 样本选取

Fig. 4 Sample selection

3 容错处理

到目前为止, 本文所描述的加载方法在没有出现故障的情况下可以很好地运行. 然而, 故障是大规模分布式存储系统中的常见现象. 在加载过程中会出现不同类型的故障, 可以将故障类型划分为软件故障和硬件故障.

对于软件故障, 本文继承了文献 [9] 中作者提出的几种故障处理方法. 例如, 1 条记录的主键与其他记录发生冲突, 本文提供了一种可选的执行模式, 即当检测到哪些记录导致确定性失败则跳过这些记录让程序继续执行. 对于硬件故障, 很少有研究来讨论这种类型的故障, 并且一些系统提供的数据加载工具也无法很好地处理这种类型的故障, 例如 Cassandra[11]. 如果加载过程中存储节点发生故障, 则需要重启加载任务来进行故障恢复, 本文把这种方法称为***基于重启全局故障恢复***. 接下来, 主要详细介绍如何处理硬件故障.

由于网络或磁盘故障导致的物理存储节点崩溃是硬件故障的一个典型情况, 而且大多数这种类型的故障都是比较严重的. 在正常执行数据加载期间, 数据被加载到一组存储节点上, 并且这些节点中的一些节点可能出现故障. 当存储节点发生故障时, 允许故障节点上已经加载的数据, 以及当前正在运行的所有加载任务并行地在其他节点上进行恢复. 然而, 考虑到 LSM 树存储系统中多副本的特性, 如果故障节点上已经加载的数据在其他节点上已经存在副本, 可以利用其他节点上的副本进行恢复, 而不需要重启加载任务从数据源进行重新加载, 这样就减少了故障恢复的时间. 本文把这种方法称为***基于副本局部故障恢复***.

在下文中, 主要以单个存储节点故障情况下为例, 详细讨论基于重启全局故障恢复方式和基于副本局部故障恢复方式.

3.1　基于重启全局故障恢复

在加载的过程中, 当检测到某个存储节点发生故障, 并不是利用新的可用节点替换故障节点, 而是利用集群中的健康节点来并行恢复故障节点上的加载任务. 算法 1 描述了加载过程的故障处理: 系统定期获取集群中可用存储节点的信息以及已经加载子表(tablet)信息(算法 1 中行 1 至行 2), 包括加载的 tablet 的数量和大小; 然后计算平均每个存活节点上已经加载的 tablet 的平均数量和大小, 在选择目标存储节点之前会根据这些信息来确定目标加载节点, 因此故障节点上的 tablet 信息不会被计算; 当系统检查每个 tablet 的副本数量是否满足系统要求时, 如果不能, 则将加载任务重新分配给其他节点. 此外, 根据集群中计算的每个节点的工作负载, 可以将负载任务均匀地分布到负载较低的节点, 可以在不同的存储节点上同时启动多个加载任务, 允许整个集群参与恢复.

算法 1　故障发现和恢复

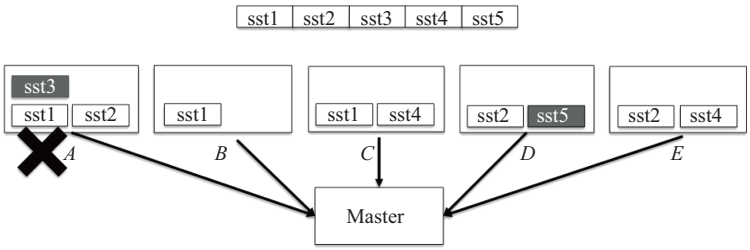
```
1: 获取集群的节点状态信息
2: 获取加载表的各个子 tablet 的加载状态
3: for 每个加载失败的子 tablet do
4:     重新分配加载任务到其他节点
5: end for
```

为了说明这种恢复方式, 以图 5 为例加以说明. 图 5(a)为正常加载过程中, 需要将 5 个分区数据(sst1-sst5)加载到含有5个存储节点(节点 $A-E$)的目标存储系统中,其中阴影块 sst3 和 sst5 表示在集群中还未存在副本, 而 sst1、sst2 和 sst4 在集群的其他节点上已经存在副本. 为了被认为是存活的状态, 每个存储节点都需要定期向主(Master)节点发送心跳(Heartbeats). 此外, 每个存储节点会将已经加载的分区向主节点汇报. 主节点检查每个分区是否满足所需数量的安全副本(默认 3 副本), 如果不满足, 则根据集群的负载将加载任务重新分配给其他存储节点. 在正常运行情况下, 为了实现负载均衡, 按一定顺序为每个分区选择目标存储节点. 在图 5(a)中, 假设存储节点 A 发生故障, 主节点会通过心跳机制检测到该节点故障并且认为故障节点上已经加载和正在加载的分区丢失(sst1、sst2 和 sst3). 图 5(b), 由于主节点记录了加载到每个存储节点上的失败的分区信息, 则会将原 A 节点上失败的加载任务重新分配给其他节点(节点B、C、D)并行恢复失败的加载任务.

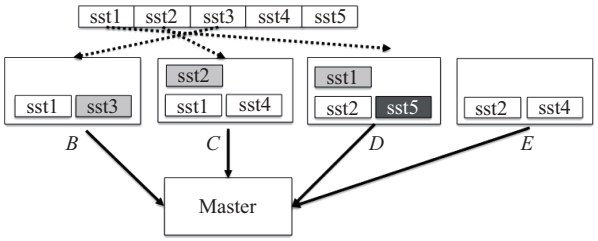
3.2　基于副本局部故障恢复

考虑到 LSM 树存储系统多副本的特性, 当故障节点上加载的数据在其他节点上已经存在副本, 则利用其他节点的副本进行故障恢复, 而不需要从数据源重新加载丢失的副本. 算法 2 描述了改进的恢复流程: 在加载过程中, 首先会检查每个分区是否满足所需的副本数量(算法 2 中行 2); 如果没有, 会根据集群的负载均衡情况选择目标存储节点(算法 2 中行 4); 当集群中的分区对应的副本数量为 0 时(即可以看作尚未开始加载第一个副本), 从数据源中选择加载分区(算法2中行 6); 当副本数量大于 0 时, 依靠目标系统的负载均衡机制从集群内的含有副本的存储节点中选择副本来复制(算法 2 中行 8). 同样在图 5(a)中节点 A 发生故障

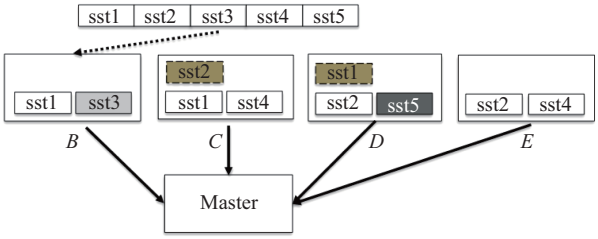
后, 在图 5(c)中, 因为 sst1 和 sst2 在其他节点上已经存在副本, 可以利用集群的负载均衡机制来恢复, 而 sst3 还未存在副本, 所以只需从数据源重新分配加载任务 sst3 给其他节点即可.



(a) **加载过程中故障**



(b) **基于重启全局故障恢复**



(c) **基于副本局部故障恢复**

图 5 故障恢复过程

Fig. 5 Recovery process

3.3 讨论

多个故障: 由于各种原因, 在同一时间遇到多个节点故障也是常见的. 一种用于修复多节点故障的简单方法是, 使用为修复单节点故障而设计的方法逐个修复多个故障节点, 但此方法效率不高. 可以逻辑地将多个节点故障视为逻辑上 1 个节点故障. 因此, 为了从多个节点故障中恢复, 类似地应用算法 2.

算法 2 基于副本的故障恢复

```
for 每个需要加载的分区 P do
    need_copy //检查分区 P 是否满足副本数
    if need_copy then
        选择存储节点
        if 分区 P 的副本数 =0 then
```

```
6:              从数据源选择加载
7:          else
8:              从其他副本选择复制
9:          end if
10:         添加新加载任务
11:     end if
12: end for
```

级联故障: 不同于节点同时发生故障的多个节点故障, 级联故障意味着当系统正在进行恢复时新的节点发生故障. 但是, 级联故障可视为连续单个节点故障. 因此, 连续调用算法 2 以从级联故障中恢复. 另外, 为了防止级联故障导致某些节点过载并影响系统的正常性能, 可以通过设置参数来限制可以并行加载的最大任务数.

4 实　　验

在本节中, 对本文提出的数据加载方法进行实验评估. 首先, 在第 4.1 节描述本文方法的实现; 然后, 在第 4.2 节描述本文的实验环境、部署和使用的数据集; 在第 4.3 节和第 4.4 节中分别验证本文提出的负载均衡加载方法和基于副本的故障恢复方法的高效性.

4.1 实现

本文的数据加载方法主要基于 Hadoop 大数据系统来实现, 并在基于LSM 树的存储系统 Cedar 上验证.

Hadoop 大数据系统的一个典型应用场景就是对数据集进行分布式排序. 在 Hadoop 大数据系统中分区的数目即为 Reduce 的任务数, 分区方法用于确定 Map 的中间输出结果发送给哪个 Reduce 处理. MapReduce[14]作业的运行时间取决于其 Map 和 Reduce 任务的总执行时间. 在 Map 阶段, Mappers 使用本地 HDFS 客户端接口并行读取本地数据. HDFS 将原始数据拆分为大小相等的存储块(默认为 64 MB, 即 dfsBlockSize 为 64 MB)存储在不同位置, 并提供负载均衡机制分配存储块到集群中的每个节点. Map 方法处理每个记录并生成一组中间的键/值对, 中间键/值对由分区方法确定发送给哪个 Reduce 方法处理, Reduce 方法生成最终的键/值对并将结果集写入到 HDFS 中. MapReduce 框架中默认使用基于哈希(Hash)的分区方法, 基于 Hash 的分区方法虽然可以实现很好的负载均衡, 但是在基于 LSM 树的存储系统中需要分区全局排序, 即每个分区不仅内部有序, 而且还维护分区之间的顺序. 所以需要自定义基于范围(Range)分区方法. MapReduce 框架本身提供了一种基于采样的全局排序的实现 TotalOrderPartitioner 及 3 种采样 API: RandomSampler、SplitSampler、IntervalSampler. 由于数据集有序的特性, 本文利用IntervalSampler方法并根据第 2 节自定义采样块数以减少全局采样的开销.

4.2 实验设置

实验环境即服务器配置如表 1 所示. 测试数据集为不同大小 (1 GB、5 GB、10 GB、20 GB)的 CSV 文件, CSV 文件中每行记录对应数据库中表的一条元组, 并且主键是有序的.

Cedar 数据库部署在 8 个节点上, 其中 1 个节点为主节点, 包含 1 个事务节点(T-node)和 1 个控制节点(即为协调节点, C-node), 其他 7 个节点分别包含 1 个存储节点(S-node)和查询节点(Q-node).

表 1 服务器配置

Tab. 1 Server setup

类型	描述
OS	CentOS release 6.5(Linux version 2.6.32)
CPU	2×Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00 GHz(6 cores/CPU)
Memory	165 GB
Network	Broadcom Corporation NetXtreme BCM5719 Gigabit Ethernet

Hadoop 集群部署在另外 10 个节点上, 其中 1 个节点为主节点, 包含 Name Node、Secondary Name Node 等, 其他 9 个节点为 Data Node 节点.

在负载均衡的加载实验中, 对于分区数目设置, 本文主要与采用预确定分区数目[13]的加载对比; 对于划分分区切分点, 本文主要与采用全局采样[7]的方法对比; 对于样本选取, 主要对比等间隔选取、随机选取和头部选取这 3 种方法; 在容错实验中, 本文主要与基于重启全局故障恢复方法[11]对比.

4.3 负载均衡

在本节中, 主要测试影响负载均衡数据加载的两个重要因素: 分区数目的设置和划分分区边界点的选择, 并观察它们对数据加载性能的影响.

4.3.1 分区数目对加载性能影响

主要测试分区数目对数据加载性能的影响. 首先测试在固定数据量(10 GB)的情况下, 分区数目设置对加载性能的影响; 然后测试在不同数据量(1 GB、5 GB、10 GB、20 GB)的情况下, 采用预确定分区数目和预计算分区数目对加载性能的影响. 为了使各个分区尽可能均匀, 划分分区边界点采用本文第 2.2 节的全局采样. 实验结果如图 6 所示.

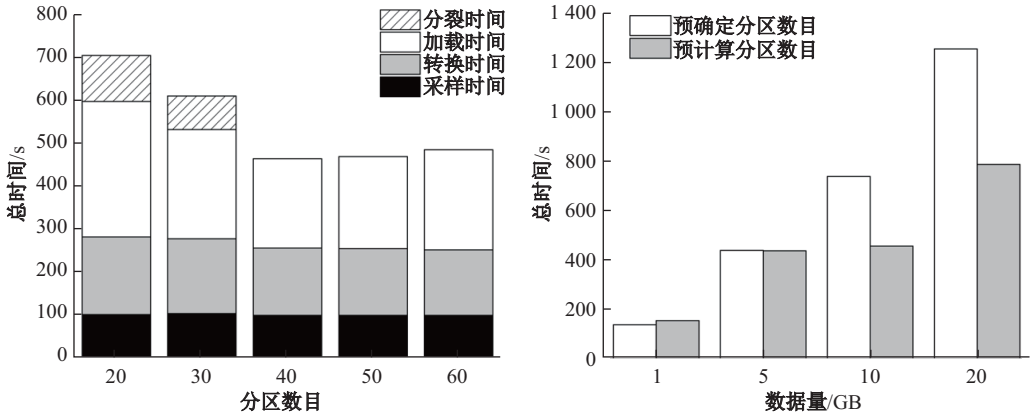


图 6 分区数目对加载性能影响

Fig. 6 Effect of the number of partitions on loading

从图 6 的左图可以看出, 当分区数目小于⌈fileSize/blockSize⌉, 即 40 时, 数据加载时间比较长. 一方面, 因为在分区格式转换阶段, 并行度比较低; 另一方面, 由于分区数目少, 每个分区的大小超过目标系统默认存储块大小, 会导致大的分区进行分裂, 产生额外的分裂开销. 当分区数目大于⌈fileSize/blockSize⌉时, 虽然提高了数据加载任务的并行度, 但由于每个分区加载任务的初始化开销, 也并没有带来很大的性能提升. 因此, 在分区数目为⌈fileSize/blockSize⌉时, 数据加载的性能相对于其他分区数目较优.

从图 6 的右图可以看出, 在不同数据量的情况下, 采用预确定分区数目的方法设置分区数目等于目标系统存储节点的数量(即 7), 而对于采用预计算分区数目的方法设置分区数目为 ⌈fileSize/blockSize⌉. 当数据量较小时, 如 1 GB, ⌈fileSize/blockSize⌉小于 7, 由于预确定分区方法的并行度更高, 所以预确定分区数目方法的加载性能优于预计算分区数目; 但随着数据量的增大, 相比采用预确定分区数目的加载方法, 预计算分区数目的加载方法由于并行度更高而获得更好的加载性能. 因此, 当加载数据量比较大时, 采用预计算分区数目的加载方法性能比采用预确定分区数目的方法更好.

4.3.2　划分切分点对加载性能的影响

在前面的实验中, 为了使各个分区尽可能比较均衡且分区大小小于等于目标存储系统的默认存储块大小, 采用了对数据源进行全局采样. 然而, 本文发现, 使用全局采样的开销比较大, 大约占总加载时间的 20%. 为了进一步减少采样的开销, 这个实验中, 主要测试部分采样对加载性能的影响. 由于部分采样可能导致分区的不均匀, 一些分区大小超过目标存储系统的默认存储块大小, 本文通过在原有分区数目设置的情况下略增大分区数目, 即加上 ε, 来使分区大小满足目标存储系统默认块大小要求. 实验结果如表 2 和图 7 所示.

表 2　不同采样比例下 ε 对加载性能影响

Tab. 2　Influence of ε on loading performance under different sampling ratios

采样方式	采样比例	ε				
		0	**1**	**2**	**3**	**4**
全局采样	1	518	**496**	497	497	499
部分采样	1/2	465	**455**	457	458	461
	1/3	457	453	**418**	429	430
	1/4	427	414	**413**	417	425
	1/5	483	478	461	445	**442**

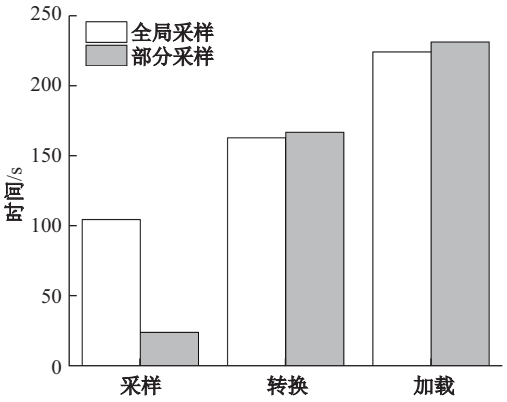


图 7　部分采样对加载性能影响

Fig. 7　Effect of partial sampling on loading

表 2 为在不同采样比例情况下 ε 取不同值时的总加载时间. 从表 2 中可以看出, 随着采样比例的减少, 由于减少了采样开销, 从而提高了加载的性能. 但实验中发现, 当采样比例减少到一定值时, 如 1/5, 已经很难得到一个相对精确的采样值, 通过增大 ε 值也很难使分区数据相对均衡且小于等于存储系统默认存储块大小. 此外, 对于一定的采样比例, 存在一个优化的 ε, 使得可以获得相对较优的加载性能.

图 7 为对应部分采样比例为 1/4 下最优的加载性能与全局采样方法(即采样比例为 1). 在

整个加载流程中 3 个阶段的时间, 包括采样时间、数据格式转换时间和最后加载时间的对比情况. 从图 7 中可以看出, 当采用全局采样时, 采样开销较高; 当采用部分采样时, 采样开销明显降低. 虽然在采用全局采样下, 转换和加载阶段的时间相比部分采样时少, 但由于采样开销比较大, 所以总的加载流程所需时间比采用部分采样的方法时间多. 因此, 采用部分采样进行数据分区的加载方法可以获得比采用全局采样进行数据分区的加载方法获得更优的总加载时间.

下面实验, 主要验证在每个采样块中按照等间隔(Interval)选取样本更适合局部有序的数据. 这里在全局采样的情况下进行测试, 实验结果如图 8 所示. 从图 8 可以看出,等间隔选取样本方法采样时间相对随机(Random)选取样本的采样开销差别不大, 但对于有序数据可以获得更好的精确度, 进而实现更好的负载均衡加载, 提高加载的性能. 另外, 对于头部(Split)选取样本方法, 虽然可以减少采样的开销, 但采样的结果并不均衡, 导致加载的时间比较长. 因此, 对于局部有序的数据, 等间隔选取样本方法在采样开销和精确度方面获得了一个比较好的权衡, 可以实现更好的负载均衡加载, 提高数据加载的性能.

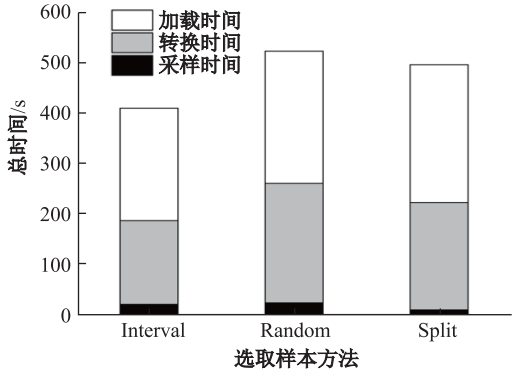


图 8 选取样本方法对加载性能影响

Fig. 8 Effect of sample selection on loading

4.4 容错

这组实验, 主要对比在单个目标存储节点故障情况下, 基于重启全局恢复和基于副本局部恢复这两种恢复方法. 对于多个节点故障和级联故障, 正如第 3.3 节中的讨论, 恢复方法类似单个节点故障. 实验结果如图 9 所示.

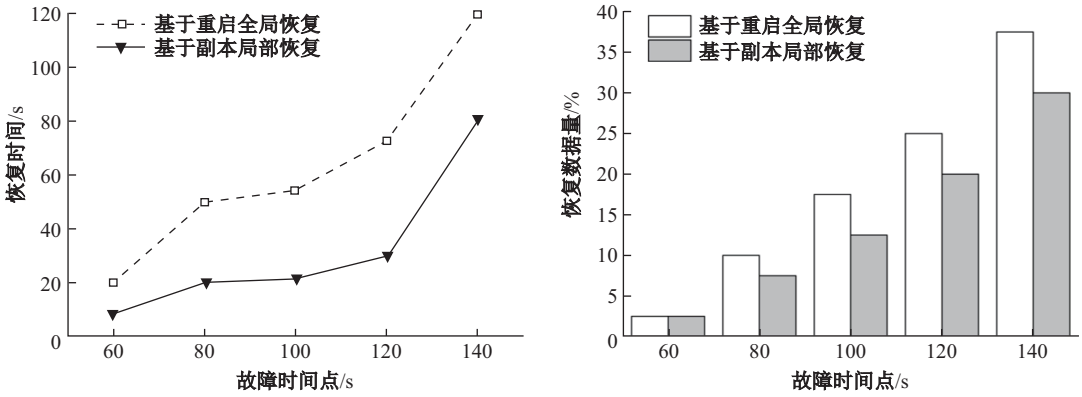


图 9 故障恢复

Fig. 9 Fault recovery

图 9 左图为不同时间点 1 个存储节点宕机情况下两种恢复方法的恢复时间, 可以看出, 基于副本局部恢复方法总是优于基于重启全局恢复方法; 图 9 右图为对应图 9 左图节点宕机时间点时, 需要恢复的副本数占总副本数的比例, 可以看出基于副本局部故障恢复比基于重启全局恢复恢复副本数少. 因此, 基于副本局部恢复方法优于基于重启全局恢复方法.

5　相关工作

对数据加载问题的研究已经有一些具有重要里程碑的工作, 来自工业界和学术界的研究人员提出了一些提高数据加载性能和增强数据加载期间容错能力的想法. 在本节中, 将简要讨论启发本文想法的一些相关工作.

当前, 已经提出了许多方法用于提高数据加载的性能, 大多数工作都是利用并行性来加速数据加载. 从通用方法开始, 该工作[15]的作者介绍了对输入数据进行分区并利用并行性来加快数据加载的想法, 然而并没有详细讨论如何保证数据分区之间的负载均衡. Silberstein 等在插入数据之前添加额外的计划阶段[7], 此阶段允许系统收集有关数据集的统计信息, 根据这些统计信息可以进行有效的数据拆分和负载平衡. 这种方法优化了数据加载到系统的时间, 但是为了尽可能均匀地划分数据对数据集进行全局采样而产生额外的开销, 对于非常大的数据集来说, 这是一个比较严重的问题, 即使在优化之后, 也可能需要数小时才能完成数据加载. 在该工作的扩展[9]中, 利用 Hadoop[10]将数据批量插入到目标系统, 但它没有考虑负载平衡的问题. Ramakrishnan 等旨在解决 Hadoop 中 Reduce 阶段负载不均衡问题[16], 这是通过将分配到 Reduce 中数据密集型的键拆分为几个中等大小的键来减少 Reduce 端负载; 并且在启动 MapReduce 作业前通过采样来识别这些数据密集型键, 将这些数据密集型键存储在分区文件中. 但是, 这种方法主要用于静态分区方案, 即分区规则是由目标存储系统内部来决定而不是由数据加载器来确定.

对于数据加载过程中的容错问题研究则比较少. 当加载的数据量比较少时, 如果目标存储节点发生故障, 可以简单地依靠重新启动加载任务来进行全局恢复, 但是当数据很大时不能忽视数据加载的故障恢复问题. 特别地, 在分布式环境中, 故障是一种常见现象, 在文献 [9] 中, 作者提出了一种支持容错的加载方法, 将 OLTP(On-Line Transaction Processing) 存储系统样式的日志记录和检查点添加到 Hadoop 作业中, 并且利用 Hadoop 本身提供的一些故障处理方法处理向 PNUTS 系统中插入大量记录失败的场景. 本文的研究工作主要利用日志结构合并树系统多副本存储来进行局部故障恢复, 而不需要将故障节点上数据重新全部进行加载.

6　总　　结

在基于日志结构合并树的存储系统上需要一种负载均衡且支持容错处理的数据加载方法, 来减少总加载时间和恢复时间. 为了实现负载均衡的数据加载, 本文提出了预计算分区数目和等间隔选取样本的部分采样方法, 而不是采用预确定分区数目和随机或头部选取样本的全局采样方法来减少采样开销, 同时使各个分区相对比较均衡, 且分区大小小于等于存储系统默认存储块大小. 此外, 为了加速故障恢复, 本文采用了一种基于副本局部故障恢复方法. 实验表明, 通过预计算分区数目和等间隔选取样本的部分采样方法分别优于预确定分区数目和随机或头部选取样本的全局采样方法. 另外, 本文的基于副本局部故障方法也优于基于重启全局故障恢复方法.

[参 考 文 献]

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

[2] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[C]// OSDI '06 Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. 2006: 205-218.

[3] LevelDB[EB/OL]. [2019-06-09]. https://github.com/google/leveldb.

[4] Hbase[EB/OL]. [2019-06-09]. http://hbase.apache.org/.

[5] OceanBase[EB/OL]. [2019-06-09]. https://github.com/alibaba/oceanbase/.

[6] TiDB[EB/OL]. [2019-06-09]. https://university.pingcap.com/

[7] COOPER B F, RAMAKRISHNAN R, SRIVASTAVA U, et al. PNUTS: Yahoo!'s hosted data serving platform[J]. Proceedings of the VLDB Endowment, 2008, 1(2): 1277-1288.

[8] SILBERSTEIN A, COOPER B F, SRIVASTAVA U, et al. Efficient bulk insertion into a distributed ordered table[C]//Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM, 2008: 765-778.

[9] SILBERSTEIN A E, SEARS R, ZHOU W, et al. A batch of PNUTS: Experiences connecting cloud batch and serving systems[C]//Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011: 1101-1112.

[10] Hadoop[EB/OL]. [2019-06-09]. https://hadoop.apache.org/

[11] Cassandra[EB/OL]. [2019-06-09]. http://cassandra.apache.org/.

[12] CDEAR[EB/OL]. [2019-06-09]. https://github.com/daseECNU/Cedar/.

[13] AZQUETA-ALZÚAZ A, PATIÑO-MARTINEZ M, BRONDINO I, et al. Massive data load on distributed database systems over HBase[C]//Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE, 2017: 776-779.

[14] DEAN J, GHEMAWAT S. MapReduce: Simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[15] BARCLAY T, BARNES R, GRAY J, et al. Loading databases using dataflow parallelism[J]. ACM Sigmod Record, 1994, 23(4): 72-83.

[16] RAMAKRISHNAN S R, SWART G, URMANOV A. Balancing reducer skew in MapReduce workloads using progressive sampling[C]//Proceedings of the 3rd ACM Symposium on Cloud Computing. ACM, 2012: Article No.16.

(责任编辑: 李 艺)