

分布式存储系统中改进的一致性哈希算法

王 康¹, 李东静², 陈海光¹

(1. 上海师范大学 信息与机电工程学院, 上海 200234;

2. 南京航空航天大学 计算机科学与技术学院, 江苏 南京 211106)

摘 要:随着网络存储系统的发展,分布式存储中的数据均匀分布和高效定位的问题越来越备受关注。现存的关于分布式系统的数据分布的可靠性和可用性等方面并不能得到有效的保证。文中提出了一种改进的一致性哈希算法,通过对 Redis 存储节点进行逻辑划分成一个组,组内采用主从的模式提高了分布式存储的一致性和可靠性,并分析了同一个组内不同读写策略的数据一致性。经过实验比较,该算法能有效地降低系统平均响应时间,提高系统吞吐量,使分布式存储系统负载更为均衡。当组内主节点宕机时,利用从节点的备份数据以及主从切换可以及时对外提供集群服务,这一点有助于实际的研发分布式存储。

关键词:分布式存储;数据读写策略;Redis;一致性哈希

中图分类号:TP301

文献标识码:A

文章编号:1673-629X(2016)07-0024-06

doi:10.3969/j.issn.1673-629X.2016.07.006

An Improved Consistent Hashing Algorithm in Distributed Storage System

WANG Kang¹, LI Dong-jing², CHEN Hai-guang¹

(1. College of Information, Mechanical and Electrical Engineering, Shanghai Normal University, Shanghai 200234, China;

2. College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

Abstract: With the development of the network storage system, the problem of normal distribution and efficient locating in the distributed storage is more concerned. The existing reliability and availability of distributed data distribution cannot be effectively guaranteed, therefore an improved consistency hash algorithm is presented. By dividing the nodes into a group, the Redis storage can improve the consistency and reliability of the distributed storage, and analyze the data consistency of different reading and writing strategies in the same group. It is verified in the experiment the algorithm can decrease the average response time effectively and raise the throughput, which makes the distributed storage system more balanced. When the group in the main node downtime, the slave node data backup and master-slave switching can timely provide the cluster service to external, which is helpful for the actual development of distributed storage.

Key words: distributed storage; data reading and writing strategy; Redis; consistent hashing

0 引言

所谓分布式存储系统^[1],就是将数据分散存储在多台独立的设备上。随着网络信息技术的快速发展,个人用户、互联网应用等产生了海量的数据。爆炸式增长的数据已经从 PB 级向 EB 级迈进,这些数据的存储和高速访问对分布式存储系统在可用性、可扩展性以及 IO 访问性能上提出了新的挑战。随着数据规模

的扩大,使得存储系统需要不断动态扩大存储规模,并且存储系统必须能够支持新的存储节点不断加入,保证数据在各个存储节点的均匀分布。然而这些数据目前集中部署在单节点存储设备上,随着数据规模的扩大,将会导致单台主机的资源(如内存、磁盘 IO)不能很好地满足海量级数据。由于后续扩容成本非常昂贵,因此迫切需要引入分布式存储系统来解决大数据

收稿日期:2015-11-03

修回日期:2016-03-04

网络出版时间:2016-06-22

基金项目:国家自然科学基金青年基金(41301407);上海市教育创新项目(09YZ154,09YZ247);上海师范大学基金项目(A-3101-12-004005);南京航空航天大学研究生创新基地开放基金(kfj20151607)

作者简介:王 康(1988-),男,硕士研究生,研究方向为数据挖掘;陈海光,副教授,硕导,研究方向为数据挖掘。

网络出版地址:<http://www.cnki.net/kcms/detail/61.1450.TP.20160622.0845.046.html>

的存放和访问问题。

如何最大限度地提高分布式存储系统的存储能力、可靠性,使数据均匀分布,在分布式存储领域已经成为一个急需解决的问题。

文中主要以 Redis 内存数据库为例,针对分布式存储系统对一致性哈希算法展开研究。主要是对已有的一致性哈希算法进行改进,通过将 Redis 存储节点进行逻辑划分成一个组,组内采用主从模式可以提高分布式存储的一致性和可靠性,并分析了同一个组内不同读写策略的数据一致性。

1 基础知识

1.1 相关工作

哈希的英文名字为 Hash,意思为散列,它将任意长度的输入值通过散列算法,变换成固定长度的输出值。这个值就是散列值,即哈希值。哈希算法的方式有很多,在分布式存储系统中主要是对文件名和路径进行哈希,决定数据的分布策略。目前广为流行的有经典的一致性哈希算法、字符串哈希算法、MD4、MD5、SHA-1、Davies-Meyer 等。

文献[2]提出了一致性哈希算法,现在在分布式系统中应用非常广泛。一致性哈希算法在移除或者添加一个服务器时,能够尽可能小地改变已存在的服务请求与处理请求服务器之间的映射关系。常用的字符串哈希算法有 BKDRHash^[3]、APHash、DJBHash、JSHash 等。MD4 算法是哈希算法中较为成熟的算法之一。MD4 算法^[4]可以对任意长度不超过 2^{64} 的消息进行处理,生成一个 128 bit 的哈希值。消息在处理前,首先要进行填充,保证 MD 填充后的 bit 位长度是 512 bit 的整数倍。填充结束后,利用迭代结构和压缩函数来顺序处理每个 512 bit 的消息分组。MD5 算法^[5]是 MD4 算法的升级版。在 MD5 算法中,原始消息的预处理操作和 MD4 是完全相同的,都需要进行补位、补位长度操作,它们的信息摘要的大小都是 128 bit。MD5 在 MD4 的基础上加入了第四轮的计算模式,每一步骤都是一一对应的固定值,改进了 MD4 在第二轮、第三轮计算中的漏洞,完善了访问输入分组的次序,从而减小其对称性和相同性。

SHA-1 算法^[6]是在 MD5 算法基础上发展而来的,其主要功能是从输入长度不大于 2^{64} bit 的明文消息中得到长度为 160 bit 的摘要值。该算法通过计算明文信息得到固定长度的信息摘要,只要原始信息改变,摘要也随之发生改变,而且变化很大。这种发散性可以检测数据的完整性,因此 SHA-1 算法在数字签名中有着广泛的应用。Davies-Meyer 算法^[7-9]是基于对称分组算法的单向散列算法。分组算法在设计和实现

上成本较低,可以构建一个基于该分组密码的哈希函数。

1.2 内存数据库 Redis

Redis 是一个开源的 key-value 存储系统,它通常被称为数据结构服务器,因为 key 可以包含字符串、哈希、链表、集合和有序集合。Redis 支持存储的 Value 类型很多,包括 string(字符串)、list(链表)、set(集合)、sorted set(有序集合)。这些数据类型都支持 push/pop、add/remove 以及取交集和并集及更丰富的操作,Redis 支持各种不同方式的排序。为了保证效率,数据都是缓存在内存中,它也可以周期性地把更新的数据写入磁盘或者把修改操作写入追加的记录文件(相当于 log 文件)。

Redis 对不同数据类型的操作是自动的,因此设置或增加 key 值,从一个集合中增加或者删除一个元素都能安全的操作。其支持简单而快速的主从复制,官网提供了一个数据,slave 在 21 s 即完成了对 Amazon 网站 10 G key set 的复制^[10]。目前全球最大的 Redis 用户是新浪微博。在新浪有 200 多台物理机,400 多个端口正在运行着 Redis,有超过 4G 的数据在 Redis 上来为微博用户提供服务。Redis 的应用场景有很多,例如取最新 N 个数据操作、取 Top N 操作、需要精确设定过期时间的应用、计数器应用、获取某段时间所有数据排重值、实时系统、反垃圾系统等等。Redis 具有速度快、持久化等特点,因此在文中用 Redis 存储节点作为实验部分的环境。

2 算法的基本思想与提出

2.1 问题描述

在分布式存储系统中,往往将一台服务器或者服务器上运行的一个进程称为一个节点^[11-13]。如果有 N 个数据存储节点,那么如何将一个键(key)映射到 N 个存储节点(按 $0 \sim N-1$ 编号)上,通用方法是计算对象的 Hash 值,然后均匀地映射到 N 个存储节点。

$\text{Hash}(\text{key}) \% N$ 一切都运行正常,但是要考虑两种情况:

(1) 一个存储节点 m 宕机了(在实际应用中必须要考虑这种情况),则所有映射到节点 m 的键都会失效。因此需要把存储节点 m 移除,此时节点个数为 $N-1$,映射公式变成了 $\text{Hash}(\text{key}) \% (N-1)$ 。

(2) 由于数据量增大访问加重,需要添加服务器存储节点,此时服务器是 $N+1$ 台,映射公式变成了 $\text{Hash}(\text{key}) \% (N+1)$ 。

在上面两种情况下,突然之间几乎所有的服务器存储节点都失效了,对于服务器而言这是一场灾难。整个系统数据对象的映射位置都需要重新进行计算,

系统无法对外界访问进行正常响应,导致系统处于崩溃状态,那么系统的可靠性和可用性就降低了。增删存储节点会导致同一个 key,在读写操作时分配不到数据真正存储的存储节点,命中率会急剧下降。

一个设计良好的分布式哈希策略应该具有良好的单调性和可靠性、可用性,即服务器存储节点的增删或宕机不会造成大量哈希重定位。基于以上原因,文中提出了一种改进的一致性哈希算法来有效解决单调性和可靠性、可用性问题。

2.2 改进的一致性哈希算法

一致性哈希算法是一种分布式算法,在分布式存储中做数据分布时比较常用,Memcached client 也选择这种算法,解决将 key-value 均匀分布到众多 Memcached server 上的问题。该算法实现思路如下:

首先求出每个存储节点的哈希值,并将其配置到一个 $0 \sim 2^{32}-1$ (哈希值是一个 32 位的无符号整型) 的圆环区间上。其次使用同样的方法求出所需要存储的键的哈希,也将其配置到这个圆环上。然后从数据映射到的位置开始顺时针查找,将数据映射到找到的第一个存储节点上。如果超过 2^{32} 仍然找不到存储节点,就会映射到第一个存储节点上。

当该存储节点所在的服务器宕机或者网络异常时,客户端将无法连接,导致系统服务处于不可用状态。一种改进性方案是采用 Master-Slave 的方式对存储节点逻辑划分成一个存储节点 Group 组,对应于物理上的一个主存储节点和 N 个从存储节点,其中从节点是对主节点的数据复制。以 Redis 存储节点为例,改进后哈希存储方案就会变成主从方式,节点映射时将映射至逻辑上的 Group。例如,当在同一个 Group 内配置一个 master 节点、一个 slave 节点,这时当 master 节点宕机时,通过 Redis 哨兵 (Sentinel) 检测到并及时进行故障恢复 (failover),将 slave 节点替换为 master 节点及时对外提供不间断服务。

改进后的一致性哈希算法具体实现过程如下:

(1) 圆环、Group 数据结构选择及初始化。以 Java 语言为例,圆环可以用 `TreeMap<Long, Group>` 数据结构来模拟,因为 `TreeMap` 中所有的元素都保持着某种固定的顺序,而且在这个环中节点之间是存在顺序关系的,所以 `TreeMap` 的 key 必须实现 `Comparator` 接口,这样易于读取指定哈希值范围的子环。Group 的定义为:

```
public class Group { private Nodemaster; private List<Node> slaves; }
```

其中,Node 表示一个存储节点: `public class Node { private String host; private int port = 6379; // redis 默认端口 private JedisPool pool; // Jedis 连接池,通过 Jedis`

API 链接 Redis 数据库}。

为此,定义 `TreeMap<Long, Group> ketamaNodes`。在这个 MAP 中,键存节点组的哈希值,值存节点组的信息对象。例如,有 4 个节点组 Group A、B、C、D 在圆环 ketamaNodes 初始化。

(2) 通过键的哈希值获取 Group 算法实现。首先定义一个字节数组 digest,其存放 key 的 MD5 信息摘要以及一个 Long 类型的键 key。然后计算这个 digest 摘要的哈希值 Hash,判断圆环 ketamaNodes 中是否存在这个哈希值 Hash,如果存在则 key 的值为 Hash,返回 key 的 Group,否则返回大于这个哈希值 Hash 的子 Map tailMap。如果 tailMap 为空,则 key 为圆环的第一个 key,否则返回 key 的值为 tailMap 的第一个 key。过程如算法 1 所示。要存储键 $k_1 \sim k_9$ 经过哈希计算后在环空间上的位置如图 1 所示: $k_1 \sim k_2$ 映射至 Group A, $k_3 \sim k_4$ 映射至 Group B, $k_5 \sim k_8$ 映射至 Group C, k_9 映射至 Group D。

算法 1: 通过键的哈希值获取 Group 算法实现 (Java 版本)

```
public Group getPrimary ( final String k ) { /* 通过键获取 Group */
    byte[] digest = computeMD5 ( k ); // 计算这个 key 的 MD5 信息摘要
    long hash = hash ( digest ); // 得到这个摘要的哈希值
    if ( ! ketamaNodes. containsKey ( key ) ) { // 如果找到这个节点, 直接取节点, 返回
        SortedMap<Long, Group> tailMap = ketamaNodes. tailMap ( key ); // 得到大于当前 key 的那个子 Map
        if ( tailMap. isEmpty ( ) ) { key = ketamaNodes. firstKey ( ); } // 如果子 Map 为空, 则返回圆环的第一个 key
        else { key = tailMap. firstKey ( ); } // 然后从中取出第一个 key, 就是大于且离它最近的那个 key
    }
    return ketamaNodes. get ( key ); // 返回这个 key 的 Group
}
```

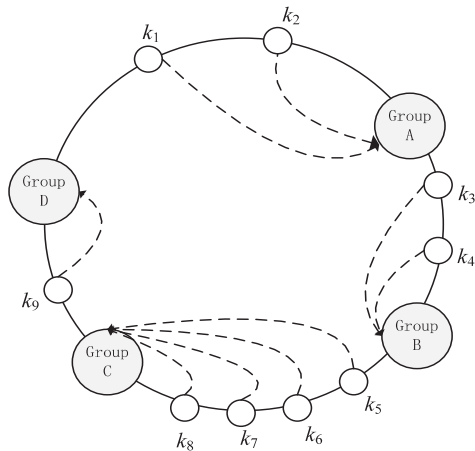


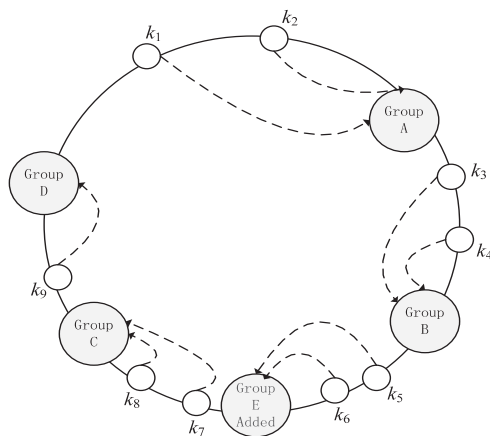
图 1 $k_1 \sim k_9$ 经过哈希计算后的空间示意图

2.2.1 存储节点增删

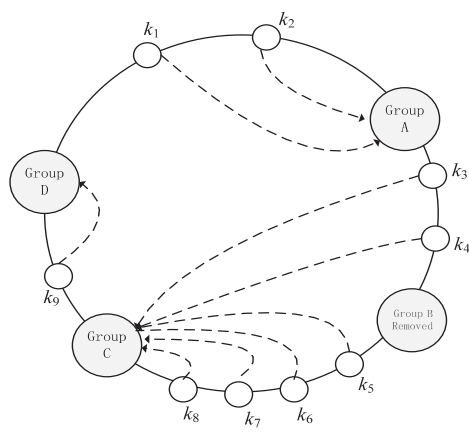
如果只是在同一个 Group 内增加或删除一个从存

储节点,则不需要做任何变动,只是同一个 Group 内节点数增多了,还是正常读写。如果是主存储节点宕机(删除)则通过故障转移机制(如 Redis 的 failover)选举出一个主节点进行数据的读写。如果增加或删除一个存储节点组,则受影响的数据仅仅是新存储节点组

到其环空间中前一个存储节点组之间的数据,其他不会受到影响。增加一个 Group E 之后,之前的 9 个数据 $k_1 \sim k_9$ 受影响的有 k_5, k_6 , 如图 2(a) 所示,删除一个 Group B 之后受影响的只有节点组 k_3, k_4 , 如图 2(b) 所示。



(a) 增加一个 Group E 后的圆环空间



(b) 删除一个 Group B 后的圆环空间

图 2 增加一个 Group E 和删除一个 Group B 后的圆环空间示意图

2.2.2 数据倾斜问题

一致性哈希算法具有随机性。当存储节点组数量较少时节点在环上分布不够均匀,会使得部分节点组分布的数据量较少,部分节点组分布的数据明显增多的情况^[14]。为解决这个问题,提出了基于虚拟节点的改进算法。其核心思路是引入虚拟节点,每个虚拟节点都有一个对应的物理存储节点,而每个物理存储节点可以对应若干个虚拟节点。引入“虚拟节点”后,映射关系就从对{数据→节点}转换到了{数据→虚拟节点}。

下面介绍引入虚拟机节点之后,圆环 TreeMap 的初始化算法过程,如算法 2 所示。

首先定义存储节点组列表的集合 nodes,虚拟节点个数 numReps 初始化为 160,定义一个 TreeMap 类型的圆环 ketamaNodes。对于所有 Group,生成 numReps 个虚拟组节点,将 16 字节的数组每四个字节一组,分别对应一个虚拟节点,计算节点组的信息摘要。对于每四个字节,组成一个 long 值数值,作为这个虚拟节点在环中的唯一 key。然后计算这个存储节点列表中每个节点组的信息摘要对应哈希值 Hash,将这个节点对应的哈希值及 Group 放入圆环中,最后返回这个圆环的 ketamaNodes。

算法 2: Group 在环形上的初始化过程(Java 版本)

```
public TreeMap<Long, Group> KetamaGroupLocator (List<Group>
nodes, int numReps) { //Group 列表及虚拟节点个数
    private TreeMap<Long, Group> ketamaNodes = new TreeMap<Long,
Group>(); //定义圆环 ketamaNodes
    for (Group node : nodes) { //对于所有 Group,生成 numReps 个虚
```

拟组节点

```
for (int i=0; i<numReps/4; i++) { //将 16 字节的数组每四个字
节一组,分别对应一个虚拟节点
    byte[] digest = computeMd5 (node. getGroupName() + i) //计算
节点组的信息摘要
    for (int h=0; h<4; h++) { //对于每四个字节,组成一个 long 值
数值,作为这个虚拟节点的在环中的唯一 key.
        long m = hash (digest, h) //计算这个节点组信息摘要的哈希值
        ketamaNodes. put (m, node); } //这个节点对应的哈希值及
Group 放入圆环中
    return ketamaNodes //返回这个圆环 ketamaNodes
```

2.2.3 Group 内数据读写策略

当一个客户端请求的 key 经过改进的一致性哈希映射到一个 Group,由于同一个 Group 内是采用主从复制的方式来保证可用性。在读写策略中,使用 master 和 slaves 来表达主从关系。于是就产生了同一个 Group 内主存储节点与从存储节点之间读写策略方式:

方式 1: master-only 策略。读写操作都在主节点,毫无疑问,这种方式一致性无法得以保证。

方式 2: master-first-slaves-async 策略。写策略:先写主节点,然后异步同步到从节点。读策略:可以读取任何节点。这种策略是弱一致性的体现,在从节点未及时同步时也会无法保证数据一致性,一般适合写操作频繁的服务,节点挂掉时最近的数据容易丢失。数据可靠性中,主备切换可能丢失数据,对于一个 key 可能存在不同值。

方式 3: master-first-slaves-sync 策略。先主节点然后同步更新到从节点。写策略:多数节点写入成功。

读策略:读取多数节点,并进行一致性验证。写入时在半数以上拷贝写成功时即返回请求。读写数据时都有特定提供服务的备份。备份挂掉半数以下时不会发生数据丢失,挂掉半数以上一般集群会选择进入安全模式(如 NameNode 的安全模式一样,进入后进行数据复制检查)。容灾时,需要各备份间互补数据,以防止不一致性,这个流程会花费一定时间,可根据需求决定彼此间是否继续提供服务,容灾结束后保证数据一致。这种方式数据可靠性高,主备切换不丢失数据。

3 实验

3.1 实验环境

服务器配置为内存 64 G、硬盘 7 T、CPU 为 4 核 * 4 线程 1 596 MHz、网络为 1 个 1 000 M 网卡,所用的操

表 1 Group 组内存储节点的可靠性测试结果汇总

业务场景	查询时已有数据量	查询 key 是否正常	插入是否 正常	平均响应 时间/ms	是否正确 返回结果	结论/建议(可靠性、一致性)
没有读/写请求下,主节点故障	1 000 万	正常	正常	24	是	支持主备切换(当主节点故障时自动升级备节点为主节点)、可靠性(可以正常查询,插入正常)、一致性(主备数据一致,数据未丢失)
有读/写请求下,主节点故障	1 000 万	不正常	不正常	37	否	在主节点 shutdown 瞬间出现部分 get 操作取不到数据(null)、set 操作不正常(控制台会输出一些异常信息)
没有从节点的主节点故障	1 000 万	不正常	不正常	45	否	搭建 Redis 服务不能使用单节点模式(无法进行主从切换),测试程序连续报错
没有读/写请求下,主节点故障恢复重启	1 000 万	正常	正常	29	是	原主节点故障恢复重启后变为备节点,主备节点的数据一致
有读/写请求下,主节点故障恢复重启	1 000 万	重启后正常	重启后正常	41	是	丢失部分数据
没有从节点的主节点故障恢复重启	1 000 万	重启后正常	重启后正常	20	是	丢失部分数据
手动新增从节点,进行读写请求,测试数据一致性	1 000 万	正常	正常	35	是	新加入的 Slave 节点会自动同步主节点的数据,同步完成后达到主备数据一致

3.2.2 性能测试

衡量系统性能好坏的一个重要指标就是系统的吞吐量,有时候人们也常称为 TPS(Transactions Per Second,单位时间内系统发生的事务数)。在研究一致性哈希算法的过程中,通过性能测试来测试分布式存储系统的 TPS 峰值。

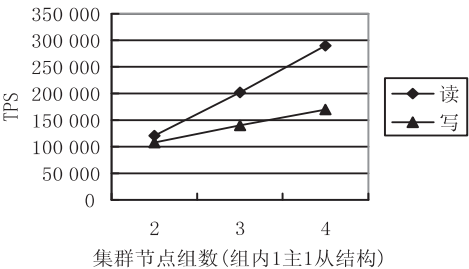


图 3 不同 Group 组数集群 TPS 性能对比图
1 000 万个请求,数据大小为 32 B 时进行高并发

作系统为 CentOS 6.4。在 4 台服务器上,分别部署 2 ~ 4 个 Redis 节点组,在组内采用 1 主 1 从结构,启用 Redis 数据库自有的复制 Replication 机制(需要配置节点间的主从关系)。

3.2 实验结果与分析

3.2.1 可靠性测试

通过对 1 000 万条 KV 数据进行客户端读写操作,在不同场景下进行节点故障测试来验证。分布式 Redis 集群的可靠性实验结果如表 1 所示。

结果表明,有读写请求下,由于主从切换有一定的延迟,主节点故障会丢失部分数据,以及没有从节点的节点组也会丢失数据,重启后恢复正常,因此需要一主多从的模式来保障整个 Redis 集群的可靠性。

测试。当进行 set 操作,增加一个节点时,写操作的吞吐量提升了 30%、读操作的吞吐量提升了 50%。通过集群节点的增加,读写性能都有提升。另外,读操作的性能要明显好于写操作,结果如图 3 所示。同样,可以明显看出,当集群内增加节点组时,读写操作的平均响应时间都有所缩短,在 10 ms 以内符合预期,结果如图 4 所示。

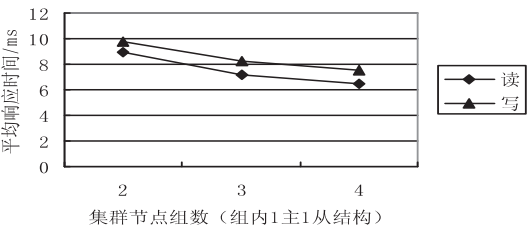


图 4 不同 Group 组数集群数据读写
操作平均响应时间对比

3.2.3 分布平均性测试

当集群的节点组数为 2 个或 3 个,进行写操作 1 000 万次时,进行数据库写操作验证,测试结果如图 5 所示。

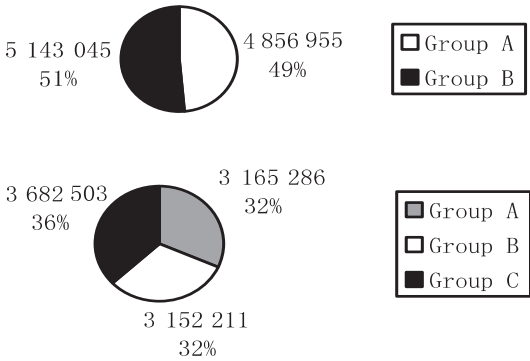


图 5 数据均匀性分布

当采用 2 个节点组时,Group 内写入的数据量分别占 51%、49%,采用 3 个节点组时,Group 内写入的数据量分别占 36%、32%、32%。节点组之间数据分布相对比较均匀,避免了数据倾斜问题的发生。

4 结束语

文中对分布式存储领域的技术展开了研究,简单介绍了 Redis 数据库及分布式系统的 CAP 理论,详细说明了普通哈希算法的不足之处。提出基于 Group 方式对物理上的存储节点进行逻辑划分,改进的一致性哈希算法提高了分布式存储系统的可靠性、可用性,并以 Redis 数据为例进行实验验证,通过增加存储节点组大大提高了集群的系统负载。

参考文献:

[1] 陆嘉恒. 分布式系统及云计算概论[M]. 北京:清华大学出版社,2013.

[2] Devine R. Design and implementation of DDH: a distributed dynamic hashing algorithm[M]//Foundations of data organization and algorithms. Berlin:Springer,1993:101-114.

[3] Kernighan B W, Ritchie D M. The C programming language[M]. [s. l.]:Prentice Hall,1988:36-40.

[4] Rivest R L. The MD4 message digest algorithm[C]//Proc of CRYPTO'90. [s. l.]:[s. n.],1991:303-311.

[5] Rivest R. The MD5 message-digest algorithm[S]. [s. l.]:IETF,1992.

[6] 张绍兰. 几类密码 Hash 函数的设计和安全性分析[D]. 北京:北京邮电大学,2011.

[7] Winternitz R S. A secure one-way hash function built from DES[C]//Proc of IEEE symposium on security and privacy. [s. l.]:IEEE Computer Society,1984:88-88.

[8] 余秦勇,陈 林,童 斌. 一种无中心的云存储架构分析[J]. 通信技术,2012,45(8):123-126.

[9] 李 正. 杂凑函数结构研究现状及新的结构设计[D]. 济南:山东大学,2010.

[10] 姜大光,奚加鹏. 分布式存储系统(OceanStore)的复制策略[J]. 计算机工程与科学,2008,30(8):144-146.

[11] 杨彧剑,林 波. 分布式存储系统中一致性哈希算法的研究[J]. 电脑知识与技术,2011,7(22):5295-5296.

[12] 赵 飞,苏 忠. 一致性哈希算法在数据库集群上的拓展应用[J]. 成都信息工程学院学报,2015,30(1):52-58.

[13] 郭 宁,张 新. 一致性哈希算法在多处进程分配的应用[J]. 计算机与现代化,2013(9):71-74.

[14] 周 瑜. 一种基于一致性 hash 算法存储资源的方法:CN, CN 103281358 A[P]. 2013.

(上接第 23 页)

[3] 边肇祺,张学工. 模式识别[M]. 第 2 版. 北京:清华大学出版社,2000.

[4] 张学工. 模式识别[M]. 第 3 版. 北京:清华大学出版社,2010.

[5] 苏高利,邓芳萍. 论基于 MATLAB 语言的 BP 神经网络的改进算法[J]. 科技通报,2003,19(2):130-135.

[6] 韩小孩,张耀辉,孙福军,等. 基于主成分分析的指标权重确定方法[J]. 四川兵工学报,2012,33(10):124-126.

[7] 齐兴敏. 基于 PCA 的人脸识别技术的研究[D]. 武汉:武汉理工大学,2007.

[8] 叶 林,陈岳林,林景亮. 基于 HOG 的行人快速检测[J]. 计算机工程,2010,36(22):206-207.

[9] 姚雪琴,李晓华,周激流. 基于边缘对称性和 HOG 的行人检测方法[J]. 计算机工程,2012,38(5):179-182.

[10] Dalal N, Triggs B. Histograms of oriented gradients for human detection[C]//Proceedings of IEEE computer society confer-

ence on computer vision and pattern recognition. [s. l.]:IEEE Press,2005:886-893.

[11] Raina R, Battle A, Lee Honglak, et al. self-taught learning: transfer learning from unlabeled data[C]//Proc of international conference on machine learning. [s. l.]:[s. n.], 2007:759-766.

[12] Le Q V, Ngiam J, Coates A, et al. On optimization methods for deep learning[C]//Proc of international conference on machine learning. [s. l.]:[s. n.], 2011:265-272.

[13] Ranzato M, Huang F J, Boureau Y L, et al. Unsupervised learning of invariant feature hierarchies with applications to object recognition[C]//Proc of IEEE conference on computer vision and pattern recognition. [s. l.]:IEEE,2007:1-8.

[14] Lecun Y, Bottou L, Bengio Y, et al. Gradient based learning applied to document recognition[J]. Proceedings of the IEEE, 1998,86(11):2278-2324.