软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(12):3192−3207 [doi: 10.13328/j.cnki.jos.004930]

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

分布式文件系统元数据服务的负载均衡框架*

孙 耀[1,2], 刘 杰[1], 叶 丹[1], 钟 华[1]

[1](中国科学院 软件研究所,北京 100190)

[2](中国科学院大学,北京 100190)

通讯作者: 孙耀, E-mail: sunyao10@otcaix.iscas.ac.cn

摘 要: 请求负载均衡,是分布式文件系统元数据管理需要面对的核心问题.以最大化元数据服务器集群吞吐量为目标,在已有元数据管理层之上设计实现了一种分布式缓存框架,专门管理热点元数据,均衡不断变化的负载.与已有的元数据负载均衡架构相比,这种两层的负载均衡架构灵活度更高,对负载的感知能力更强,并且避免了热点元数据重新分布、迁移引起的元数据命名空间结构被破坏的情况.经观察分析,元数据尺寸小、数量大,预取错误元数据带来的代价远远小于预取错误数据带来的代价.针对元数据的以上鲜明特点,提出一种元数据预取策略和基于预取机制的元数据缓存替换算法,加强了上述分布式缓存层的性能,这种两层的元数据负载均衡框架同时考虑了缓存一致性的问题.最后,在一个真实的分布式文件系统中验证了框架及方法的有效性.

关键词: 元数据服务器;分布式文件系统;负载均衡;预取;缓存

中图法分类号: TP316

中文引用格式: 孙耀,刘杰,叶丹,钟华.分布式文件系统元数据服务的负载均衡框架.软件学报,2016,27(12):3192−3207. http://www.jos.org.cn/1000-9825/4930.htm

英文引用格式: Sun Y, Liu J, Ye D, Zhong H. Load balancing framework for metadata service of distributed file systems. Ruan Jian Xue Bao/Journal of Software, 2016,27(12):3192−3207 (in Chinese). http://www.jos.org.cn/1000-9825/4930.htm

Load Balancing Framework for Metadata Service of Distributed File Systems

SUN Yao[1,2], LIU Jie[1], YE Dan[1], ZHONG Hua[1]

[1](Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Request load balancing is the core issue in distributed file system metadata management. To maximize the throughput of the metadata service, an adaptive request load balancing framework is critical. This paper presents a distributed cache framework above the distributed metadata management schemes to manage hotspots rather than managing all metadata to achieve request load balancing. Compared with the existing distributed metadata load balancing framework, it has a higher degree of flexibility of the two-tier load balancing structure, and is stronger on the perception of the overall load. It also avoids hot spots redistribution and namespace structure destruction caused by metadata migration. Compared with data, metadata has its own distinct characteristics, such as small size and large quantity. The cost of non-use metadata prefetching is much less than data prefetching. Based on this study, a time period-based prefetching strategy and a perfecting-based adaptive replacement cache algorithm are devised to improve the performance of the

* 基金项目: 国家自然科学基金(61170074, 61202065, U1435220); 国家高技术研究发展计划(863)(2013AA041301); 国家科技支撑计划(2015BAH18F02)

Foundation item: National Natural Science Foundation of China (61170074, 61202065, U1435220); National High-Tech R&D Program of China (863) (2013AA041301); National Key Technology Research and Development Program of the Ministry of Science and Technology of China (2015BAH18F02)

收稿时间: 2014-11-14; 修改时间: 2015-08-31; 采用时间: 2015-10-13; jos 在线出版时间: 2015-11-15

CNKI 网络优先出版: 2015-11-16 09:22:21, http://www.cnki.net/kcms/detail/11.2560.TP.20151116.0922.003.html

distributed caching layer to adapt constantly changing workloads. Finally, the presented approach is evaluated with a Hadoop distributed file system cluster.

Key words: metadata server; distributed file system; load balancing; caching; prefetching

大数据存储解决方案需要满足高可扩展性,分布式文件系统因其良好的可扩展性成为大数据存储系统的核心,元数据是新型分布式文件系统数据访问模型的关键组成部分,包括文件路径、目录结构以及文件属性描述信息.为了实现高可扩展特性,新型分布式文件系统采用独立集群管理元数据.然而,这种新型分布式文件系统架构与互联网应用的特点给元数据集群的请求负载均衡问题带来了巨大的挑战.在新型分布式文件系统文件访问流程中,元数据的一个重要作用是数据对象的定位.客户端访问一个文件数据首先需要获取这个文件对应的元数据信息,比如文件位置信息.因此,元数据的管理效率决定着文件的访问性能.当前,许多互联网企业利用这种架构的分布式文件系统存储数据,例如谷歌的 GFS[1]、Facebook 优化过的 HDFS[2].基于此类分布式文件系统的互联网应用普遍具有一些非常重要的特点,包括 Petabyle 规模的数据集群[3]、大量的并发请求[4,5]、高比例的小文件存储服务(图片等)[6].

本文着重解决元数据集群负载倾斜频繁变化问题.当并发请求数量超过一个元数据服务器的处理能力时,这个元数据服务器的性能出现下降,集群出现负载不均衡情况,表现为平均响应时间上升,元数据集群整体吞吐量下降.出现负载倾斜的元数据服务器节点越多,集群吞吐量越低.因此,维持元数据集群负载均衡是避免分布式文件系统出现性能瓶颈的关键.与数据集群相比,元数据集群有 3 个特点.

- 第一,元数据集群规模远小于数据集群规模,而每个元数据服务器节点接收到的客户端请求却远高于每个数据服务器节点[1,2,7].一个已部署的分布式文件系统经常包含几百个数据服务器节点,十个元数据服务器节点;
- 第二,元数据请求的响应时间远小于数据请求响应时间,操作更迅速;
- 第三,元数据集群潜在的热点更为复杂,包括突发性热点、周期性热点以及持久性热点.多种热点并存的场景中,热点出现的时间、位置具有不确定性.由于一个目录或子树被频繁访问而导致的突发性高负载[8,9]情况出现频率很高.

元数据集群服务的这些特点,使得元数据集群出现负载倾斜的频率高于数据集群.

已有的分布式元数据管理架构难于处理负载倾斜频繁变化的问题.例如,子树划分的方式会带来请求分布不均衡的问题.当出现请求负载不均衡的情况时,需要迁移热点元数据到其他计算资源相对空闲的节点.但是,迁移操作会给元数据命名空间的层次结构带来损坏.基于哈希的分布式元数据管理方式可以均衡元数据的分布[10,11],但是当一个文件或者目录包含的数据成为热点时,这种方式仍不能解决负载倾斜的问题.一些分布式文件系统利用缓存的方式解决热点的问题,对于局部热点而言,缓存是一种有效的方法.然而,如果大量客户端同时请求缓存内容,缓存的位置无论是在客户端还是服务器端都不是一个好的选择.基于以上分析,我们建立了一个独立于客户端与服务器端的缓存层,专门管理热点元数据.为了保证系统的可扩展性,我们利用 DHT 组织管理元数据缓存.DHT 可以弹性控制缓存大小和缓存节点数量,该缓存层可以部署在一个独立的集群也可以部署在元数据服务器集群.对于缓存一致性的问题,我们设计了一种可调的缓存更新策略.

在负载倾斜频繁变化的场景中,客户端的请求分布是不可预知的.我们的目标是均衡客户端请求的分布,如何准确快速地预测负载倾斜节点以及热点元数据是一个挑战.通过缓存替换算法中时间与空间的预测机制可以获得热点元数据,但是仅仅依靠现有的缓存替换算法难以预测节点为潜在的负载倾斜节点.鉴于此,我们采用一种预取方法,预先判断节点成为潜在负载倾斜节点的可能性,并缓存其中的热点,降低出现负载倾斜节点的概率.传统的监控预测手段是先监测再计算得出负载倾斜节点,当负载倾斜频繁变化的时候,这种方法效果不理想,并且很难设置监测的时间滑动窗口,设置的时间滑动窗口太长会漏掉负载高峰值,设置的时间滑动窗口太短则调整开销过大.本文结合问题的难点与元数据固有特点,采用一种敏捷的监测判断方法预测潜在的负载倾斜节点,即:在监测的同时计算当前的负载倾斜节点,将判断依据由滑动窗口内的平均负载值转变为滑动窗口内的负载变化趋势.在负载倾斜频繁变化的场景中,这个改变有利于判断潜在的负载倾斜节点.基于这种思想,我们

设计实现了一种新的预取机制和一种基于预取的缓存替换算法.然而,错误的预取是无效的.幸运的是,无效元数据预取的代价远小于数据,这是本文采用概率思想设计预取策略的一个重要原因.预取过程中,在准确率与实效性之间做权衡,选择最佳方案.

本文的贡献主要包括以下几个方面:

(1) 建立了一个两层的元数据负载均衡框架,自适应管理潜在的热点,解决了元数据集群负载倾斜频繁变化导致的系统性能瓶颈问题;

(2) 依据元数据和元数据集群特点,设计了一种预取机制,可快速定位将要发生负载倾斜的节点;

(3) 设计实现了一种基于预取的缓存替换算法;

(4) 设计实现了一种缓存可调一致性策略.

框架的适用范围是:

HDFS[1]以及其他采用独立集群管理元数据的分布式文件系统[2−5,7,9].

1 背景与问题分析

1.1 元数据管理架构及访问模式

在分析问题之前,先介绍一下 HDFS(hadoop distributed file system).如图 1 所示,主要包括 3 个部分:客户端、元数据服务器、数据服务器.客户端可以是一个应用、一个 MapReduce 程序或者是一个 NoSQL 数据库.通常情况下,数据服务器集群是由几百个节点组成的,用来存储实际的物理数据块,每个数据块都有多个备份.元数据服务器集群则为几个高性能计算机组成,负责存储文件系统命名空间以及数据块与数据服务器节点的映射关系.数据存储在磁盘上,为了快速响应,元数据是基于内存管理的.文件系统访问文件数据的流程分两个步骤:首先客户端向元数据服务器发出请求,获取文件位置信息;然后,客户端根据元数据向数据服务器发送请求,读取对应的数据块.

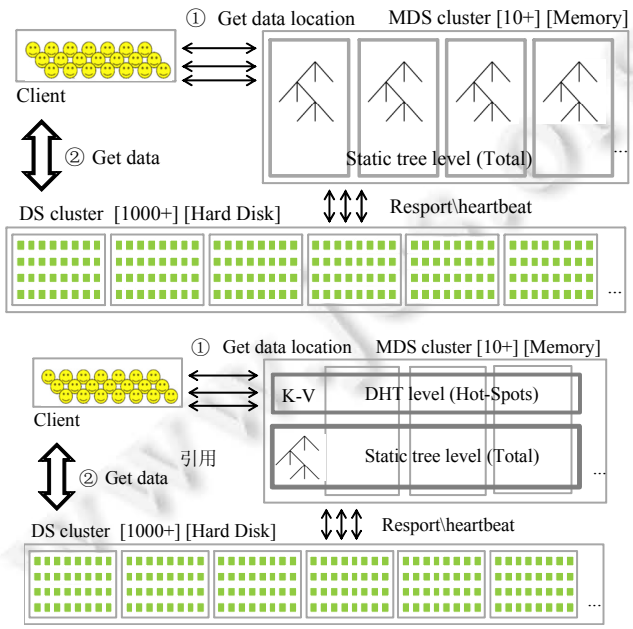

Fig.1 Comparison between the original architecture and the optimized architecture for HDFS

图 1 HDFS 原架构(上图)与优化后架构(下图)对比图

本文在 HDFS 原架构的元数据服务器集群之上部署一个完全独立的分布式缓存层,专门管理热点元数据.

优化后的 HDFS 架构一方面可以采用树形结构管理元数据(此种技术实现方式更有利于分布式文件系统的元数据查询效率),另一方面可以避免树形结构在突发高负载模式下的性能瓶颈问题.与现在流行的分布式缓存架构相比, 本文缓存架构优势主要体现在以下两点:

- 第一,缓存位置.提出的分布式缓存架构独立于元数据服务器集群及客户端,采用这样的逻辑架构设计,不仅可以灵活应用于现有的分布式文件系统,还避免了客户端缓存状态同步导致的系统性能开销,同时,可以灵活控制缓存的位置、数量;
- 第二,缓存替换算法.现在流行的缓存架构采用 LRU,LFU,ARC 等缓存替换算法,现有缓存替换算法缺少预测分布式环境出现负载倾斜节点位置的功能,提出的基于预取技术的缓存替换算法可以有效预测负载倾斜发生位置,并将可能导致负载倾斜的热点元数据预取至缓存空间,进而提升分布式缓存框架在负载倾斜频繁变化场景中的性能.

Roselli 等人[10]指出,元数据操作占全部文件系统操作的 50%以上.以 HDFS 为例,一个元数据集群服务负载分为两部分:一是负载来自客户端请求,如获取文件路径、建立一个新的子目录、删除一个目录以及重命名操作等;二是负载来自数据集群的报告以及心跳请求.当数据量改变或者数据集群节点数量改变时,元数据服务器承担管理者角色,根据数据服务器节点的心跳、报告信息调整数据的分布放置,于是会产生一系列的元数据操作.文献[11]显示:来自数据集群的元数据操作占全部元数据操作的 30%,这部分操作产生的负载相对固定.但来自客户端的负载是无法预测的,不可预知的客户端请求负载会导致元数据服务器集群发生负载倾斜的情况.

1.2 元数据管理架构及访问模式

为研究突发高负载模式对元数据集群总体吞吐量产生的影响,本文采用 Hadoop 的元数据服务基准测试工具 NameNodeBench(简称 NNBench),针对不同数量的客户端并发访问场景进行测试.NNBench 主要用来度量 HDFS(hadoop distributed file syste)的元数据服务性能与扩展性[12],用户可以通过不同的参数配置模拟不同访问负载,比如文件数量、文件大小以及副本数量等.在 10 节点元数据服务器集群,为模拟高突发负载场景,改进了 NNBench,使其支持多线程参数配置.为精简篇幅考虑,测试环境及负载的详细情况见第 4 节的表 2、表 3.

采用两个参数模拟真实应用环境的 FLS(frequently load skew)场景:一个是每个客户端的并发线程数量,另一个是每个元数据服务器节点上热点的数量.利用不同数量的客户端线程和热点元数据项模拟客户端请求在元数据服务器集群的动态分布.不断地增加并发数量,改变参数时间间隔的变化会引起负载倾斜频繁发生变化.HDFS2.0 版本中,Federation 结构采用了经典的 client side mounts table 方法[1],所以客户端是知道元数据的分布情况的,进而可以控制负载倾斜的变化.图 2 是一个元数据服务器不同时间段内负载不均衡分布的情况,其他元数据服务器情况相似.横轴表示一个滑动窗口长度,纵轴表示在不同时间点请求到达的情况(KTPS 表示每秒中处理请求事务的数量).

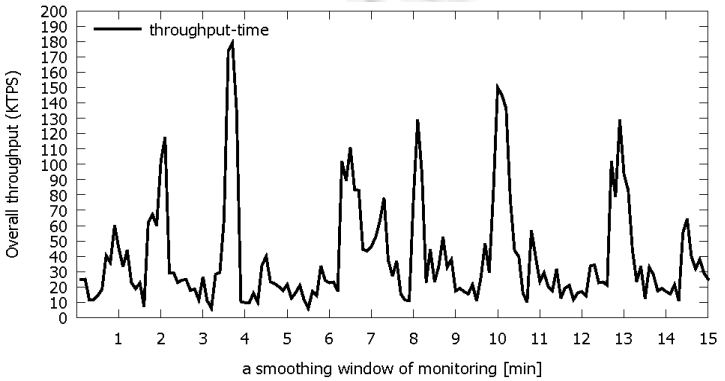

Fig.2 Arriving request distribution in a monitoring smoothing window

图 2 一个滑动监控窗口内的请求分布

图 3 表达的是不同并发线程场景中的元数据服务器集群各个节点平均响应时间,当每个客户端并发线程数量设置为 30,50,100 时,元数据集群中各个节点的最长平均响应时间与最短平均响应时间的差值为 18ms、39ms、81ms.从这组测试结果可以看出:随着并发访问数量的增加,元数据集群频繁出现负载倾斜的情况,负载倾斜越频繁,元数据响应时间越长.

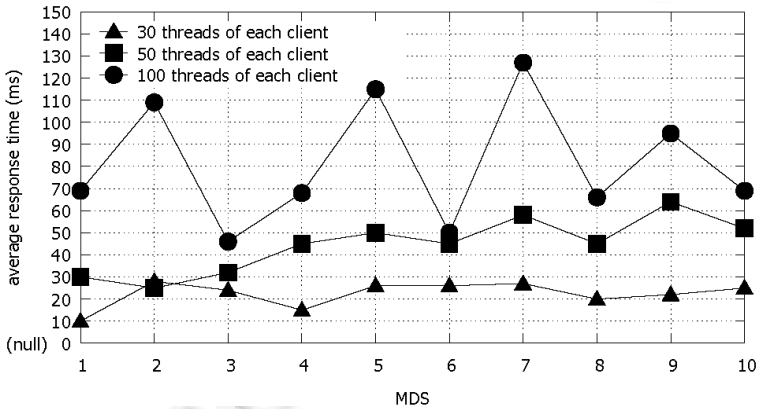

Fig.3 Degrees of FLS with different numbers of concurrent threads

图 3 不同并发访问负载情况下的负载倾斜频繁度

2 关键机制

在本节中,首先形式化描述问题模型FLS,然后介绍负载均衡框架包含的3个关键机制,负载均衡框架如图4所示,3 个关键机制分别是一种基于时间片段的预取策略 TPPS(time period-based prefetching strategy)、基于预取技术的缓存替换算法 PARC(perfecting-based adaptive replacement cache algorithm)以及一种可调缓存一致性策略 ACCS(adjustable cache consistency strategy).该负载均衡框架总的目标是减少元数据集群负载倾斜发生的频率,保持客户端请求的分布均衡.

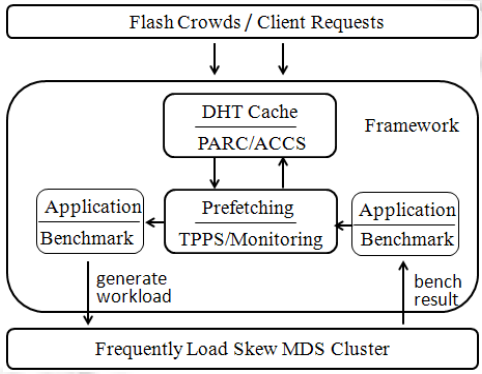

Fig.4 Overview of the solution

图 4 解决方案的总体思路

表1列出了模型中的变量,n代表元数据集群节点数量,在一个负载变化周期内,m代表元数据集群中出现负载倾斜的节点数量($m<n$).负载倾斜分布描述为

$$S=(P,P,\ldots,P) \tag{1}$$

Table 1 Symbols used in the approach

表 1 方法中用到的符号列表

Symbol	Meaning
n	The number of MDSs
m	The number of load skews in T
p_i	Request number of ith MDS in
P_i	A random variable for the event that p_i is a winner
S	Load skews distribution
T	A time smoothing window
t	A load skew adjustmenting period, equal $n\times t$

在此模型中,P_i代表时间 t_i内第 i 个元数据服务器节点接收到的访问请求数量.为了不失一般性,将 P_i表达为一个单调递减序列:

$$p_1>p_2>\ldots>p_n \tag{2}$$

真实系统中的负载类型是多种多样的,如均衡负载、热点负载等.为了表达清晰,模型中假设元数据集群只应对热点负载,这是由于热点负载是导致元数据集群节点出现负载倾斜情况的主要原因.在云存储服务场景中,热点负载又分为多种情况,如持续热点、周期性热点、突发性热点等.每一种类型的热点均可以制造出负载倾斜的场景,其中,突发性高负载是最容易制造出负载倾斜场景的热点类型,同时,发生频繁负载倾斜的情况也主要缘于此热点类型.在数学范畴,突发性意味着不可预知,即随机性.不同于随机性概念的是突发性具有时间约束属性,在模型中,这种时间约束属性刻画为一个时间周期.这个时间滑动窗口表示为时间序列,描述如下:

$$T=t_1+t_2+\ldots+t_n \tag{3}$$

t_i代表一个时间周期中的一个时间段,突发性高负载模式下,P_i的大小分布是随机的,时间 t_i内,p_i是否为最大值是一个随机事件.

核心思想:基于公式(3),利用 t_i时间段内的 p_i评估元数据服务器的负载倾斜情况,而不是用一个时间周期 T 内的 p_i 评估元数据服务器的负载倾斜情况.p_i存在先后顺序,换句话说,采用一个时间周期内的负载变化趋势鉴别负载倾斜度,而不是时间周期内的负载平均值.这样的处理方式更有利于定位潜在的负载倾斜节点,从而可以更加及时地感知频繁负载倾斜导致的性能抖动.

2.1 一种基于时间片段的预取策略(TPPS)

在负载倾斜频繁变化场景中,挑战在于如何快速定位潜在的负载倾斜节点.因此,本文重点关注预取的位置和预取动作发生的时刻.在我们设计实现的预取策略中,准确性和时效性是两个重要的突破点.对于准确性而言,目标是找到最有潜力发生负载倾斜的节点.对于时效性而言,目标是尽可能的缩短定位时间.基于公式(1)~公式(3),我们采用众所周知的雇佣问题模型[13,23]实现我们的监测算法.

图 4 解决方案中提出的预取技术分为两个步骤:第 1 步,TPPS 定位预取位置(发生负载倾斜的元数据服务器节点);第 2 步,查询导致元数据服务器节点发生负载倾斜的热点元数据.雇佣问题模型应用于预取第 1 步,文章选用雇佣问题模型,目的是定位发生负载倾斜的元数据服务器节点位置.选择雇佣问题模型的原因有两个:第一,雇佣模型描述的是获胜者更新的频繁度问题,这个特点符合对负载倾斜频繁度刻画的要求;第二,在线雇佣模型本质上是解决如何更快速地找到获胜者,这个特点恰好满足预取的时效性要求.

本文提出的 TPPS 方法是从雇佣模型演化而来的,在此模型中,雇主采用一对一顺序的方式面试竞争者,直至选取到最适合人选.这样的模式本质上是度量获胜者更新频率问题.算法 1 基于这个模型描述了鉴别元数据集群负载倾斜变化频率问题.在问题场景中,需要在时间 T 内逐个时间段内对比找到 m 个负倾斜节点.如算法 1 的第 5 行表达式描述的情况,如果 p_i是最佳选择,那么将它作为预取的目标,根据表 1 描述,P_i是一个 t_i时刻第 i 个元数据服务器接收到访问请求数量是否为最高值事件的指示器随机变量,在突发性访问负载场景,p_i 的取值范围是 0~100%,从 p_1~p_{i-1}取胜的几率同等于 p_i.所以 p_i是最佳选择的概率为 1/i,依据指示器随机变量理论:

$$E[P_i]=1/i.$$

结合公式(1),进一步得出:

$$E[S]=E\left[\sum_{i=1}^{n}P_i\right]=\sum_{i=1}^{n}E[P_i]=\ln n+O(1) \tag{4}$$

即:在 FLS 模型中,T 时间周期内,元数据集群发生负载倾斜次数为 $m=O\ln n$.与此相比,如果 S 是一个递增序列,那么 $m=1$,对于 FLS 来说,这是最差的情况;如果 S 是一个递减序列,那么 $m=n$,对于 FLS 来说,这是最好的情况.

为了达到快速定位的目标,我们对算法 1 做了优化.比较前 k 个元数据服务器节点的负载情况,根据负载对比情况选取一个当前最高负载节点 *BestWeight*.然后,继续对比余下的节点,第 1 个高于 *BestWeight* 节点负载的节点被认为是最有潜力成为 T 时间周期内负载最高的节点.优化后的算法描述见算法 2,其中的关键点是采用一个概率的方式减少寻找最高负载节点的时间.难点在于如何确定 k 值,保证准确性,基于概率理论分析得到算法 2 准确定位负载最高节点的概率为 n/e.换句话说,如果用 $k=n/e$ 来实现我们的算法,则可以不低于 $1/e$ 的概率选取到目标节点.

输入:一个连续时间段内,请求负载总量在元数据服务器集群的分布情况;

输出:一个连续时间段内,请求负载量可能达到最高值的元数据服务器节点位置.

算法 1. Identify load skew MDSs.

1: best load skew *node*←0
2: for p_i←p_i to p_n do
3: compare p_{i-1} and p_i
4: if p_i is better than best then
5: *best*←*i*
6: select p_i
7: end if
8: end for

算法 2. Online identify potential load skew MDSs.

1: *bestscore*(current best load skew node)←−∞
2: for *i*←1 to *k* do
3: if p_i>*bestscore* and *bestscore*>*warning level* then
4: *bestscore*←p_i
5: end if
6: for *i*←*k*+1 to *n* do
7: if p_i>*bestscore* then
8: return p_i
9: end if
10: end for
11: return pn
12: end

如图 5 所示,为了提高预取的准确率,同一时刻采用不同长度区间测试模型的性能.我们将一个时间窗口平均分成多个等长度的时间段,采用连续不同的时间段内的负载值作为不同元数据服务器负载倾斜定位的量化依据,此采样方式决定了算法的时间效率.

算法 2 与算法 1 的开销主要体现在计算时间的长度,算法 2 以准确率换取了更少的计算时间,所以算法 2 在效率方面优于算法 1.之所以可以选择牺牲准确率,是因为研究对象(元数据)的自身特点决定的,因元数据尺寸小的特点,预取元错误元数据的代价远远小于预取错误数据的代价,这也是选用算法 2 的基础.

证明:假设不存在两个节点在同一个时间段内负载完全一致的情况,为了以上述概率定位到潜在负载倾斜节点,需要满足两个必要条件:第一,负载最高值必须发生在时间片 t_i 上;第二是算法不能在时间片 t_{k-1} 到 t_{i-1} 之间

出现 P_i.令 S 表示成功选择最高负载值的事件,S_i 表示成功选择到最高负载的事件发生在时间片 t_i 上,由于不同的 S_i 不相交,有 $\Pr\{S\}=\sum_{i=1}^{n}\Pr\{S_i\}$.由于当最高负载发生在前 k 个任何一个时间片上时算法不会奏效,所以,

$$\Pr\{S\}=\sum_{i=k+1}^{n}\Pr\{S_i\}.$$

用 O_i 表示在时间 t_{k+1} 到 t_{i-1} 没有出现最高负载的事件,用 B_i 表示负载最高值恰好发生在时间片 t_i 上.事件 O_i 与 B_i 是两个相互独立事件,事件 O_i 只依赖于时间 t_1 到 t_{i-1} 负载值的相对顺序,而 B_i 只依赖于时间片 t_i 上的负载值是否大于其他时间片上的负载值.时间片 t_1 到 t_{i-1} 的负载值出现的相对顺序如何,并不影响时间片 t_i 上的负载值是否为最高;而时间片 t_i 上的负载值多少,也同样影响不到时间片 t_1 到 t_{i-1} 的负载值的出现次序.因此,

$$\Pr\{S_i\}=\Pr\{B_i\cap O_i\}=\Pr\{B_i\}\Pr\{O_i\}.$$

因为最大负载值可能出现在 t_1 到 t_n 的任何一个时间片上,所以 $pr\{b_i\}$ 的概率是 $1/n$.如果 O_i 发生,则最大负载值在前 k 个时间片其中一个上发生.因此,$pr\{O_i\}=k/i-1$,$pr\{O_i\}=k/(n(i-1))$.根据公式 $\Pr\{S\}=\sum_{i=1}^{n}\Pr\{S_i\}$,有:

$$\Pr\{S\}=\sum_{i=k+1}^{n}\{S_i\}=\sum_{i=k+1}^{n}\left\{\frac{k}{n(i-1)}\right\}=\frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}.$$

利用积分来近似约束这个和数的上界和下界,有:

$$\int_k^n\frac{1}{x}\mathrm{d}x\leqslant\sum_{i=k}^{n-1}\frac{1}{i}\leqslant\int_{k-1}^{n-1}\frac{1}{x}\mathrm{d}x.$$

解积分,得到:

$$\frac{k}{n}(\ln n-\ln k)\leqslant\Pr\{S\}\leqslant\frac{k}{n}(\ln(n-1)-\ln(k-1)).$$

因为算法目的是最大化 S 成功的概率,即最大化 $pr(S)$的下界,于是,对上限求导得 $k=n/e$.

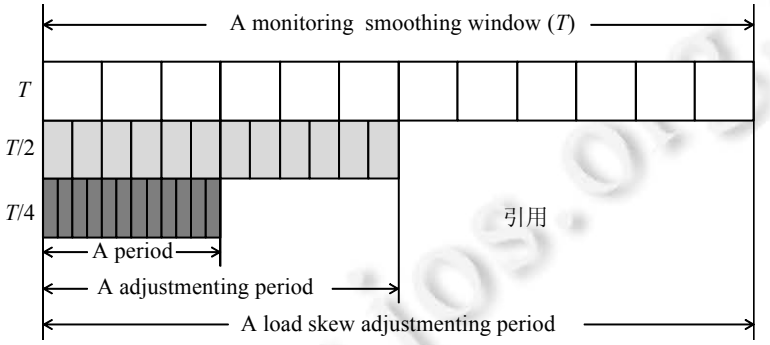

Fig.5 In a sliding window, multiple different lengths of adjustment periods at the same time to identify the FLS

图 5 一个滑动窗口内,同一时刻采用不同长度区间测试模型性能,目的是加快预测速度,提高准确率

2.2 基于预取技术的缓存替换算法(PARC)

如图 6 所示,PARC 结构包含 4 个链表,分别为 ghost LFU 链表、ghost LRU 链表、LFU 链表和 LRU 链表.ghost LFU 链表和 ghost LRU 链表用于放置只被访问过一次的元数据项,LFU 链表和 LRU 链表用于放置被访问两次或者两次以上的元数据项,客户端访问请求的是 LFU 和 LRU 链表.在 PARC 算法中,预取环节获得的热点元数据项放置于 ghost LFU 链表头部,这是因为在 PARC 算法中,预取的热点元数据项被认为是将来最有可能被访问到的元数据项.当最新的访问请求在 LFU 列表中命中的元数据项与预取的热点元数据项相同时,此热点元数据项由 ghost LFU 链表迁移至 LFU 链表对应位置,等待新一轮访问的到来.LFU 链表与 LRU 链表中的元数据项根据访问请求的变化规律相互间迁移调整.

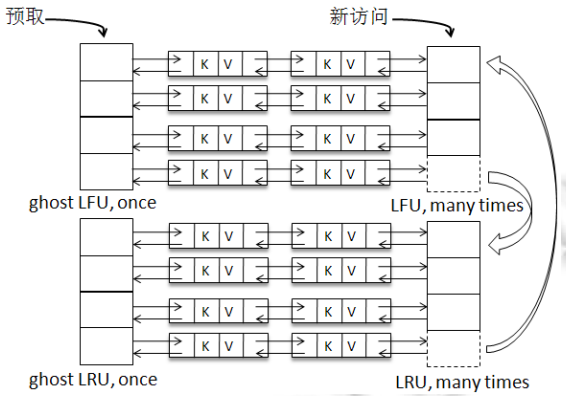

Fig.6 Structure of PARC

图 6 PARC 算法结构图

基于预取的缓存替换算法的含义是:将预取的内容填充到缓存之中,以此提高缓存的前瞻性,减少突发性高负载带来的性能抖动.基于预取机制,缓存替换算法可以监测到引起 FLS 的数据项,同时也可以将无用的数据项提前从缓存中移除.

PARC 中,关键点在于利用预取得到的元数据项填充 ghost LFU 和 ghost LRU.这样的策略可以有效避免错误预取内容对真实缓存空间的直接浪费.除此之外值得注意的是,这是一个可扩展的缓存结构.LFU 和 LRU 的大小作为一个可以调整的变量,可以适应不同的 I/O 负载模式.最终的目标是结合预取与缓存的优点,提高缓存层的性能.正是基于这样的原始目的,采用平均响应时间为度量指标评价 PARC 性能,而不是缓存命中率.TPPS 利用初始化、周期性的调整缓存内容保证缓存层性能.

2.3 可调缓存一致性策略(ACCS)

元数据一致性问题分为两个方面:一方面是元数据服务器中副本之间的同步,另一方面是前端缓存与后端服务器之间的同步.本文的解决方案里,服务器副本之间采用强一致性策略,缓存与服务器之间采用一种可调的一致性策略.如图 7 所示:写操作路径为客户端、元数据服务器;读操作路径为客户端、元数据缓存层、元数据服务器.写操作、常规读操作采用强一致性策略,突发性高负载模式下采用可调一致性策略,可调参数为时间偏差和顺序偏差.

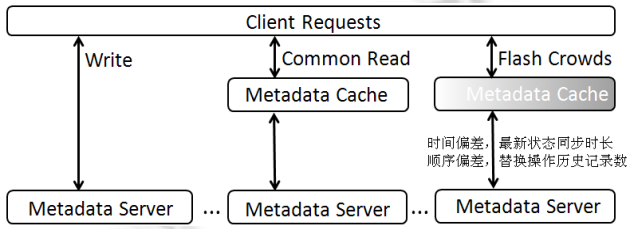

Fig.7 Metadata read and write operation path

图 7 元数据读写操作路径图

本文关注元数据集群负载倾斜问题,将引起负载倾斜的热点元数据划分为两类:

- 第 1 类,依据元数据树形命名空间结构特点,靠近根节点位置的元数据被访问的概率大于远离根节位置的元数据,此类热点元数据属于持久性热点元数据;
- 第 2 类,由客户端突发性高负载模式产生的突发性热点元数据.

缓存第 1 类热点元数据时,采用强一致性策略;缓存第 2 类热点元数据时,采用缓存更新时间偏差与缓存更

新顺序偏差可调的弱一致性策略.

3 系统实现

如图 8 所示,元数据采用树形结构设计实现,缓存热点元数据为 K-V 结构:Key 表示文件路径信息,Value 表示数据块位置信息.

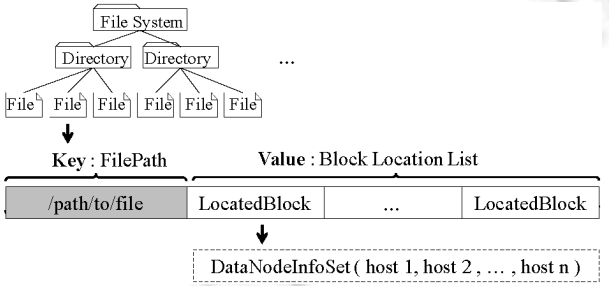

Fig.8 K-V data structure instead of subtree

图 8 采用 K-V 数据结构替代树形数据结构

本文采用全复制策略建立分布式缓存层,采用这个策略的原因有两个:第一,采用分区策略会出现跨节点请求负载,这样的情况会影响缓存层的性能;第二,由于元数据具有尺寸小的特点,热点元数据的体量比数据小得多,单节点的缓存空间可以容纳所有热点元数据.采用分布式缓存技术解决元数据服务器集群请求负载均衡问题,挑战之一是前端缓存内容与后端服务器内容的一致性的问题.经研究发现:大量的云存储应用对一致性要求不高,可以容忍弱一致性协议.以零售业电子商务中图片存储为例,卖家更新一部分商品的图片,并不需要买家实时查看到更新的图片,一段延时是可以被容忍的,比如几分钟甚至是几十分钟.我们的方法主要解决的是这类弱一致性需求的应用.

3.1 TPPS

TPPS 的核心是将一个时间窗口切分为多个时间段,以便框架阶段性分析判断负载变化趋势.能否实时监控并输出性能变化信息是实现 TPPS 算法的前提条件,从实现方式的角度,这里需要对监测手段进行必要说明.为了监测元数据服务器性能变化,我们在元数据集群部署了当前主流的 ganglia 监测系统.此款由加州大学伯克利分校开发的集群性能监控软件非常强大,可以实时地监测集群以及集群的每个节点的各个性能指标.我们解析了 ganglia 的数据结构,从而实现了更细粒度的监测任意时间段的负载参数值,同时,也使得 ganglia 更适合我们的实验方式. TPPS 算法可以根据集群目前的负载倾斜情况自动执行,具体描述见算法 1.

3.2 PARC

PARC 采用全复制策略管理分布式缓存层,这样,客户端可以随机地选择任何一个缓存节点读取热点元数据项.在 FLS 模式中,少量的热点即能引起突发性访问负载,所以缓存的大小并不依赖于整体的数据量,这一点被文献[1]证实.缓存层部署在元数据集群,元数据缓存层与元数据服务层分享集群内存资源.加之元数据本身的特点,即尺寸小.基于以上因素,我们设置每个节点上的元数据缓存大小为 200M.在 PARC 中,我们设置的两个缓存队列存储空间比例为 4:1,这样的比例设置是因为:缓存存储的是 block,预备缓存存储的是 block identifiers,缓存尺寸大于预备缓存尺寸.这样的实现方式可以加速存取效率,提高缓存性能.

3.3 ACCS

ACCS 利用一致性偏差度量体系中的时间偏差与顺序偏差变量实现缓存一致性可调策略,依据监测数据及框架阶段性统计分析结果,实现不同一致性强弱策略的动态切换.其中,时间偏差是指服务器端发生改变的元数据状态同步至前端缓存的时间长度,ACCS 依据突发性高负载模式特点,支持自定义设置时间偏差参数,例如

200ms,500ms,1000ms.顺序偏差是指缓存节点在一定的时间间隔内执行缓存替换操作历史记录数量,ACCS 依据客户端的位置和数量,在一定的时间间隔内,不同客户端读取的缓存版本数量允许不一致,同一客户端的不同会话读取的缓存版本数量也允许不一致,进而实现客户端级与会话级的弱一致性保证机制.

4 实验分析

本节阐述一些实验评价的细节,选取吞吐量、响应时间作为评价 TPPS 与 PARC 性能的指标.选择在一个由 10 个节点组成的元数据集群环境测试本文方法.HDFS 利用多个节点分布式管理元数据,叫做 federation 架构.在这个架构中,每个元数据服务器节点(MDS)负责管理一部分元数据,即,一个命名空间子树.树形的命名空间结构是分布式文件系统元数据的主流数据结构.因为实验目的是测试元数据访问性能,所以 10 个节点均部署为元数据服务器节点.

4.1 实验环境和负载设置

实验环境配置见表 2,实验负载见表 3.采用 HDFS NNbenchmark 作为性能测试基准,为了达到元数据请求高并发负载模式,我们扩展了 NNbenchmark 实现机制,使其支持多线程操作,模拟多客户端访问负载.除此之外,本文以自主研发云存储项目作为测试实体,名为 Cloudshare,一个支持企业文档共享协作业务的云存储平台.

Table 2 Implementation details of the testbed

表 2 实验环境配置情况

Item	Configuration
Hardware	Intel i7-2600 3.4GHz (4-cpu), RAM 16GB, HDD 2TB
OS	Ubuntu-11.04 x86 64
JDK	Hotspot 64-Bit Server VM (build 1.6.0 27-b07)
Hadoop	Version-0.23.3, similar with hadoop2.0.0
Test client	NNbench
Monitor	Ganglia
Network	1000Mbps Ethernet, Catalyst-3750 10GigE switches

Table 3 Details of nnbench work loads

表 3 实验负载分布

Item	Workload-1	Workload-2
Operation type	create write	open read
Request distribution	uniform	hotspot
Job count	1	10
Task count/job	10	10
Thread count	10×100	10×10×100
Smoothing Window (min)	15	15
File count (*1000)	1000	10
Block size (bytes)	1	1

4.2 TPPS的准确性与时效性验证

验证 TPPS 方法的准确性与时效性测试中,循环执行测试用例 20 次,分别记录集群总体吞吐量指标的最大值、最小值和平均值.为了保证测试结果的一般性,在不同长度的时间窗口内运行 TPPS 算法.图 9 是测试结果,横轴代表每个客户端线程的数量,纵轴代表集群的总体吞吐量.每个误差条代表集群总体吞吐量的最大值与最小值,曲线代表集群总体吞吐量的平均值变化情况.其中,一个时间滑动窗口和 1/4 时间滑动窗口的测试结果分别用三角形和圆形曲线表示,两条曲线在横轴“63”处交汇.在“63”前,三角形曲线高于圆形曲线;而在“63”后,三角形曲线低于圆形曲线.这说明 TPPS 算法对负载倾斜频繁度是有敏感性的.实验设计规则是:并发访问数量越多,负载倾斜频繁度越高.在并发数量少、负载倾斜频繁度低的情况下,TPPS 时间窗口长度越长,越有利于预测负载峰值位置,TPPS 性能越高;而在并发数量多、负载倾斜频繁度高的情况下,TPPS 时间窗口长度越短,越有利于预测负载峰值位置,同样,TPPS 性能也越高.所以在为 TPPS 设置不同时间滑动窗口长度时,出现了性能交叉的情况.图 10 是执行预取算法 2 的测试结果,三角形曲线、圆形曲线、方块曲线分别代表在同时执行算法 3 次(其中,

执行时间依次为 1 时间窗口、1/2 时间窗口 1/4 时间窗口)、2 次、1 次(1 时间窗口).测试结果表明:同一时刻执行算法次数越多,则预取的准确率越高.但执行时间的长短是有限度的,预取的代价不能在一个足够小的时间范围内被容忍.根据公式(4),我们同时运行 ln*n* 次算法.

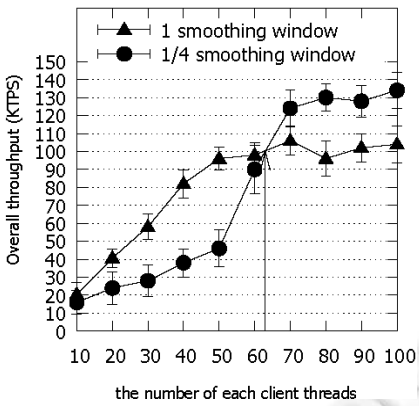

Fig.9 Accuracy comparison of algorithm 1 with different length adjusting periods

图 9 算法 1 不同长度时间内准确率对比

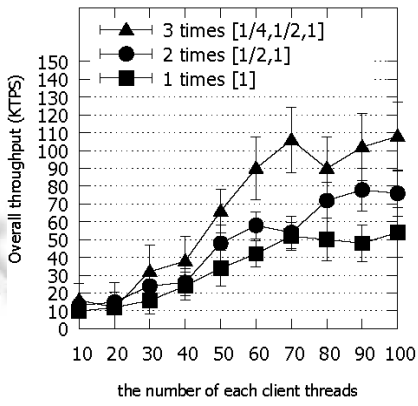

Fig.10 Accuracy comparison of algorithm 2 with different concurrent adjusting times

图 10 算法 2 不同运行次数的准确率对比

综合图 9 图 10 两组实验可以看出,两个预取算法的准确性都很高.对于算法的准确性而言,我们在一个时间滑动窗口同时运行两个算法.测试结果如图 11 所示,算法 2 达到 100KTPS 吞吐量的时间为 8 分钟,而算法 1 则慢了 5 分钟;同时,算法 2 的最大吞吐量为 140KTPS,即,算法 2 在时效性方面优于算法 1.

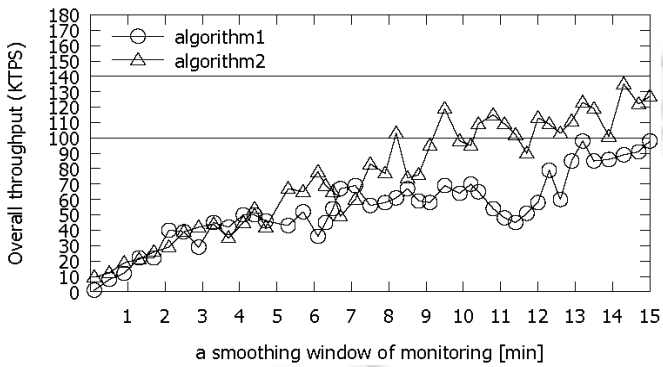

Fig.11 Comparison quickness between two algorithm

图 11 两种算法的时效性对比

4.3 PARC与其他缓存替换算法的性能对比

采用平均响应时间作为对比测量 PARC,ARC,LRU 性能的指标.基于大量的实际测试与参考文献数据,一次常规的元数据请求响应时间为 10ms 以内,所以我们将这个值(10ms)作为判断一个元数据服务器性能是否出现下降的基准线.测试结果如图 12 所示,算法 PARC 基本达到基准线,算法 ARC 差一些,两者的测试差距正是预取算法作用的表现.LRU 算法忽略了请求负载的变化频度,所以从这组实验看,LRU 算法基本是无效的.同时,这组实验也表现了 FLS 负载模式对元数据集群服务的负面影响,如方块曲线.

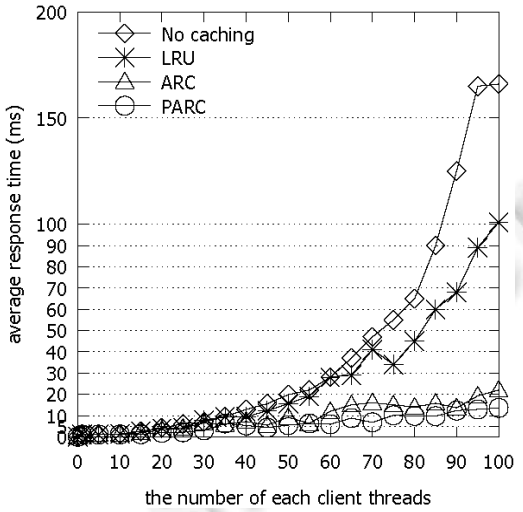

Fig.12 Compare performance of different cache algorithm

图 12 不同缓存替换算法性能对比

4.4 ACCS与强一致性策略的性能对比

缓存一致性策略采用 pull-based 方式实现.一致性协调器负责检测服务器端与缓存端数据一致性,即,同一个元数据缓存内容如果被检测出与服务器不一致,则根据更新时间确定该缓存是否失效:如果大于预设时间间隔,则视为失效缓存;如果小于或等于预设时间间隔,则视为可用缓存.

本组实验设置时间间隔分别为 1s,5s,10s 以及无时间间隔,每个客户端并发线程数量设置为 100,实验运行 20 次,测试结果如图 13 所示.随着允许不一致的时间间隔的减少,元数据平均响应时间越来越长,且呈不稳定态势逐渐明显.

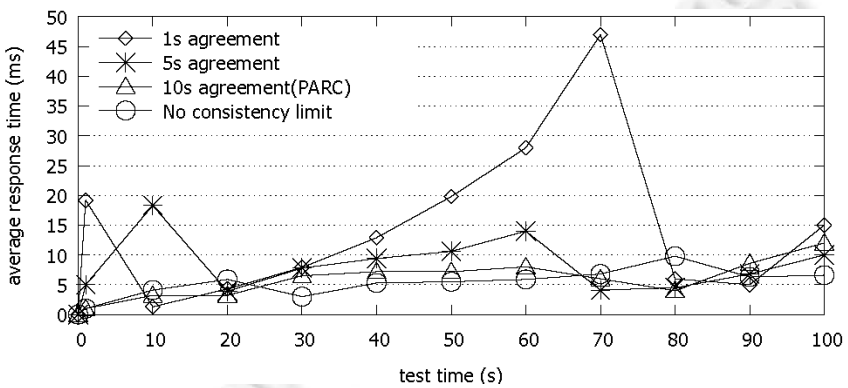

Fig.13 Comparison performance different consistency

图 13 不同一致性策略的性能对比

4.5 本文框架与子树、哈希性能对比

采用总体吞吐量作为对比测量子树、哈希以及我们的框架性能的指标.实验运行 20 次,每个客户端并发线程数量设置为 100.测试结果如图 14 所示:负载均衡效果最差的是子树架构,最好的是我们的两层架构.子树划分的元数据分布式管理架构在 FLS 负载模式中处于低性能的情况,哈希结构中出现的负载不均衡情况是由于同一时刻多个客户端同时访问一个目录或者文件造成的.本文提出的两层架构弥补了上诉两种架构的不足,独立

缓存层可以有效应对对突发性高负载模型引起的 FLS 场景.同时,从实验结果的误差情况看,我们的两层架构的误差范围也小于另外两种架构,这说明两层架构具备很好的稳定性.

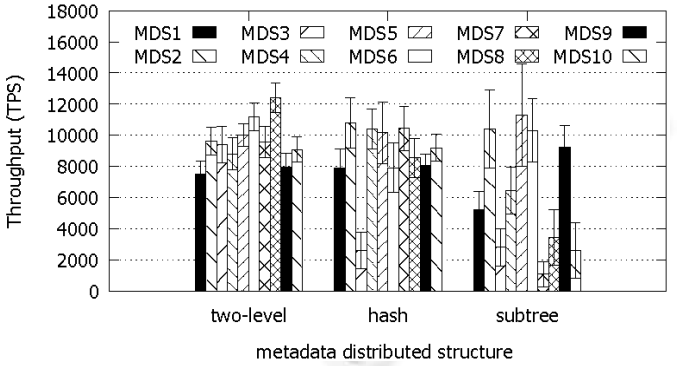

Fig.14 Comparison performance of different structure

图 14 3 种元数据管理架构的性能对比

5 相关工作

5.1 元数据动态负载均衡技术

已有研究工作运用动态调整的思想解决动态负载均衡问题,例如:

- 文献[14]提出的 DROP 采用 0-1knapsack 模型鉴别过载节点和空闲节点,然后执行一个迁移操作,重新平衡集群负载,并且在这个过程中保证元数据的目录关系不被破坏.该方法本质上依靠存储的负载均衡带动请求的负载均衡,同时也用到了缓存技术;
- 文献[15]提出在集群前端放置一个小体积的缓存专门管理热点,从而达到均衡全局访问负载的目的.该工作主要证实了一个观点:前端缓存大小与数据量大小无关,而与集群节点数量相关.一系列的验证实验都是围绕对抗性负载模式展开的,但是此方案不是针对元数据特点设计实现的;
- 另一个架构是文献[3]提出的两层无中心元数据管理模型,用一个全局副本策略管理最近被访问到的元数据.本质上,这是一个用准确率换取性能的方法.

以上分析的几个代表性的相关工作从 3 个主流的方面给出了解决动态负载均衡的思路,分别是再分布、缓存以及多副本技术.

本文方法同样用到了缓存、副本技术,但不同之处是:建立了一个完全独立的分布式缓存层,专门管理热点;同时,首次提出利用预取技术提高元数据缓存的性能.方案中的缓存层可以部署在不同的位置,适用于不同的系统.方法可以灵活应用于不同的 SLOs,在面对负载倾斜频繁变化问题时灵活度更高.

5.2 预取与缓存结合技术

预取技术最初是被用来解决 IO 延时问题[16]的,本文的目的是利用预取技术提高特殊场景中缓存的性能.预取与缓存结合的方法被广泛研究[16−18],文献[12]中,结合多种缓存替换算法,分析了预取的作用,得出的结论是,预取技术可以有效提高缓存性能.从文献[12]我们也受到启发,选择 ARC 算法作为基础算法.文献[16,18]支持此观点,不过他们提出,错误的预取会浪费缓存资源.基于这个观点,他们设计了一系列的规则,尽量降低错误预取对缓存性能的影响.

与以上工作相比,本文方法是基于概率的策略集成预取和缓存技术.这是缘于元数据的特点——预取错误的元数据带来的代价远小于预取错误的数据带来的代价.同时,ARC 算法[19]中的多队列机制也有利于我们调整预取对缓存的影响.

5.3 元数据一致性维护技术

元数据是新型分布式文件系统数据访问模型的关键组成部分,元数据层的强一致性,是系统准确实现数据副本一致性策略的基础.已有研究工作从如何降低元数据的强一致性维护成本方面展开了广泛的研究[4,20],文献[4]中,从减少两阶段提交协议中不必要的日志写入次数以及消息传递次数来减少强一致性开销.文献[20]观察到元数据分布式事务中子操作可以并发执行,中间结果可延时、批量提交等特点,提出一种面向实际应用的一致性放松协议,保持系统可扩展性的同时提高系统元数据服务性能.文献[21]同样采用缓存弱一致性技术提升元数据服务性能,客户端缓存的一致性维护机制是租约,不同时间长度的租约构成了一组弱一致性策略,此策略可以有效避免大量客户端缓存强一致性维护带来的性能开销.然而,此一致性策略应用范围方面存在一定的局限性,负载倾斜频繁变化时,即,热点位置频繁变化时,缓存一致性租约长度不能及时调整.ACCS 采用实时监测与阶段性统计的手段,可以感知负载倾斜变化,可以及时调整缓存一致性租约长度.文献[9]同样仅对读操作放松一致性要求,对于写操作仍然采用强一致性策略,将客户端读操作的一致性控制分为严格顺序一致、新旧偏差一致以及客户端自定义规则.ACCS 与文献[9]的区别是一致性控制粒度可以细化到会话级,这个区别对于突发性高负载模式有不可忽视的作用.本文针对突发性高负载模式,研究管理热点元数据过程中的一致性维护问题,首要考虑的是分布式元数据服务的性能指标.通过对元数据进行分类,将强一致性管理集中在写操作与后端服务器中实现,前端缓存采用了动态可调的弱一致性策略提高性能.

6 结束语

本文首先在元数据分布式管理框架之上设计实现了分布式缓存层,以有效应对潜在的各类热点以及突发性高负载模式.然后,我们提出了新的预取方法 TPPS 与新的缓存替换算法 PARC.在 FLS 场景中,预取技术与缓存技术的结合,有效地提高了缓存层的性能,在一致性方面考虑了时间偏差与顺序偏差因素对缓存性能的影响.最后,通过在 HDFS 元数据集群环境中的一系列实验结果,证明了本文提出框架、方法的有效性.

References:

[1] Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. In: Proc. of the IEEE Mass Storage Systems and Technologies (MSST). Incline Village, 2010. 1−10.

[2] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Proc. of the SOSP. 2003.

[3] Weil SA, Pollack KT, Brandt SA, Miller EL. Dynamic metadata management for petabyte-scale file systems. In: Proc. of the SC. 2004. [doi: 10.1109/SC.2004.22]

[4] Xiong J, Hu YM, Li GJ, Tang RF, Fan ZH. Metadata distribution and consistency techniques for large-scale cluster file systems. TPDS, 2011,22(5):803−816. [doi: 10.1109/TPDS.2010.154]

[5] Weil SA, Pollack KT, Brandt SA, Miller EL. Dynamic metadata management for petabyte-scale file systems. In: Proc. of the SC. IEEE, 2004. [doi: 10.1109/SC.2004.22]

[6] Beaver D, Kumar S, Li HC, Sobel J, Vajgel P. Finding a needle in Haystack: Facebook' sphotostorage. In: Proc. of the OSDI. 2010.

[7] Zhu YF, Jiang H, Wang J, Xian F. HBA: Distributed metadata management for large cluster-based storage systems. TPDS, 2008, 19(6):750−763. [doi: 10.1109/TPDS.2007.70788]

[8] Ari I, Hong B, Miller EL, Brandt SA, Long DDE. Managing flash crowds on the Internet. In: Proc. of the MASCOTS. 2003. 246−249. [doi: 10.1109/MASCOT.2003.1240667]

[9] CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In: Proc. of the FAST. 2015.

[10] Roselli D, Lorch JB, Anderson TE. A comparison of file system workloads. In: Proc. of the USENIX Technical Conf. 2000. 41−54.

[11] KVShvachko. HDFS scalability: The limits to growth. Usenix35, 2010,35(2):6−16. www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf

[12] Xu QQ, Arumugam RV, Yong KL, Mahadevan S. DROP: Facilitating distributed metadata management in EB-scale storage systems. In: Proc. of the MSST. 2013.

[13] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introducetion to Algorithms. 3rd ed., Edinburgh University Press, 2011.

[14] Xu QQ, Arumugam RV, Yong KL, Mahadevan S. DROP: Facilitating distributed metadata management in EB-scale storage systems. In: Proc. of the MSST. IEEE, 2013.

[15] Fan B, Lim H, Andersen DG, Kaminsky M. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In: Proc. of the SOCC. 2011. [doi: 10.1145/2038916.2038939]

[16] Cao P, Felten EW, Karlin AR, Li K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. ACM Trans. on Computer Systems, 1996,14(4):311−343. [doi: 10.1145/235543.235544]

[17] Butt AR, Gniady C, Hu YC. The performance impact of kernel prefetching on buffer cache replacement algorithms. In: Proc. of the SIGMETRICS. 2005. [doi: 10.1145/1064212.1064231]

[18] Teng WG, Chang CY, Chen MS. Integrating Web caching and Web prefetching in client-side proxies. IEEE Trans. on Parallel and Distributed Systems, 2005,16(5):444 −455. [doi: 10.1109/TPDS.2005.56]

[19] Megiddo N, Modha DS. ARC: A self-tuning, low overhead replacement cache. In: Proc. of the FAST. 2003.

[20] Yi LT, Shu JW, Ou JX, Zhao Y. Cx: Concurrent execution for the cross-server operations in a distributed file system. In: Proc. of the MSST. 2012. [doi: 10.1109/CLUSTER.2012.65]

[21] BIndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In: Proc. of the SC. Bestpaper, 2014. [doi: 10.1109/SC.2014.25]

[22] Nicolae B, Moise D, Antoniu G, Bouge L, Dorier M. BlobSeer: Bringing high throughput under heavy concurrency to hadoop map/reduce applications. In: Proc. of the IPDPS. 2010. 1−11. [doi: 10.1109/IPDPS.2010.5470433]

[23] Babaioff M, Immorlica N, Kleinberg R. Matroids, secretary problems, and online mechanisms. In: Proc. of the SODA. 2007. 434−443.

[24] Chen T, Xiao N, Liu F. Adaptive metadata load balancing for object storage systems. Ruan Jian Xue Bao/Journal of Software, 2013,24(2):331−342 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4177.htm [doi: 10.3724/SP.J.1001.2013.04177]

附中文参考文献:

[24] 陈涛,肖侬,刘芳.对象存储系统中自适应的元数据负载均衡机制.软件学报,2013,24(2):331−342. http://www.jos.org.cn/1000-9825/4177.htm [doi: 10.3724/SP.J.1001.2013.04177]

孙耀(1982－),男,吉林桦甸人,博士生,主要研究领域为网格分布式计算,软件工程,大数据存储.

叶丹(1971－),女,博士,研究员级高工,博士生导师,CCF 高级会员,主要研究领域为网络分布式计算,软件工程.

刘杰(1982－),男,博士,副研究员,CCF 专业会员,主要研究领域为网络分布式计算,软件工程.

钟华(1971－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为网络分布式计算,软件工程.