·计算机工程与应用·

MUSE：一种面向云存储系统的高性能元数据存储引擎

段翰聪，向小可，吕鹏程

(电子科技大学计算机科学与工程学院 成都 611731)

【摘要】该文设计了一种高性能的面向云存储系统的元数据存储引擎(MUSE)。首先，其底层物理存储模块采用LSM-tree模型的高速key-value存储引擎LevelDB方案，通过设计多缓存表和多线程紧凑机制对该方案进行优化，使其可以充分利用内存和多核CPU并行能力；其次，提出了基于多I/O通道的元数据存取调度机制。通道之间读写操作隔离，聚合多个通道为上层提供高并发随机I/O读写能力；此外，针对上层目录命名空间管理，提出路径分割映射和全路径映射策略两种策略，可基于不同的应用场景在性能与可用性间进行折中选择。系统测试结果表明，MUSE能够很好地适应海量小文件存储场景，相对于其他元数据存储系统在性能上有显著的提升。

关 键 词 I/O; LSM-tree; 海量; 元数据; 性能; 小文件; 存储引擎

中图分类号 TP311 **文献标志码** A doi:10.3969/j.issn.1001-0548.2016.03.011

MUSE: A High-Performance Metadata Storage Engine for Cloud Storage System

DUAN Han-cong, XIANG Xiao-ke, and LÜ Peng-cheng

(School of Computer Science and Engineering, University of Electronic Science and Technology of China Chengdu 611731)

Abstract In the cloud storage systems, the accesses of massive small files metadata will generate a large number of random disk I/O requests, which will become the performance bottleneck of the entire storage system. In this paper, metadata unit storage engine (MUSE), a kind of metadata storage engine for cloud storage system, is proposed to support massive small files storage with high performance. Firstly, LevelDB, a high speed key-value storage engine based on LSM-tree, is used as underlying physical storage module. Secondly, LevelDB is enhanced by introducing multiple buffer tables and multiple compaction threads, which take full advantages of memory and multi-core processor. Thirdly, a new metadata accesses scheduling mechanism on multiple I/O channels is proposed. Channel is an independent data storage pipe formed by binding the independent thread to the independent physical disk. In this way, the access operations are isolated between channels, and then the aggregation of multiple channels can provide high concurrency random I/O. In addition, MUSE proposes two namespace management strategies: Split-path mapping strategy and absolute path mapping strategy, aimed to make trade-off according to different application scenarios by users. Benchmarks show that MUSE can support the massive small files storage scene and outperform other metadata storage systems.

Key words I/O; LSM-tree; massive; metadata; performance; small file; storage engine

随着互联网服务数量和数据量的激增，需要存储的非结构化数据量也在快速增长。传统的存储方案不能满足海量文件服务，因此可扩展的分布式存储系统应运而生，但是元数据的存储策略一直是分布式存储系统性能的瓶颈。

从对磁盘介质的分析看出，磁盘最适合顺序的大文件I/O方式，而不适合随机的小文件I/O方式，然而目前大多数的分布式存储系统的元数据组织都是针对大文件存储和可靠性[1]进行设计的，因此基于海量小文件的应用在性能和存储效率方面要大幅降低，甚至无法工作。以下是目前存储系统在海量小文件情况下元数据管理存在的问题：

1) 由于文件的inodes在磁盘中随机存储，访问文件时，路径中所有目录分量的inodes都需要被读

收稿日期：2014 – 04 – 18；修回日期：2015 – 09 – 10
基金项目：国家重大科技专项(2012ZX03002-004-004)
作者简介：段翰聪(1973 –)，男，博士，副教授，主要从事P2P内容分发网络、分布式存储、操作系统等方面的研究.

取，这将生成大量的磁盘随机I/O。

2) 存储系统元数据管理通常采用hash、B+树来组织索引目录，这种方法在目录存在大量文件情况下检索效率明显下降，无法做到有效的扩展。

3) 本地文件系统使用块来组织磁盘数据，当数据内容尺寸小于数据块的小文件时，造成大量磁盘空间浪费、缓存利用率极低。

4) 随着计算机硬件的发展，多CPU核心和多存储设备的机器越来越流行，而当前的分布式存储系统并没有充分利用多CPU核心和存储设备的性能。

本文设计和实现了一种存储系统元数据存储引擎来解决上面的问题，试图在数据存储体系上提高数据访问的性能，从而适应海量小文件的存储场景。

1 相关工作

存储系统元数据存储已经有很长的研究历史，其中一些元数据管理的工作是非常有代表性的。

1.1 分布式文件系统元数据管理模型

GFS[2]和HDFS[3]的优点在于大文件的读操作、续写操作的性能很高。但是，为了不使元数据服务器磁盘I/O成为系统瓶颈，二者都将元数据全部保持在内存，而这使得内存大小成为整个系统扩展的瓶颈。并行文件系统如PVFS[4]支持条带化数据分布来获取高传输速度并减小文件元数据的大小，同时PVFS使用BerkeleyDB做元数据服务器的底层存储引擎，而BerkeleyDB采用B+tree作为索引模型；总之，在海量数据处理中，传统的存储方式将导致非常低的读写性能。

1.2 数据存储模型

针对海量小文件场景，ops/sec是衡量元数据存储系统性能最关键的因素，因此可以借鉴数据库的索引组织方式。

关于单机数据存储索引模型，最为经典的是B+树模型，它通过降低树的层次来减少I/O次数从而提高I/O效率。在B+树中顺序插入大量条目，速度非常快，因为每次都会在最后一个节点上顺序插入；如果有随机的插入、更新、删除等，会出现大量随机I/O，每当更新一个条目，则该条目所在的节点上的其他条目同样会被I/O，因此在上层随机读取或更新文件时会造成大量无效的随机I/O操作。

LSM-tree[5]模型使用基于内存的缓冲区来缓存更新操作，并且更新操作都被当做一个标记条目插入内存，直到某个阈值后批量刷入磁盘，在磁盘文件到达一定阈值后进行紧凑操作：多个旧文件的条目整合，之后按序生成一个新文件到磁盘。批量写策略使得它比B+树更加高效。越来越多的数据库使用LSM-tree做索引，其中最著名的分布式系统有BigTable[6]、Cassandra[7]和LevelDB[8]。LevelDB是google公司开发的key-value存储库。LevelDB在LSM-tree基础上做了很多创新来增强读性能，如引入层次机制，将缓存的更新条目持久化到磁盘，而磁盘数据通过紧凑算法依次从低层刷入高层，因此层次越高生成的文件数目越少，数据越老。通过层次机制可以限制生成的数目，并且加速读操作。同时通过使用Bloom Filter[9]来减少需要搜索持久化文件的数量。但是如何能结合现代机器多核多盘的优势对LSM-tree模型进行改进，是一个挑战。

Fractal-tree模型是Buffer-tree[10]的一个变种，它同样将更新操作看做消息存储到相应节点的相应消息缓冲区：即更新时候将消息放入root节点之后即完成更新操作，当root节点相应的消息缓冲区满后，消息被自上而下逐步下刷和整理到叶子节点。但是目前该模型的工程实现尚不成熟，基于Fractal-tree做索引的产品只有TokuDB[11-13]。

2 设计与实现

2.1 系统综述

本文所述的元数据存储引擎来自于自行研发的分布式文件系统cloud store(CSTORE)。整个系统由数据存储服务器、元数据存储服务器、规则服务器、客户端4个部分组成。客户端通过FUSE向用户提供近似POSIX文件系统接口，访问文件时客户端先从元数据服务器MUSE读取文件元数据，然后根据文件元数据信息从数据服务器中获取具体的文件数据，而规则服务器负责数据服务器集群的负载均衡。

单机存储模型层次如图1所示，整个元数据存储引擎分为3层：1) Key-Value存储引擎层；2) 命名空间管理层；3) 通道处理层。当网络I/O层接收到文件访问请求，会将请求下发给通道处理层；通道处理层根据一定策略将请求分发给特定管道；特定管道接收到请求后对请求的路径名进行解析，从而定位到文件Inode对应的key下发给key-value存储引擎层；最后key-value存储引擎层通过key读取相应的value从而找到该文件的元数据。

图1 单机存储模型层次

2.2 Key-Value存储引擎层

由于LevelDB数据索引采用LSM-tree模型，相对于其他模型有很大的性能优势，因此选用LevelDB做底层物理存储引擎。为了达到极限性能，在LevelDB原有的基础上作进一步的优化。由于LevelDB同一时间只允许有一个缓存表提供服务，在高速插入条目的情况下，缓存表插入速度与紧凑操作速度相差太大会导致缓存表溢出但还来不及持久化，此时LevelDB会睡眠导致整体性能下降。本文采用开辟多个缓存表来解决这个问题。另外，LevelDB只启用一个线程进行紧凑操作，因此在LevelDB全负荷运行时最多只能利用2个CPU核心，因此将紧凑操作改为多线程，充分利用多核CPU。通过在LevelDB的基础上进行改进后形成的key-value存储引擎作为元数据的本地存储能将元数据的更新操作转换为大量结构化数据的插入，这极大地减少了磁盘寻道时间。

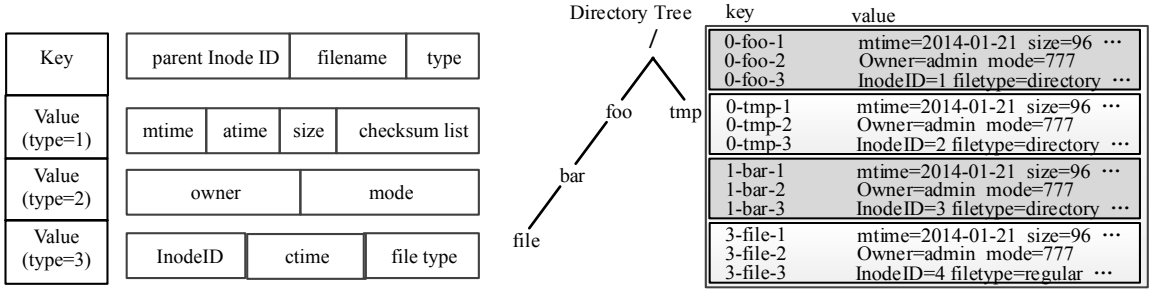
2.3 命名空间管理层

2.3.1 路径分割映射

一般通用的场景下，文件的新建、更新、删除、修改文件名、目录名、目录列表都需要得到良好高效的支持，使用LevelDB条目的value域来记录文件或目录的元数据。普通文件元数据信息包括：名称、版本、权限、创建时间、修改时间、类型、大小和文件片信息；目录元数据信息主要包括：目录名称、权限、创建时间、修改时间、大小。文件名称直接反映为用户命名空间显示的文件名称。文件类型主要用于区分普通文件和目录。文件块信息是一个列表，存储了该文件所有的文件块信息。每个表项包含这个文件块的SHA-1校验值，使用这个校验值可以定位到存储这个块的数据服务器。

为了加快目录列表操作，利用LevelDB的key全局排序特性，路径分割映射策略key、value格式如图2a所示，将条目的Key格式化为3部分：父目录Inode号、文件名称、类型。依次解释这3个字段的用意为：1) 使用父目录Inode号作为第一部分意味着同一目录下的条目在逻辑上会被连续存储。2) 第二部分直接设置为文件名称而非文件Inode号，因为在同一目录下不可能出现文件名相同的多个文件，因此在同一目录下文件名可以唯一指示一个文件，更重要的是当用文件名代替Inode号，可以在列表操作中减少一次路径分量解析的过程，从而加速列表操作。3) 考虑到一些文件或目录的部分属性被会比另一部分属性更加频繁的被更新，如修改时间、访问时间和文件大小被修改的次数远远大于创建时间、拥有者、访问权限等属性，因此将第三部分设置为类型字段，类型=1代表需要被频繁修改的属性；类型=2代表很少被修改的属性；类型=3代表永远不会被修改的属性。这样可以减少每次元数据更新时的I/O数据量以及日志记录的数据量。

路径分割映射策略如图2b所示。读取路径为“/foo/bar/file”的文件的元数据时，首先会从已知的Inode号为0的root目录“/”开始逐个解析路径分量：遍历key第一部分为“0”的key-value对，直到找到第二部分为“bar”的key-value对，则在其value中读取“/foo/bar”的Inode号为1，再遍历key第一部分为“1”的key-value对，直到找到第二部分为“file”的key-value对，即可得到“/foo/bar/file”的元数据信息。经过测试发现，这种路径分割映射方式在一般通用的场景会比传统的文件系统在性能上有很大的提高。



a. 路径分割映射策略key、value格式　　b. 路径分割映射策略示例

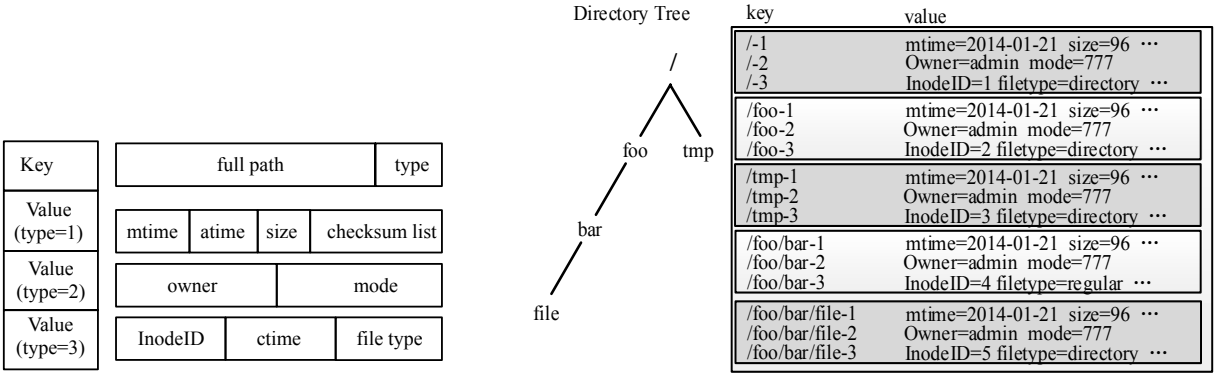
图2　路径分割映射方式的命名空间管理

2.3.2 全路径映射

当文件所在的路径层次很深时，海量文件处理中路径分量的解析工作会相当繁重，本文提出了另一种路径映射方法：全路径映射方法。a 全路径映射策略key、value格式如图3a所示，将条目的Key格式化为2部分：文件的完整路径和类型。依次解释这

2个字段的用意为：1) 使用文件的完整路径作为Key的第一部分使得可以仅仅访问LevelDB一次即可得到相应文件或目录的元数据，使得可以完全抛弃路径分量解析的繁重工作。2) 使用类型字段作为第二部分的用意与前面路径分割映射方式的意图完全相同。条目Value的值同样与前面路径分割映射方式的完全相同。

全路径映射策略如图3b所示，在读取路径为”/foo/bar/file”的文件的频繁修改的元数据，则直接可以读取Key为”/foo/bar/file-1”的Value值即可。但是当遇到更改某一目录的文件名操作的请求时，由于该目录下所有条目的key中都存储了该目录的旧目录名，因此该目录下的所有条目的key都需要被更新，这无疑是全路径映射的一个缺点。因此本文建议在目录深度超过3时需要超高性能且不存在频繁更改目录文件名的场景中使用全路径映射方式，而在一般通用的场景中使用路径分割映射方式来部署元数据服务器。在本文的系统中默认使用路径分割映射方式，用户可以在配置文件中修改为全路径映射方式。



a. 全路径映射策略key、value格式　　b. 全路径映射策略示例

图3　全路径映射方式的命名空间管理

2.4　通道处理层

CPU和磁盘带宽最容易成为元数据服务器的性能瓶颈，目前很多已存在的系统在设计上并没有充分利用多核CPU和多磁盘带宽，如对数据存储位置没有太多的考量而直接存储在磁盘的逻辑分区，这无疑加重了磁盘的随机I/O而导致磁盘性能下降。

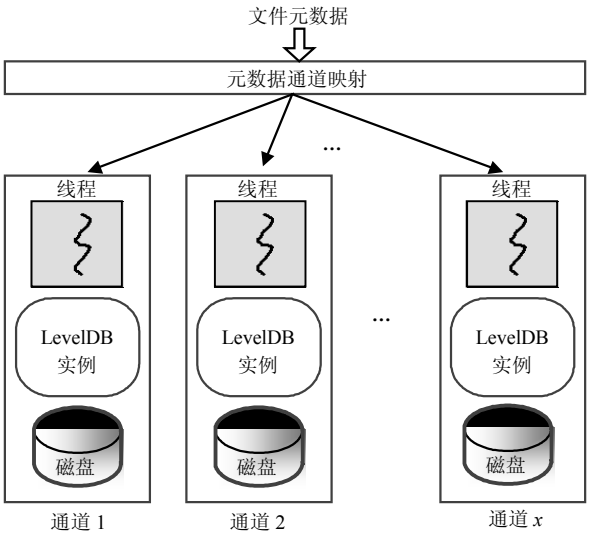


图4　引入通道的文件元数据存储

本文提出通道机制来解决这个问题。通道是指将独立的CPU核心、独立的物理磁盘进行绑定而形成的独立数据存储管道。每个通道可以独立进行数据I/O，多个I/O通道的读写相互隔离。因此，通道中的I/O请求不受其它请求的影响。引入通道机制后，可以在每个通道上只部署单一底层存储引擎，最大程度减小磁盘的随机I/O。通道方案支持针对服务器的不同硬件配置信息来指定多个通道共同服务。

当引入了多通道，随之而来的问题就是通道间的负载均衡如何保证。这里采用Round-Robin调度算法[14]，将到来的新建文件元数据请求均匀的分散到各个通道，平衡各通道压力。

3　评估测试

通过一系列的测试用例来评估元数据存储引擎的性能。该引擎将在多个元数据服务器多磁盘环境下进行测试。测试条件如下：元数据服务器MUSE、规则服务器、客户端均运行在由8台服务器通过GigE NIC相连接而组成的集群中。每台服务器由16核心CPU、8块15 000 RPM的磁盘、16 GB内存组成，使用XFS作为本地文件系统。

测试程序使用的是业界常用的测试文件系统的性能工具IOMeter[15]。

按照如下顺序进行测试：1) 单通道下两种不

同命名空间管理方式的性能；2) 多通道下MUSE的性能；3) 与其他元数据存储系统的性能进行比较。

3.1 两种不同命名空间管理方式的性能对比

首先测试单通道元数据服务器的性能。本文分别测试全路径映射命名空间管理方式和路径分割的命名空间管理方式的性能。该项测试统一向单通道服务器上传和下载目录深度为5的0大小的文件，因为向分布式文件系统CSTORE上传0字节文件，客户端只会与元数据服务器交互，而不会与数据服务器交互。理论上路径分割的命名空间管理方式每访问一个文件至少需要5次磁盘I/O，而全路径映射命名空间管理方式只需要一次磁盘I/O，两种不同命名空间管理方式的性能如图5所示，可看出全路径映射命名空间管理方式性能是路径分割命名空间管理方式性能的3倍。由于大量目录第一次被访问后被存储引擎缓存，因此两者的性能相差只有3倍而非5倍，由此可知，当深度更大时，通过路径分割的命名空间管理访问一个文件需要更多的访问次数，更会加大与全路径映射命名空间管理方式之间的性能差距。

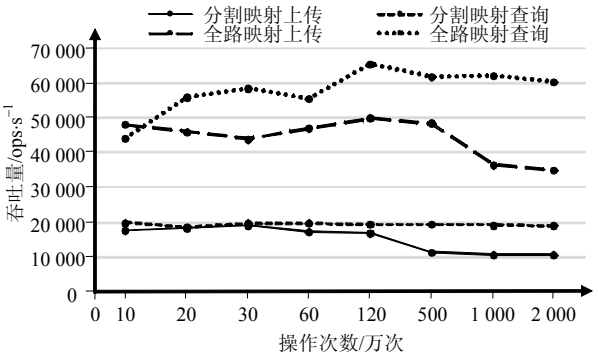
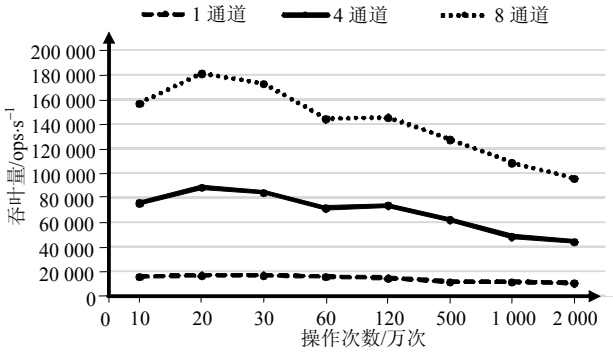


图5　两种不同命名空间管理方式的性能

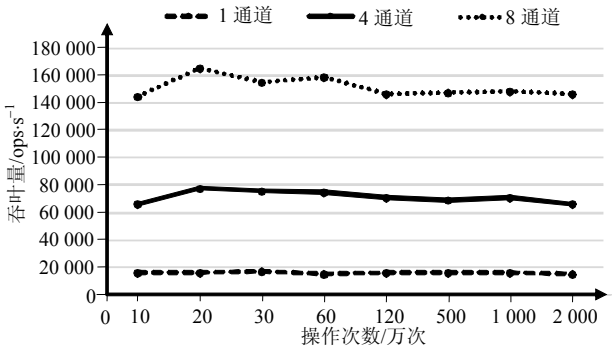
3.2 多通道扩展性和性能

本项测试使用多通道的方式来测量通道的扩展性。在这里使用与前面相同的测试用例来测试4通道和8通道情况，并与前面的单通道性能做比较。

多通道MUSE的性能如图6所示，可看出在相同文件数量的情况下，4通道每秒元数据操作数超过了单通道每秒操作数的4倍，这是因为4通道测试中每个通道只分到总文件量的四分之一的工作负载；8通道情况与4通道情况类似，只是随着文件数量的递增，性能开始缓慢下降到单通道性能8倍以下，这是由于文件数量到达一定很高的阈值，元数据向多通道映射造成了性能瓶颈。总之，这个结果论证了单元数据服务器下，多通道的性能基本可以达到线性扩展。



a. 多通道MUSE上传性能



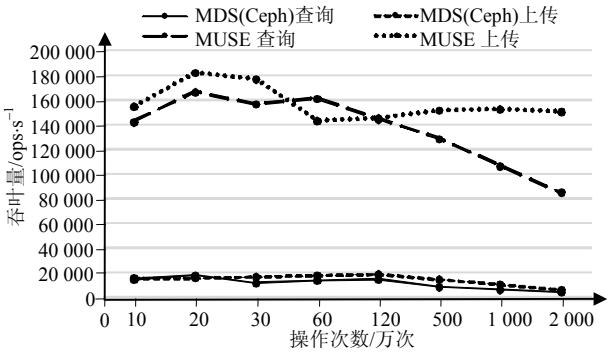
b. 多通道MUSE查询性能

图6　多通道MUSE的性能

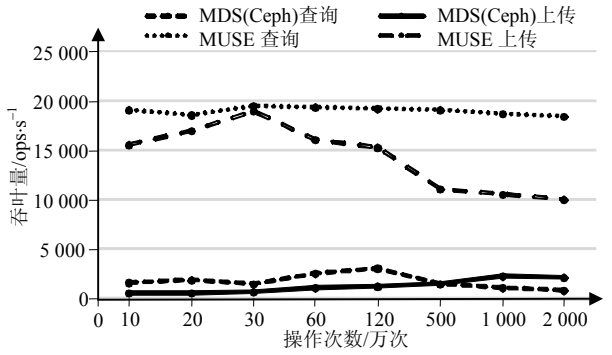
3.3 与其他元存储系统的性能对比

选择PVFS的元数据服务器manager node(MGR)来与本文的MUSE做对比，因为PVFS是一个经过实践检验的存储系统并且有独立的元数据服务器。因此在相同环境下测试MGR(集群)。MUSE与MGR的性能对比如图7a所示，随着文件数量的增加，两个系统的性能均出现下降，但是MUSE性能依然远超MGR。这是因为PVFS过分依赖本地文件系统，随着文件数量的增多而导致性能急剧下降。而MUSE采用LSM-tree模型做数据索引，相对于MGR采用的B+tree模型(MGR使用BerkeleyDB做索引)有很大的性能提升。

最后，由于LevelDB在Ceph最近的版本中被使用做对象存储模型，因此同样部署了Ceph的元数据服务器metadata server(MDS，CEPH的元数据服务器)与本文的MUSE做比较(单节点)。MUSE与MDS的性能对比如图7b所示，MDS在120万文件访问操作时候达到性能最高峰，之后性能下降。虽然MUSE的文件上传性能下降较严重，但是依然远超MDS。这是因为MDS为了保证强一致性而导致性能低下，而MUSE在LevelDB的基础上做了进一步的优化，而且通过引入多通道来充分利用多核多磁盘服务器的性能。



a. MUSE与MGR的性能对比(集群)



b. MUSE与MDS的性能对比(单节点)

图7 其他元数据存储系统性能对比

4 结 论

海量小文件的元数据管理一直是存储领域的一个难点。使用LSM-tree模型代替传统的B+tree等模型作为元数据存储索引。多缓存表、多线程紧凑操作、多通道可以更大程度上利用现代机器大内存、多CPU、多磁盘的硬件优势；路径分割映射的命名空间管理方式最大程度上满足了一般通用场景的性能需求，相对的，全路径映射的命名空间管理方式则在特定的极少更改目录名的场景下对命名空间的性能做了极限提升。性能测试表明，在多台元数据服务器和多磁盘的条件下，MUSE在元数据访问性能方面远超其他元数据存储系统。在未来的工作中，将在保证MUSE的性能的前提下，致力于公有云存储元数据安全性的问题。

参 考 文 献

[1] ZHANG L, ZHU L G, ZENG S F. Metadata update strategy with high reliability[J]. Applied Mechanics and Materials, 2013(411): 382-385.

[2] GHEMAWAT B S, GOBIOFF H, LEUNG S. (2003) The google file system[C]//ACM SIGOPS Operating Systems Review. Indianapolis, USA: [s.n.], 2010.

[3] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]//2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). Incline Village, Nevada, USA: IEEE, 2010:1-10.

[4] CARNS P H, LII W B L, ROSS R B, et al. PVFS: a parallel file system for Linux clusters[C]//Proceedings of the 4th Annual Linux Showcase Conference. Atlanta, Georgia, USA: 2000: 391-430.

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

[6] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: a distributed storage system for structured data[J]. Proceedings of Usenix Symposium on Operating Systems Design & Implementation, 2006, 26(2): 205-218.

[7] LAKSHMAN A, MALIK P. Cassandra-a decentralized structured storage system[J]. ACM Sigops Operating Systems Review, 2010, 44(2): 35-40.

[8] SANJAY G, JEFF D. LevelDB: a fast and lightweight key/value database library[EB/OL]. [2012-12-28]. http://code.google.com/p/leveldb/.

[9] BlOOM B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426.

[10] ARGE L. The buffer tree: a technique for designing batched external data structures[J]. Algorithmica, 2003, 37(37): 1-24.

[11] PERCONA INC. TokuDB[EB/OL]. (2011-1-1). https://www.percona.com/software/mysql-database/percona-tokudb.

[12] ESMET J, BENDER M A, FARACH C M, et al. The TokuFS streaming file system[C]//Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems. Boston, USA: USENIX Association, 2012: 14.

[13] BENDER M A, FARACH C M, FINEMAN J T, et al. Cache-oblivious streaming B-trees[C]//Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures - SPAA '07. Santorini, Greece: ACM, 2007: 81-92.

[14] SHREEDHAR M, VARGHESE G. Efficient fair queuing using deficit round-robin[J]. IEEE/ACM Transactions on Networking, 1996, 4(3): 375-385.

[15] DANIEL S. IOMeter: an I/O subsystem measurement and characterization tool for single and clustered systems [EB/OL]. (2003-2-1). http:// www.iometer.org/.

编 辑 叶 芳