软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(5):1302−1321 [doi: 10.13328/j.cnki.jos.006234]

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

一种监控系统的链路跟踪型日志数据的存储设计*

尤 勇[1], 汪 浩[2,3], 任 天[1], 顾胜晖[2,3], 孙佳林[1]

[1](美团点评,上海 200335)

[2](南京大学 软件学院,江苏 南京 210023)

[3](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 孙佳林, E-mail: jialin.sun@meituan.com

摘 要: 随着软件系统越来越复杂化和分布化,为系统提供具有完善功能的监控服务显得越来越重要.APM (application performance management)系统通过采集软件系统运行时的各项指标数据来分析软件的运行状态,例如CPU、内存使用率、垃圾回收的耗时、QPS 等指标.此外,APM 系统也会在软件运行中生成各种日志数据.通常来说,它能提供的监控数据分为 3 种:指标统计数据、链路跟踪(tracing)数据以及离散事件记录.这些数据有助于系统或者服务的维护人员理解运行状态,从而确保系统或者服务的稳定运行.基于开源的 APM 监控系统——CAT 系统,提出了一种针对 tracing 类型数据的存储设计方案,通过内存块批量写入的方式提升存储效率,并设计了两级索引以提高查询效率.从线上的真实运行数据来看,该方案在写入性能和查询性能方面均有较好的表现.

关键词: 监控系统;日志存储;两级索引

中图法分类号: TP311

中文引用格式: 尤勇,汪浩,任天,顾胜晖,孙佳林.一种监控系统的链路跟踪型日志数据的存储设计.软件学报,2021,32(5): 1302−1321. http://www.jos.org.cn/1000-9825/6234.htm

英文引用格式: You Y, Wang H, Ren T, Gu SH, Sun JL. Storage design of tracing-logs for application performance management system. Ruan Jian Xue Bao/Journal of Software, 2021,32(5):1302−1321 (in Chinese). http://www.jos.org.cn/1000-9825/6234.htm

Storage Design of Tracing-logs for Application Performance Management System

YOU Yong[1], WANG Hao[2,3], REN Tian[1], GU Sheng-Hui[2,3], SUN Jia-Lin[1]

[1](Meituan-Dianping Group, Shanghai 200335, China)

[2](Software Institute, Nanjing University, Nanjing 210023, China)

[3](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: With the software system becoming more and more complex and distributed, it is more and more important to provide monitoring services with complete functions for the system. APM (application performance management) system analyzes the running state of software by collecting various indicator data of software system, such as CPU, memory utilization, the consuming time of garbage collection, QPS. In addition, the APM system can also generate various types of logs during the operation of the software. Generally speaking, it can provide three types of monitoring data: statistic metrics, tracing data, and discrete event records. The data can help the maintenance personnel of the system or service understand the running state, so as to ensure the stable operation of the system or service. Based on the open-source APM monitoring system (i.e., CAT system), this study proposes a storage design scheme for tracing data. It

* 基金项目: 国家自然科学基金(62072227, 61802173); 国家重点研发计划(2019YFE0105500); 江苏省政府间双边创新项目(BZ2020017); 计算机软件新技术国家重点实验室(南京大学)创新项目(ZZKT2019B01)

Foundation item: National Natural Science Foundation of China (62072227, 61802173); National Key Research and Development Program of China (2019YFE0105500); Intergovernmental Bilateral Innovation Project of Jiangsu Province (BZ2020017); Innovation Project of State Key Laboratory for Novel Software Technology (Nanjing University) (ZZKT2019B01)

本文由“面向持续软件工程的微服务架构技术”专题特约编辑张贺教授、王忠杰教授、陈连平研究员和彭鑫教授推荐.

收稿时间: 2020-09-15; 修改时间: 2020-10-26; 采用时间: 2020-12-15; jos 在线出版时间: 2021-02-07

improves the storage efficiency by memory block which is designed for batch writing logs, and query efficiency by the structure of the two-level index. Through analyzing the real on-line running data, the proposed scheme has sound performance in both write performance and query performance.

Key words: monitoring system; log storage; two-level indexing

当前,软件系统和服务所处理的业务越来越复杂;相应的,软件系统的规模和复杂度也在逐渐扩大.与此同时,为了支持高可用、高并发以及软件系统演进的同时持续提供系统服务,很多软件系统选择分布式部署的微服务架构,并引入 APM(application performance management)系统来对系统的各个服务进行监控.相关监控数据的采集和分析是服务监控的基础,开发人员通过对监控数据的分析,可以实时地了解到软件系统或者服务的运行状态,并且在软件系统或者服务发生异常后,通过诸如指标数据或者日志数据分析出导致异常的原因,进而及时处理故障,恢复服务.显然,分析并处理问题的实时性非常关键,这对监控数据的采集与存储提出了极高的要求.在监控数据中,有一类日志数据,用以记录一个软件系统或者服务运行时的动态行为.此前,大部分日志采集系统采用的都是集中的存储方式,但是随着业务扩大和用户数量的快速增长,日志数据量日渐庞大,再加上为了及时性,还需要支持较高的读写性能,这种集中的数据存储架构越来越难以满足目前很多软件服务监控对日志的存储和读写的要求.所以日志存储系统同样需要分布式架构才能满足整体的性能要求,通过存储集群的方式对外提供日志的存储与查询操作.如果使用现有的 HDFS 技术来实现日志文件的分布式存储,虽然让存储集群的整体性能提高很多,但是它需要对日志文件进行分块,并在集群的各个节点上存储副本[1],这样同时也增大了磁盘存储压力.所以,设计一个更轻量的分布式存储架构也同样很有必要.该轻量的分布式架构能够满足日志在存储集群中较为均匀地分布,使得存储集群整体的性能很高.

CAT[2]是美团开源的用于系统监控的项目,目前被广泛应用于国内多个行业的多个在线系统/服务中.截至 2020 年 6 月 30 日,CAT 在著名的开源软件平台 GitHub 上,同类型项目中的 Watch 数量(1.2K)、Fork 数量(4.4K)、Star 数量(13.6K)等均居首位,这充分说明了该监控框架受欢迎的程度.CAT 支持多语言客户端,内部定义了多种监控数据类型,Transaction 用于记录耗时较长的调用过程、Event 用于记录离散简单的事件、Problem 用于记录异常、Heartbeat 用于记录 CPU/JVM 等心跳数据.Transaction 数据通过嵌套的形式表示调用关系.此外,logs 数据同样会在项目的运行中生成.针对被监控服务采用 CAT 所产生的 Transaction 类型日志,本文对该类型数据设计了一种存储方案,使用数据文件、索引文件分别存储对应数据,并在此基础上设计了两级索引结构,文中将对该方案的系统架构与索引设计等进行详细的介绍.需要说明的是:本文将要讨论的日志数据为 CAT 中 Transaction 类型的监控数据,它同样是一种链路跟踪型(tracing)日志数据,在第 1 节中会对此类型数据进行介绍.

本文第 1 节首先针对单条日志的数据格式以及监控数据类型进行说明,提出设计方案面临的难点与挑战.第 2 节针对日志数据的特点分析多个现有的存储方式,证明它们并不适合于存储日志;此外,分析两种索引结构的优缺点,并提出本文存储方案的设计要求.第 3 节对日志存储的设计方案进行详细分析,并指出不足之处.第 4 节介绍存储方案的性能效果.第 5 节讨论本文索引结构的优缺点以及对可优化的问题进行说明,并提出适合的解决方案.最后指出未来的工作,并对全文进行总结.

1 介 绍

1.1 链路跟踪型日志数据

通常情况下,监控系统会跟踪一次请求中所有的 RPC 调用和功能模块调用,并记录分布式中的调用节点关系、节点内容的调用路径、部分参数值的等信息,并形成完整的调用链数据,此类型数据即为链路跟踪型数据(tracing 数据).Tracing 数据是通常意义下的链路跟踪型日志数据,在微服务架构的广泛应用下,各个后端服务节点之间存在大量的相互调用,所以记录一次调用中经过的所有节点、对应的某些参数以及节点内部的功能模块调用,对于系统的运维有着至关重要的作用.在系统或者服务出现异常时,一条完整的 tracing 数据可以给开发人员提供很好的问题排查依据.相关人员能够根据全链路数据梳理在一次调用中所有节点的依赖关系,从而定位

出现问题的原因.比如,根据链路日志数据可以生成各个模块间的因果关系图(一般是有向无环图),再通过图的相关算法分析出现问题的节点[3−7].生成 Tracing 类型的监控数据有较多的开源技术框架都能实现,例如 X-Trace[8]和 Dapper[9].Tracing 数据可以标识一个全局唯一 id,此外,每个节点中的监控数据段同样可以标识一个 id 值,而这些技术框架则通过将完整请求中所有节点内部的监控数据串联起来形成完整的 Tracing 数据.

1.2 CAT的日志数据类型

1.2.1 Transaction 数据

被监控服务使用 CAT 所生成的监控数据为 Transaction 监控数据,它可以视作 Tracing 中一个单节点内的监控数据,也同样是链路跟踪型日志数据.一个 Transaction 内部能够嵌套多个其他类型的监控数据,并记录下函数调用中的参数、自定义数据以及耗时等.在图 1 中展示了一条完整的 Transaction 数据,从中可以看出整体的数据结构、嵌套关系、自定义数据等.

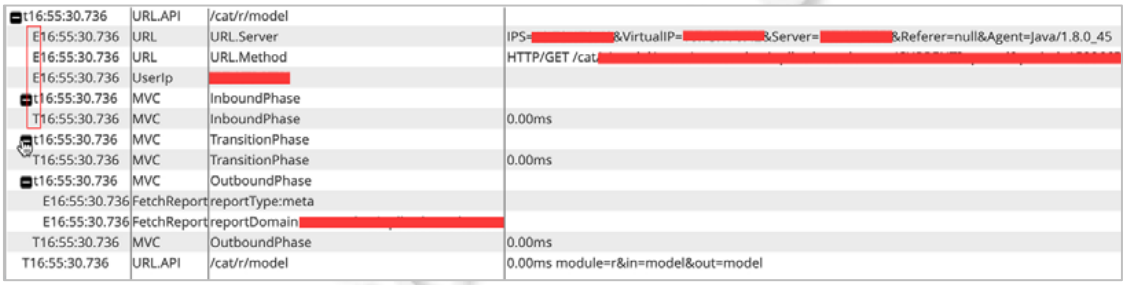


Fig.1 Visualization of “LogView”

图 1 LogView 的数据可视化

1.2.2 LogView

本文将一条完整的 Transaction 类型日志数据定义为 LogView,每条 LogView 中会记录一次请求调用中单台机器单个进程上的服务实例的函数调用过程以及各种自定义的数据.此外,每条 LogView 具有全局唯一的 id 标识值,该值由应用名、机器 ip 等信息组成.LogView 的 id 的统一格式为{domain}-{ip}-{hour}-{index},它由 4 部分组成.

- {domain}:指被监控的应用名.
- {ip}:ip 地址的 16 进制表示,指被监控应用所部署机器的 ip-v4 地址.
- {hour}:表示该条 logView 发生的小时,使用时间戳表示.
- {index}:递增值.index 需要以一定的周期将持久化到单独的文件中,当前采取了每 3s 刷新到磁盘一次的策略,这样可以避免机器宕机后,index 重置为 0 导致大量 id 重复.宕机重启后,首先从文件中加载之前的 index 值,并在此值基础上进行自增操作.此外,在多线程并发运行的情况下,可以通过对 index 值进行原子操作实现自增,不存在因同步引起的性能问题.每过 1 小时后,该值会重置为 0 重新开始计数.

图 1 展示了一条完整的 LogView 日志数据,从图中的长方框可以看出有 3 种类型的监控数据(即 E/t/T),t 是 Transaction 的开始标识,T 是结束标识,而 E 代表 Event 类型监控数据.整体来看,存在两层 Transaction 嵌套关系以及 3 个内部 Transaction 的并列关系,并且内部有多个 Event 类型.从该 LogView 中可以获取到多项监控指标,比如调用是否成功、开始与结束时间戳、耗时、自定义的分类名、自定义的附加数据.例如,图中的第 1 行与最后一行为一个 Transaction,第 1 列数据标识了数据类型、起始与结束的时间戳;而对应的第 2 列与第 3 列的信息是自定义的两级分类:一级分类为“URL.API”,二级分类为“/cat/r/model”;从第 4 列的信息可以看出,最外层 Transaction 的整体耗时小于 1ms,还附加了其他自定义的数据“module=r&in=model&out=model”.

Tracing 的完整数据可以视作将多个 LogView 数据按时间先后和调用关系关联起来的数据集合,数据集合中含有多个节点级的 Transaction 数据,且 Tracing 数据同样会具有唯一的 id 标识.

1.3 难点与挑战

CAT 的日志数据同样也是一种链路跟踪型日志数据.在美团的线上真实环境中,一条 LogView 日志数据单条字节量可能仅几十字节,也可能达到几兆字节;并且在美团线上服务中,CAT 所监控的所有软件系统每秒产生的 LogView 日志达到了百万条.所以,如何对海量日志数据进行分布式存储、如何提高存储效率、如何节省磁盘空间以及在查询时如何快速获取日志数据,都是存储设计方案所面临的难点与挑战.此类型链路跟踪型数据存在以下两点特征.

- 具有唯一的 id 标识;
- 单条日志数据的字节量相差很大,单条 LogView 字节量最小仅几 KB,最大可达几 MB.

本文存储方案考虑将 id 标识值利用起来创建索引结构,并通过数据文件存储日志数据、索引文件存储索引信息的方式来实现.此外,在真实的线上业务需求中,在LogView的写入与查询方面具有不同的性能要求,同样也是设计的挑战.

- LogView 写入.在美团的真实业务中,每秒产生的 LogView 量可达上万条,所以要求存储方案在存储方面具备较高的 TPS(transaction per second),例如:通过批量写入来减少磁盘 IO 以提高写入效率,使得存储方案的写入性能满实际的业务需求.
- LogView 查询.相对于 LogView 高频的写入操作,LogView 的查询操作是偶发的、离散的.偶发性:日志查询请求的频率是相对很低的,在实际业务中查询完整 LogView 的请求每秒仅上百次;离散性:在较小时间窗口内,对同一条 LogView 进行查询操作的情况很少发生.所以对于查询并不要求很高的性能,只要求单次查询能够在 100ms 内完整获取到原始的 LogView 数据即可.

2 相关技术方案分析

本节主要对现有的存储方案进行分析,以及分析可选的主流索引结构的优缺点,提出了本文存储方案的各项设计目标.

2.1 现有存储方案

LogView日志在生成时,会创建一个全局唯一的id值,用于区分每一条日志.同时也需要注意:线上实际业务中,每天 LogView 数据量会达到上百 TB;另一方面,为了方便业务进行故障定位和复盘,数据至少需要保存 2 周时间,这样就需要维护 PB 级数据量的存储和查询.下面介绍几种主流的数据库以及日志存储方案,并根据日志的数据特征解释它们是否适合 LogView 存储.

2.1.1 MySQL

MySQL[10]是目前主流的关系型数据库管理系统,默认使用 Innodb 存储引擎,通过建表的方式存储数据.如果使用MySQL存储LogView日志数据,建表时需要创建“id”字段与“日志数据”字段,存储日志数据时,以行的格式插入 LogView 的 id 值与对应的字节数即可,并使用 id 字段作为主键创建索引以提高查询的效率.如使用 VARCHAR 类型存放 LogView,在 MySQL 5.7 版本中,每行最大可存放 65 535 字节(即 64KB)数据.而 LogView 日志的数据最大字节量可达 MB 级别,64KB 大小的空间无法满足所有日志数据的大小,此类型数据不合适存储 LogView 数据.

MySQL 提供了 Blob 与 Text 类型两种类型的数据[11],存储大对象与字符串类型的数据.此类型的数据原则上可以用于存储 LogView 数据,但是前面提到:在实际业务中.LogView 每天写入会达到上百 TB 数据量,高峰期的 TPS 超过百万.为了应对这种大流量的数据写入,使用 MySQL 必然需要进行分库分表,并且搭建多个存储集群来支持,同时也需要维护一套分库分表的策略关系,在储存进行扩容或者缩容时,需要做好历史数据访问的兼容,增加整个系统搭建成本和复杂度;另一方面,MySQL 应对的储存场景比较丰富,支持数据的增删改查、事务等,所以 MySQL 整个架构的设计就比较复杂,并且数据存储时会自动额外增加很多隐形字段来支持实时的删除、修改.对于 LogView 的存储场景是不涉及数据的修改和删除,MySQL 提供的诸多能力对于此种场景是一种

浪费;另外,MySQL 为了支持这些能力,对整体性能做了很大折衷,所以使用 MySQL 存储从成本和技术运维角度并不适合.

此外,MySQL 基于 B+树建立索引,该方式由于需要建立索引树,对写入性能有一定影响.但是 B+树却提供了很高的查询性能,在第 1.3 节中提到了,LogView 是一种写多读少的场景,因此需要一种支持写入性能很高的存储技术.但是查询效率的要求并不高,从查询的角度而言,MySQL 应对的场景同样不适合 LogView.

2.1.2 MongoDB

MongoDB[12]是一个面向文档存储的数据库,可以将每条 LogView 日志数据以文档的形式存储在数据库中.而实际每日上百 TB 的 LogView 日志数据量并不合适存放在单个 MongoDB,需要采取 MongoDB 集群存放海量的数据.集群引入分片机制并存储数据的多份副本以保证数据的一致性[13],而海量日志数据的多副本存储对存储资源而言是一种浪费,所以该方式同样不适 LogView 的存储.

2.1.3 Key-Value

Key-Value 数据库是一种典型的 NoSQL 数据库,相关的数据库有 Memcached[14]、Redis[15]、TiKV[16].此类型数据库通过在内存中维护海量的键值对数据,在查询时直接使用 key 值即可,此类型的数据库将键值数据维护在内存中,从而使得系统整体的读写性能非常高[17].以 Redis 为例,它具有非常高的读写速度,此外,它还支持数据持久化到硬盘中、支持多种类型的数据、支持主从部署.如果选取 Redis 存储 LogView 日志数据,要求将日志数据都维护在内存中,但是每条日志数据字节量是相对较大的,所以能够在维护内存中的 LogView 的数量必然不会很大,从而无法将庞大数量的日志维护在内存中,所以同样也不适合于日志的存储.此外,分布式事务 Key-Value 数据库 TiKV 不仅能够支持分布式的数据存储,同样充分利用内存提供了很高的查询性能.但是 TiKV 是基于事务的数据库,强调数据的强一致性、数据的完整性等,在数据的写入时存在较多约束,写入性能受到影响.TiKV 的设计中,会对数据会创建多份副本存放在集群的不同节点中,对于 LogView 的海量数据而言,集群中存放多份数据副本会造成极大的存储开销.

综上,Key-Value 类型的存储方案同样不适合用于 LogView 日志的存储与查询.

2.1.4 ELK

ELK 是目前热门的日志管理与分析系统,ELK 是 Elasticsearch[18]、Logstash[19]、Kibana[20]的简称,Elasticsearch 是搜索与分析引擎,Logstash 负责日志数据的采集,Kibana 是基于 web 的图形界面.存储与查询主要与 es(elasticsearch)有关,它可以快速地储存、搜索和分析海量数据.原则上,es 可以用来存储日志数据,但是 ELK 这种高度集成化的日志管理框架在后期的定制化需求中存在局限性,其中,仅有 es 能够用于 LogView 日志的数据存储.但是由于 LogView 的数据特征,es 也仅能用于存储 LogView 的 id 与对应存储位置的索引信息.虽然 es 集群能够提供非常高的查询性能,支持高效丰富的查询筛选功能,而前面提到 LogView 搜索条件比较简单(根据 id 检索到具体的日志即可),并且日志数据量十分庞大,每天上百 TB 的存储量,es 不能十分高效存储这种海量数据.通常,es 单个索引存储上限仅几 TB,每日需要创建上百个索引去存储这些日志,极大地增加了运维成本,并且需要存储 2 周的 LogView 数据,则需要维持 10 余个 es 集群、上千个索引,运维开发难度极大.所以,单独设计完整的存储与查询方案会有更好的可用性.

2.2 索引结构

在实现 LogView 日志的存储时,同时也需要考虑合适的索引结构用以提高查询的效率.在日志数据写入磁盘后,需要充分利用日志的 id 标识、磁盘中的地址信息这两项数据来建立索引结构,这样在查询时即可直接根据 id 值获取到该条日志在磁盘中的位置信息.

下面分析两种在数据库广泛应用的索引结构,并指出它们的优缺点.

2.2.1 Hash 索引结构

Hash 索引[21]是基于哈希表建立的,哈希表是由数组与链表两部分组成,如图 2 所示.

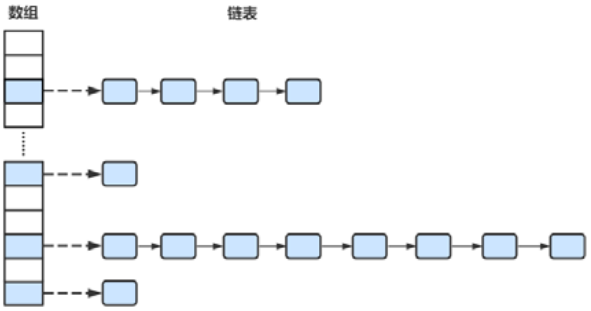


Fig.2 Structure of hash indexing

图 2 Hash 索引结构

在创建 hash 索引时,首先使用哈希函数对 LogView 的 id 值计算得到哈希值,再对哈希值与数组容量大小进行位运算,得到的值即为它在数组空间中的下标位置;之后新建一个节点,并将 id 值、地址信息记录到节点中;最后,将节点添加到链表尾部即可.

在使用 hash 索引查询 LogView 时,利用 id 值查询,起始同样是计算哈希值、位运算定位下标两个步骤,下标定位过程的时间复杂度为 O(1),这也是 hash 索引的性能突出优势.在获取到下标值之后,即可找到对应的链表,最后对链表进行遍历并一一比对.

hash 索引存在以下优缺点.

- 优点:索引的设计简捷、高效;使用 hash 函数定位数组空间的下标时,时间复杂度为 O(1).
- 缺点:(1) 查询效率不稳定,部分查询效率非常快,而部分则非常慢.这是因为 hash 索引结构中的链表部分的长度并不平均.如图 2 所示,图中的 4 条链表中的节点数目差距较大,所以在链表的遍历时所用的时间也不一致.所以,一个合适的哈希函数能够让 LogView 在哈希表中的分布更加均匀(即数组中不存在空洞、链表长度较平均),而这个函数的选择是比较困难的.(2) 在索引结构的数组部分存在较多的空洞,如图 2 所示,数组中较多的空间并没有用到,即较多的内存空间内浪费了.(3) 针对数组部分的 hash 函数难以找到合适的,此外,数组大小的初始化值同样不好定义,并且在数组空间满了之后,需要对数组空间进行扩容操作,实现上同样相对麻烦.最后,在链表的节点中,需要记录的信息较多,除了地址信息外,还需要额外将 id 值记录下来,这样在遍历链表时才能进行一一对比确定是否为指定的 LogView.

2.2.2 B+树索引

B+树[22]是当前主流数据库存储引擎(比如 Innodb)中常用的索引结构.在构建 B+树时,需要不断添加树节点以及调整树的结构;而在查询时,则直接根据树节点数据定位到根节点,获取到根节点中存储的地址信息即可.如图 3 所,B+树索引通常在 2~4 层,所以磁盘的 IO 次数也在 2~4 次,查询性能很高.

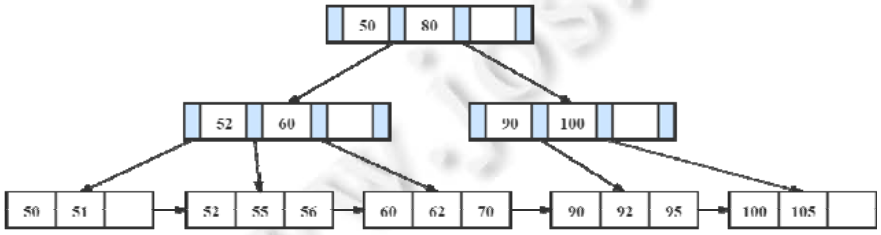


Fig.3 Indexing structure of B+ tree

图 3 B+树索引结构

在使用 B+树的构建索引结构时,同样需要针对 id 值建立索引树,B+树索引结构存在以下优缺点.

- 优点:B+树索引的查询性能稳定且高效;不同于 hash 索引需要预定义较大的内存空间并且可能会发生

扩容情况,而 B+树仅在需要时才开辟内存空间,相对而言,B+树索引可以节省更多的磁盘空间.

- 缺点:在构建索引的过程中,B+树索引结构需要频繁进行插入、删除节点以及调整树结构等操作,树的构建过程比较复杂.

本节对两种主流索引结构进行了分析,实际上,它们同样不合适于链路跟踪型日志的存储与查询,需要设计一个简单易理解、并且查询性能稳定的索引结构.

2.3 存储方案的设计目标

在第 1 节中提到了,利用 HDFS 技术实现存储系统的分布式架构对磁盘空间的压力较大,且从 HDFS 中读写文件的速度是要显著慢于单节点机器的,需要设计一个更轻量的分布式架构,在满足整体存储性能的同时提供较好的查询效率;此外,针对现有的多种数据库和 ELK 框架进行了分析.总的来看,上述方案并不合适于 LogView 数据的存储,需要设计一种新的存储方案,该方案需要满足以下 4 个目标.

(1) 轻量的分布式系统架构;

(2) 适合存储 LogView 这类链路跟踪型的具有较大字节量的日志;

(3) 设计简单易理解、查询性能稳定的索引结构;

(4) 符合第 1.3 节中提到的对于 LogView 的写入与查询性能的不同要求.

3 存储方案设计

本节主要对日志存储方案的系统整体架构设计、日志存储与写入的设计、索引结构的设计、内存空间使用情况进行详细的分析介绍,最后列举了存储方案中存储存在的几项特殊情况的处理方法.

3.1 系统架构设计

日志存储的整体架构设计如图 4 所示,被监控的应用服务集群直接将监控日志上报到存储集群,由存储集群对数据进行处理分析并存储成相应的数据文件与索引文件.

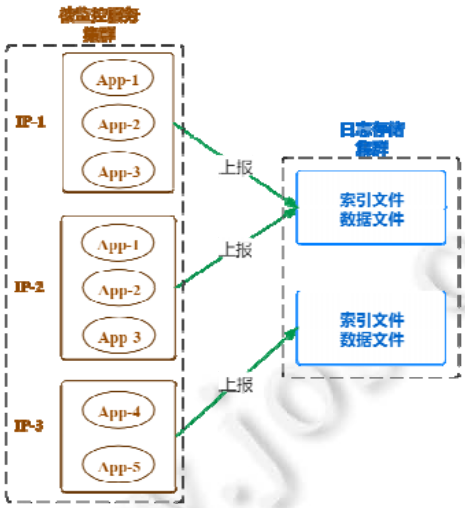


Fig.4 System architecture of logs storage

图 4 日志存储的系统架构

3.1.1 被监控应用集群

被监控集群中的每台机器上都存在多个被监控的应用服务,所以在上报日志信息时,每条日志信息都会记录下对应的应用名、机器的 ip 标识,可以区分该条日志数据来源于哪台服务器上的哪个服务.

被监控的机器上的链路日志上报至存储集群的哪台机器上,是通过配置的路由信息进行指定的,比如图 4

中被监控集群中的第 1 台机器的日志指定上报到存储集群中的第 1 台机器上.对于大量微服务运行的情况,由于扩缩容、资源利用等因素,导致某个微服务实例在不同时间点上分配了相同 ip,而实际运行的机器已经发生了改变,对于此类基础设施方面的变动,同样会被记录在路由信息中,即,同样能够使用 id 根据路由信息找到微服务实例变化之前该 id 所对应的文件存放机器.

3.1.2 存储集群

在日志存储集群的机器上,针对上报的日志信息生成对应的数据文件与索引文件,以小时粒度分别进行存储,即,把 1 个小时内所有来自某个指定应用服务的所有机器的日志数据存储在同一组数据文件、索引文件中.总的来说,则是将日志存储成文件后,通过多级文件夹路径、文件名来区分日志数据源.

- 一级路径:日期,标识不同年月日的日志数据.
- 二级路径:小时,标识在哪 1 个小时内的日志数据.
- 文件名,标识应用名.

反过来,在知道某个 LogView 的 id 时,根据第 1.2.2 节中介绍的 id 设计,可以从以下几个步骤定位到具体的数据文件与日志文件.

(1) 根据 id 值获取到被监控机器 ip.

(2) 根据 ip 与路由信息得知该条日志具体上报到了日志存储集群的哪一台机器上.

(3) 由 id 值解析出该条日志对应的文件的文件路径、文件名.

(4) 读取索引文件与数据文件解析真实的日志数据.

当前,采用配置路由信息的方式指定日志数据应该上报的对应机器,设计思路更易理解、实现也更便捷,只需要拉取一次路由信息即可.相对于直接使用 HDFS 技术搭建分布式文件系统,查询性能显著优于 HDFS 集群,但是固定的配置信息也存在局限性:存在某些机器的负载会很高、某些会很低的情形,会导致存储集群的整体性能并不稳定,可以通过周期性修改路由信息的方式(负载均衡)使得存储集群的性能更加稳定.

3.2 日志写入设计

日志在写入磁盘时,按照不同应用、不同小时存储成对应的一组日志文件,分成数据文件与索引文件:数据文件({domain}-{ip}.dat)用于存储日志数据,索引文件({domain}-{ip}.idx)用于记录每条 LogView 在数据文件中的位置信息(磁盘地址).

在往数据文件中写入数据时,针对海量的上报日志,采取批处理的方式能够大幅减少磁盘的 IO 操作,有效地提高存储效率.所以存储方案设计了“内存块”模式,以块为单位进行磁盘 IO 操作.此外,采用了当前主流的 Snappy 压缩算法,将原始日志数据压缩成字节数据再存储到数据文件中,能够节省较多的磁盘空间.

3.2.1 内存块

创建数据文件后,在往文件中写入 LogView 数据时,为了减少磁盘的 IO 操作,会在内存中维护一定大小的块,将每一条 LogView 的数据压缩后添加到内存块中.当内存块满后,会自动扩容内存块大小,然后将内存块持久化到文件中.在实际中,该内存块大小设置为 256KB,因为 32 位和 64 位的系统的页大小为 4K,8K,取两个数的倍数较好.一方面,为了尽可能减少磁盘写入次数,块的设置越大越好;另一方面,为了兼顾内存容量以及数据丢失的风险,内存块的设置不宜过大.之后,基于真实环境的多次实验与 LogView 的数据特征,确定了 256KB 更合适.

每条 LogView 的字节量大小不一致,所以在对一条 LogView 使用压缩算法压缩后,需要记录字节量的大小 logSize,并且在添加到内存块中时,需要首先添加 logSize 值到内存块中,设计使用 4 字节空间存储 logSize 值.在内存块将写入数据文件时,同样也需要添加内存块实际的字节量大小 blockSize 到数据文件中,并且需要记录下该条日志在块中的偏移量 offset.如果添加最新的 LogView 后数据块大小大于 256KB,内存块会自动扩充到容纳下当前 LogView 的大小,并在此时将内存块所有数据写入数据文件中.内存块中包含多条日志压缩后的数据以及对应的 logSize、内存块大小 blockSize 这 3 类数据,并且在内存块写入数据文件后,需要记录下内存块在数据文件中的首地址 headAddress.如图 5 所示,内存块字节量大小(*blockSize*)、日志字节量大小(*logSize*)与 LogView 日志压缩数据这 3 种数据会写入数据文件中,而内存块首地址(*headAddress*)、日志偏移量(*offset*)会与 id 值通过

建立索引的方式写入索引文件中.在第 3.3 节中会详细介绍索引结构的设计思路.

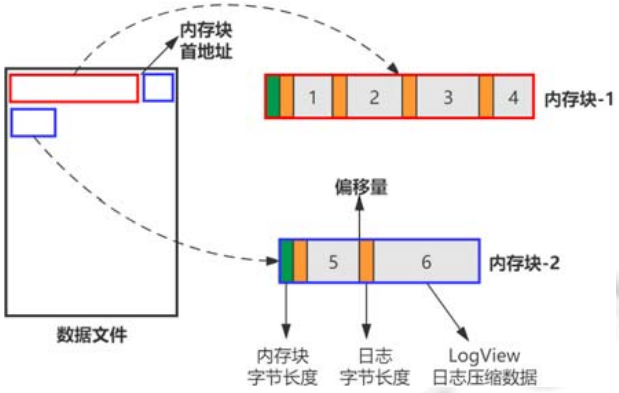


Fig.5 Storage format of data file

图 5 数据文件存储格式

一条完整的 LogView 持久化到磁盘文件中需要 4 个过程.

(1) 记录 LogView 压缩后的字节长度 *logSize*,使用 4 字节存储日志长度信息并添加到内存块中,添加日志数据到内存块中,并将长度信息的起始位置作为该条日志在块内偏移量 *offset* 记录下来.

(2) 内存块内容超过 256KB 后,内存块首 4 字节数据更新为实际内存块的字节长度 *blockSize*.

(3) 将内存块持久化到文件后,记录下在数据文件中字节块的首地址 *headAddress*.

(4) 将“LogView 在块内的偏移量”、“内存块的首地址”这两项信息存储起来用于创建索引文件.

整个过程对应的代码如算法 1 所示.

算法 1. 日志写入.

Input:

一条日志数据:*logView*;

内存块:*blockArray*.

Output:

写入成功.

```
function blockWrite(logView,blockArray)
  logData←compression(logView)          //压缩成字节数据
  logSize←len(logData)                  //单条 logView 的字节量大小,4 字节表示
  totalSize←len(blockArray)+logSize+4
  if totalSize>256KB then
    //自动扩容到适合大小
    blockArray.append(logSize)
    blockArray.append(logData)
    blockSize←len(blockArray)           //计算当前实际内存块大小
    blockArray[0:4]←blockSize           //更新块的头部 4 字节数据,为当前实际大小
    flush                               //写入文件,刷新到磁盘
    Call createIndex(·)                 //调用算法 2,创建索引结构
    return
  end if
  blockArray.append(logSize)
```

blockArray.append(*logData*)

end function

在反过来进行查询操作时,能够通过 *headAddress* 和 *blockSize* 读取指定大小的块到内存中,然后根据块内偏移量 *offset* 定位到内存块从哪里开始是表示指定 LogView 的数据.偏移量的后 4 个字节表示 LogView 占据的字节量大小 *logSize*,从而可以读取到对应 LogView 被压缩后的所有字节数据,再经过解压缩即可获得 LogView 的原始日志数据.

3.2.2 压缩算法性能比较

表 1 展示了多种主流的快速压缩算法的各项指标的数据,机器配置为 Core i9-9900K CPU@5.0GHz[23].从表格可以看出:lz4 的压缩与解压缩的速度都是最快的,而压缩率比较小.CAT 使用的压缩算法是 Snappy,虽然该算法的压缩与解压缩速度不如 lz4,但是 lz4 的压缩效率受数据块的大小影响较大,而 Snappy 受此影响较小.事实上,每条日志的数据量大小也是不定的,所以后者更加合适.Snappy 是目前很成熟的压缩与解压缩算法,它具有更加稳定的性能,所以同样选择性能稳定均衡的 Snappy 更合适.

Table 1 Performance comparison of compression algorithm

表 1 压缩算法的性能对比

Compressor name	Ratio	Compression (MB/s)	Decompress (MB/s)
zstd 1.4.5 -1	2.884	500	1 660
quicklz 1.5.0 -1	2.238	560	710
lzo1x 2.10 -1	2.106	690	820
lz4 1.9.2	2.101	740	4 530
snappy 1.1.8	2.073	560	1 790

3.3 两级索引的结构设计

在创建数据文件之后,需要创建对应的索引文件,且当内存块写入数据文件后,同样需要更新对应的索引文件.

存储方案设计了两级索引,每级索引中的索引条目(entry)空间大小为 8byte.如图 6 所示:正常情况下,存在一个一级索引和 4K−1(4×1024−1)个二级索引,且一级索引中存在 4K 个 Entry,每个二级索引中也同样存在 4K 个 Entry,所以所有一二级索引总计占用 4K×4K×8bytes(即 128MB)的空间.如果当前一级索引中所有 Entry 都被使用了,会在索引文件后继续创建一个新的一级索引和 4K−1 个二级索引;而索引文件中是否存在多个一级索引,可以通过文件中的大小进行判断:如果索引文件的大小超过了 128MB,则会存在多个一级索引.

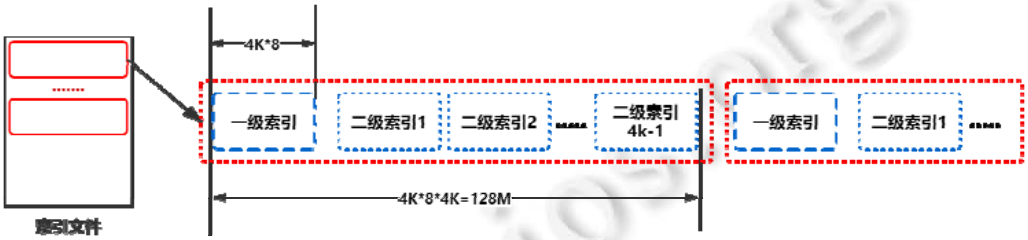


Fig.6 Structure of two-level indexing

图 6 两级索引结构

- 一级索引(header)

如图 7 所示,一级 Header 索引中存在 4K−1 个有效 Entry 与 4K−1 个二级 Segment 索引一一对应.每个 Entry 占据 8byte 的空间,且 8byte 共 64 位分别用于存储 ip 地址与{index}的哈希值,高 32 位存储 ip-v4 地址,低 32 位存储 index 模 4K 的哈希值.

- 二级索引(segment)

如图 7 所示,共计 4K−1 个 Segment 索引,且每个 Segment 下对应存在 4K 个 Entry,每个 Entry 同样占据 8byte

的内存空间.Entry 用于存储内存块在数据文件中的首地址以及 LogView 在块内的偏移量.图中 8byte 的前 40bit 用于存储内存块的首地址 headAddress,后 24bit 用于存储块内偏移量 *offset*.在计算机系统中,地址的编码是以字节为最小单位,故 40bit 可表示最大 1TB 的磁盘文件大小;而后 24bit 用于存储块内偏移量,可表示 2^{24} 字节量大小,而内存块的大小的初始值设置为 256KB(2^{18} byte),所以 24bit 完全满足要求.

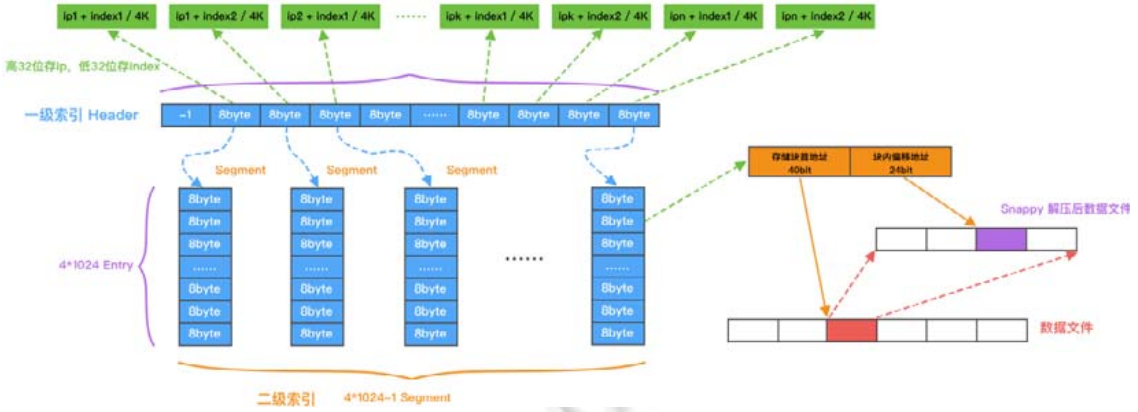


Fig.7 Creation and usage of indexing

图 7 索引的创建与使用

在理解一级索引与二级索引的结构之后,以一条 LogView 为例,说明索引的创建过程和索引的查询过程.

- 索引创建过程

在内存块写入数据文件中后,需要针对 LogView 的 id 创建索引并存储内存块在数据文件中的首地址和日志在块内的偏移量.图 7 中可以清晰地看出一级索引与二级索引在内存中的关系,它们都在内存中的同一个字节数组中,所以索引构建过程主要是对字节数组进行操作.创建索引的过程如下.

(1) 在根据 LogView 的 id 获取到 *ip* 与 *index* 值后,拼接"*ip*"与"*index*/4K"成为 8 字节的数据,表示为 *A*.

(2) 遍历所有一级索引 Entry 的值,如果存在与 *A* 值相等的数据,则直接返回对应 Entry 的位置 *L*;如果不存在,则顺序添加一个新的 Entry 值 *A*,对应 Entry 的位置为 *L*,并创建一个对应 Segment 索引.

(3) 访问对应的(即第 *L* 个)二级 Segment 索引,其中具有 4K 个 Entry,计算 *index*%4K 的值为 *T*,则将对应 Segment 索引中第 *T* 个 Entry 空间用于存储块的首地址以及块内偏移量.

算法 2 的代码完整展示了索引创建中,字节数组的操作过程.

算法 2. 索引创建.

Input:

索引文件读取为完整的字节数组,每个元素(entry)占 8byte:*array*;

id 的组成部分:*ip*,*index*;

内存块首地址:*headAddress*;

LogView 在内存块中的偏移量:*offset*.

Output:

创建完成.

function *createIndexing*(*array*,*ip*,*index*,*headAddress*,*offset*)

Header←*array*[0:4K] //0~4K−1 的 Entry 为一级索引段

A←*ip*+*index*/4K //拼接 *ip* 与 *index* 的哈希值

addressInfo←*headAddress*+*offset* //高 40 位存首地址与第 20 位存偏移量信息

i←1

```
Header[0]←−1                          //赋初始值
for each Entry∈Header do:             //顺序遍历一级索引中所有元素(entry)
  if Entry is NULL then:
    Header[L]←A                       //不存在则新建
  end if
  if Entry==A then:
    L←i
    break
  end if
  i←i+1
end for
Segment←array[L×4K:(L+1)×4K]          //一级索引 Entry 对应的二级索引段
T←index%4K                            //取余的值为下标位置
Segment[T]←addressInfo                //二级索引 Segment 段中第 T 个 Entry 存储 8byte 的位置信息
end function
```

- 索引查询过程

索引的查询与创建过程类似,即通过 id 信息定位到对应的二级索引段中对应的 Entry 空间,获取其中存储的 8 字节的位置信息,最后根据位置信息读取数据文件即可.算法 3 为对应过程的算法代码.

算法 3. 索引查询.

Input:

索引文件读取为完整的字节数组,每个元素(entry)占 8byte:*array*;

LogView 的 id 标识:*id*.

Output:

日志数据.

```
function queryLogView(array,id)
Header←array[0:4K]                    //0~4K−1 的 Entry 为一级索引段
ip, index←id                          //解析 id 值
A←ip+index/4K                         //拼接 ip 与 index 的哈希值
i←1
for each Entry∈Header do:             //顺序遍历
  if Entry==A then:
    L←i
    break
  end if
  i←i+1
end for
Segment←array[L×4K:(L+1)×4K]          //一级索引 Entry 对应的二级索引段
T←index%4K                            //取余的值为下标位置
addressInfo←Segment[T]                //获取位置信息
headAddress, offset←addressInfo       //解析 8byte 的地址信息
block←readFile(headAddress)           //读取特定大小的块
data←getData(block,offset)            //读取块中指定字节量
```

LogView←*decompress*(*data*) //解压缩

return *LogView*

end function

索引查询的过程如下.

(1) 首先,根据 LogView 的 id 值解析出 ip 地址,再根据日志上报的路由信息、{hour}数据和应用名信息定位到该日志存在哪台机器上哪个文件目录下,以及对应的索引文件和数据文件.此外,根据 id 获取到 *index* 值,并拼接 *ip* 与 *index*/4K 的值为 *A*.

(2) 遍历一级索引中所有的 Entry,找到与 *A* 值相等的一级 Entry 值,即可定位到对应的二级 Segment 索引.

(3) 根据 *index*%4K 的值定位到二级索引中的 Entry 位置,获取对应 Entry 中的数据,即可分析得到该 LogView 存放的内存块的首地址以及块内偏移量.

(4) 根据内存块首地址(*headAddress*)获取首部 4 个字节的数据,得到内存块具体的字节大小(*blockSize*),将指定字节量大小的块加载到内存中,再根据偏移量位置(*offset*)及后续 4 字节的长度信息(*logSize*),获取指定的长度字节数据后经过解压缩即可得到对应 LogView 的原日志数据.

3.4 磁盘空间管理

在索引文件存储的时候,一级索引的空间会全部创建,而二级索引只会在新建一级索引 Entry 值时才会创建.如图 8 所示,以 *ip*1,*index*(0~8K−1)为例,对应会创建两个一级索引 Entry 值,即“*ip*1,0”与“*ip*1,1”.一级索引中条目 Entry 值为“*ip*1,0”对应的 *index* 范围为(0~4K−1),根据 *index*%4K 的值,刚好与对应的二级索引 Entry 值一一对应,刚好占据 4K 个 Entry 值.“*ip*1,1”对应所有 LogView 的 *index* 范围为(4K~8K−1),而每个 LogView 根据 *index*%4K 的值刚好与图中第 2 个 Segment 索引中 4K 个 Entry 一一对应.以 *ip*2,*index*(0~3K−1)为例,占据了一级索引的第 4 个 Entry,且对应的二级索引中实际只占据了 3K 个 Entry 的空间.

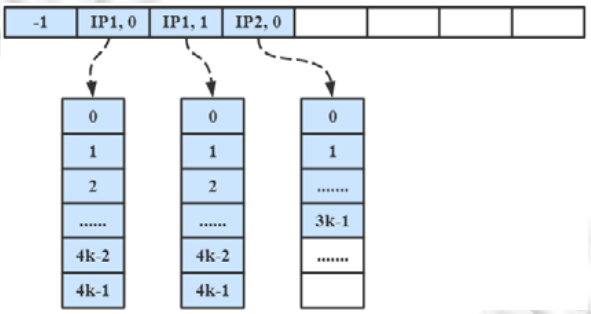


Fig.8 Memory usage of two-level indexing

图 8 两级索引的内存占用

总的来说,上述例子中来自 *ip*1 与 *ip*2 所有的日志数据在索引文件中仅占据了 128KB(4K×8+4K×8×3)的大小.二级索引的 Entry 值是按顺序存放的,顺序写数据的性能很高;此外,二级索引中不存在空洞空间,即不存在内存碎片,并且二级索引空间是在需要的时候才开辟,这些都可以有效的节省磁盘空间.

3.5 特殊情况处理

前面说明了正常情况下的日志数据存储与查询流程,下面对几种特殊情况发生后的处理方式作出说明.

(1) 读写冲突

在内存块写入数据文件时,如果此时有日志查询的请求到来,并且该条请求日志也正好在这部分内存块中,此时同时写和读数据文件时,理论上存储冲突.所以日志存储与查询的流程设计如下.

- 首先查询内存空间,如果存在,则直接从内存中返回日志数据;
- 如果不存在,则到磁盘文件中查找.

这样可以解决上面提到的读写冲突问题,并且在内存块完整写入磁盘以及索引数据更新到文件中后,才能将对应的内存块数据清空.

(2) 集群机器宕机

当存储集群中的某台机器宕机后,内存中的数据是会丢失的;此外,如果在内存块写入文件时出现宕机,数据块只写入了一部分,该部分数据就成为了无效的脏数据块.

理论上,监控日志的存储并不要求高可用、不可丢失性,日志存储系统是可以容忍少量的数据丢失的;在牺牲了一定的数据完整性的前提下,带来更便捷的设计与更高的性能.以 MySQL 为例,它引入了事务、bin log 与 redo log 这 3 种机制,确保在宕机时内存中的页数据也不会出现丢失,但是这种机制在实现上是非常复杂的,对于日志存储系统并不合适.

(3) LogView 的 id 重复

在一台被监控的服务器机器上,某一个应用在某小时内产生的 LogView 需要赋值 id 标识,此时需要针对 id 标识中的{index}段进行自增.如果此时发生进程中断或者机器宕机,内存中自增的 *index* 值会丢失,从而该小时内的 *index* 值重置为 0,导致大量的 id 重复出现,所以需要周期性地将 *index* 值持久化到磁盘文件中,存在以下两个问题.

- 时间周期太长会导致服务重启时重复 id 较多;
- 周期太短会因为存储 *index* 值的文件频繁 IO 操作影响机器性能.

当前采取每 3s 持久化 *index* 值到磁盘一次的策略,是多次真实实验后选出的较为合适的时间周期.

当出现 id 重复时,会直接更新之前 id 对应索引块中的地址信息为最新的地址.例如:当前小时内(3 600s)在第 3 002 秒发生宕机,再次启动时,会读取之前最新的 *index* 值(第 3 000 秒时的 *index* 值),重启后生成的 *index* 值会与第 3 000 秒~第 3 002 秒之间产生的 id 发生重复.在创建对应索引时,对应的一级索引 Entry 和二级索引 Entry 的空间中都已经存在相应数据了,此时采取直接覆盖的方式,更新为最新的地址,确保存储的信息是最新的.

4 效果评价

本节采集了部分真实环境的运行指标数据来体现本文存储方案的读写性能指标;此外,还随机选取了多组数据文件与索引文件来说明日志的数据特征与磁盘空间的占用情况.

4.1 性 能

本文存储方案的性能通过集群 TPS、机器的硬件指标等来体现写入性能、通过耗时来说明查询性能.在第 2 节中提到了多种相关的用于日志存储的数据库、存储方案,而它们并不适合于日志的存储,所以此处并没有测试对比它们的性能.

4.1.1 写入性能

在 LogView 日志写入数据文件时,需要等待内存块满后才能刷新到文件中,所以单条 LogView 写入的耗时与内存块合适能满有关,即与日志产生的频率有关;此外,内存块写入数据文件实际上仅是将内存块刷新到磁盘的过程,这主要与机器的配置有关,与存储方案设计的关系很小,所以本节不通过耗时来衡量写入性能.而通过真实运行环境中的各项机器硬件指标以及 TPS 指标来体现.

此处采集了线上服务与线下测试两个集群中的真实运行数据,线上的后端集群中共存在上百台服务器,各集群中硬件配置相同,随机选取了两个集群中的单台机器,它们的硬件配置(CPU 核数/内存/磁盘)如下.

- 线上环境机器 *A*:32 核/128G/5.44T.
- 线下测试环境机器 *B*:16 核/64G/3.76T.

上述两台机器(*A* 和 *B*)于 2020 年 5 月 1 日的 11:00~12:00 的 1 个小时内处理并存储了大量来被监控集群上报的日志数据,此处同样使用 CAT 系统采集了该小时内两台机器运行时的硬件指标、所处理的日志的各项特征指标,见表 2~表 4.

Table 2 Write performance

表 2 写入性能

	LogView 总量	TPS	CPU 平均使用率(%)	LogView 平均字节量(byte)	内存块数量
线上 *A*	173 060 000	47K	34.31	1 830	1 209 022
线下 *B*	59 320 000	16K	43.18	1 043	240 495

Table 3 Byte distribution of LogViews (%)

表 3 LogView 的字节分布情况(%)

	0~0.5KB	0.5KB~1KB	1KB~2KB	2KB~5KB	5KB~1MB	1MB~
线上 *A*	15.12	30.49	40.38	10.5	3.47	0.04
线下 *B*	27.68	44	21.61	5.1	1.6	0.01

Table 4 Hardware load

表 4 硬件负载

	CPU 使用率(%)			内存使用率(%)			硬盘使用率(%)		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
线上 *A*	29.56	34.31	56.84	59.10	59.12	59.28	0.1	5.83	83.7
线下 *B*	26.68	43.18	78.67	81.59	81.62	81.64	0.5	24.71	100

表 2 中的第 1 行数据指出线上机器 *A* 在该小时内共处理完成了 173 060 000 条 LogView 日志,单台机器的 TPS 达到了 47K,而在这段时间内的 CPU 平均使用率仅为 34.31%,说明 TPS 还有很大的提升空间,即该机器还能处理更多的日志数据,而整个线上存储集群存在 150 台机器,存储集群整体的 TPS 大概为 6.88M(150×47K)的水平,完全满足了日志存储的性能要求;此外,每条 LogView 的平均字节量为 1 830byte(压缩后),并且在存储数据文件时在内存中共计创建了 1 209 022 个内存块,用于 LogView 的批量存储.表 2 中的第 2 行数据则是线下机器 *B* 的对应数据.

表 3 给出了线上线下两台机器在该时间段内所有 LogView 的字节量(压缩后)分布情况.以线上机器 *A* 为例,数据说明大部分 LogView 的数据量都在 KB 级别.虽然 MB 级别的 LogView 占比较小,但由于 LogView 的总数十分庞大,MB 级别的 LogView 仍然高达 6 万多条.这进一步证实了第 1.3 节中提到的 LogView 的数据特征.

表 4 对应的是两台机器在该小时内 3 项重要的硬件指标数据.在指标数据采集时,采集了秒级的 CPU 使用率、内存平均使用率、磁盘平均使用率,并计算了小时内的最小、平均、最大的指标数据.以线下机器 *B* 为例,在此小时内,平均的 CPU 使用率、内存使用率、磁盘使用率的平均值分别为 43.18%,81.62%,24.71%.此时,机器 *B* 的 TPS 达到了 16K 的水平.从表 4 和图 9 的硬件性能数据可见:虽然硬盘利用率在某些时间内达到了 100%,但仅仅是短暂出现,大部分情况下都在 80%以下.说明这段时间内的 TPS 同样还有提升的空间,这也体现了单台机器很高的并发性能.

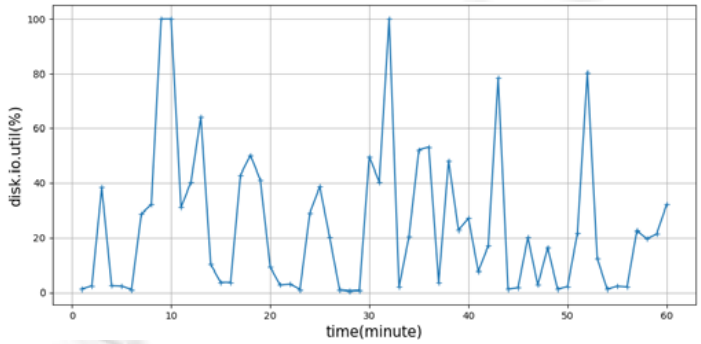


Fig.9 Disk util-rate of the offline *B*

图 9 线下机器 *B* 的硬盘使用率

4.1.2 查询性能

根据 LogView 的 id 查询对应日志数据时,首先读取索引文件,查询对应内存块的首地址与偏移量;其次读取数据文件中指定大小的内存块,并根据偏移量读取指定大小的字节;最后进行解压缩,将整个过程称为一次日志查询,该过程花费的时间定义为 Query Time.

日志的查询操作具有以下特点.

- 偶发性、离散性;
- 过程较长,涉及索引检索、数据块读取、解压缩.

日志查询不同于传统的应用于 OLTP(on-line transaction processing)服务的数据库(比如 MySQL),OLTP 服务需要进行大量的数据增删改查操作;而日志查询操作则完全相反,仅在需要排查异常时才会需要拉取到完整的日志数据,而这种操作发生的频率远小于日志的存储,所以日志的查询对并发性能的要求不高,并且相应的 QPS(query per second)要求同样较低.

此处同样采集了线上真实数据,统计了在 2020 年 5 月 1 日 11:00~12:00 期间的日志查询的请求,记录了每分钟内所有的日志查询请求以及对应的平均查询时间,图 10 展示了每分钟内所有查询的平均 Query Time.

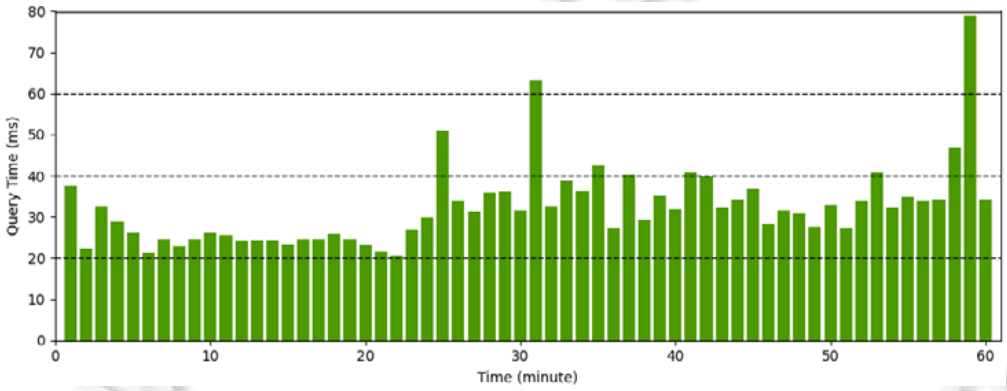


Fig.10 Consuming time of querying logs

图 10 日志查询的耗时

在此小时内,后端存储集群总计接收了 606 238 次日志的查询请求,系统整体在该小时内的 QPS 仅为 168,充分说明了日志查询操作的偶发性、离散性特征.

从图 10 中可以看出:大部分的日志查询耗时(query time)在 20ms~80ms 范围内;此外,最小、平均、最大的耗时分别为 20.6ms,32.29ms,78.9ms.整体而言,日志查询的耗时能够满足在第 2.3 节中所提出的 100ms 内的性能要求.

4.2 磁盘空间的占用

本节从线上真实运行环境中随机选取了存储集群中一台机器上的多组数据文件与索引文件,通过对实际生成的多组文件的分析,能够证实第 3.4 节磁盘空间管理的真实性.时间段同样为 2020 年 5 月 1 日 11:00~12:00,随机选取了 5 组索引文件与数据文件.表 5 的数据展示了对应的几组文件大小,从表格数据得到每组数据文件与索引文件在磁盘中的空间占用大小.

Table 5 File size

表 5 文件大小

	Application-1	Application-2	Application-3	Application-4	Application-5
数据文件(byte)	1.4G	190M	2.8M	1.1M	143K
索引文件(byte)	5.8M	3.9M	224K	64K	96K

对表格数据的分析能够证实两点结论.

(1) 索引结构设计简捷易于理解、磁盘空间管理上的真实性.基于对两级索引结构的理解,能够分析出索引文件中各级索引的详细组成.以 Application-3 为例,索引文件为 224KB 大小,即文件中存在 1 个一级 Header 索引以及 6 个 Segment 二级索引(8byte×4K+8byte×4K×6=224KB).进一步分析,一级索引中仅有 6 个 Entry 空间被占据使用了,二级索引中最多有 24K 个 Entry 被占据,即最多存了 24K 个 LogView.上述的推论过程能够体现出两级索引结构设计简捷易理解的特性,反推过程也能证实设计方案在磁盘空间管理上的有效性,即,在需要的时候才开辟新的二级索引空间.

(2) LogView 特征.单条字节量差距很大,Application-4 的索引文件、数据文件大小分别为 64KB,1.1MB,而 Application-5 的文件大小分别为 96KB,143KB.虽然 Application-4 的索引文件比 Application-5 的更小,但是数据文件却更大.这能够说明 Application-4 中的 LogView 数量较少,但是它里面的 LogView 的平均字节量却相对很大.这也证实了在第 1.3 节中指出的关于 LogView 特征的真实性.

5 讨 论

本节主要针对当前日志存储方案中存在的限制与不足进行说明,并对后 3 点不足提出了可行的解决方案.

(1) 关于性能测试

在第 4 节中,通过线上的真实运行数据,从读写性能、内存使用情况两方面来说明了本文存储方案的效果.由于 LogView 日志的数据特点,文中没有对相关存储方案进行测试实验并对比读写性能,仅说明了当前方案的各项性能,但该部分中的真实数据已经能够体现整体的读写性能也满足在第 2 节所提出的几点要求.

衡量写入性能、查询性能的线上数据实际上是同时发生的,理论上,查询的操作对写入性能存在一定的影响,即,在第 4.1.1 节中的运行数据存在一定的误差.但是在该时间段内,存储系统整体的写入 TPS 为 6.88M,而整体的查询 QPS 仅为 168,所以实际上查询操作对存储集群的影响微乎其微,可以忽略.

(2) 存在单点故障问题

如图 2 所示:每台机器上报日志的对象机器是通过路由信息指定的,每条 LogView 日志值仅存于日志存储集群的单个机器上.所以当某台机器出现宕机后,所有存在该机器上的 LogView 日志都无法访问到也无法进行日志上传,即,当前存储系统的架构存在单点故障问题.在前面提到,可以使用 HDFS 技术搭建分布式文件系统,但是会增加系统整体的磁盘存储压力,并且查询效率会相对降低.而实际上,少量的日志数据丢失是系统可以容忍的,所以考虑引入心跳检测的方式,在机器宕机后及时重启服务或者临时更改上报的路由信息.

(3) 地址空间可能会不够

在二级索引中,每个 Entry 具有 8byte 的空间,用于存储内存块的首地址和块内偏移量.而前 40bit 存储首地址时,能够表示的最大地址空间为 1TB.如果在当前小时内的日志数据量很大,对应数据文件的大小超过了 1TB 的大小,那么 40bit 的空间就不够用了.

目前考虑两种解决方案.

① 划分小文件,超过 1TB 的文件划分成多个小文件,并额外存储小文件的隶属关于用于日志的查询;

② 扩大 40bit 的空间.

第 2 种方式在实现上会比第 1 种更便捷也更易理解,但是会额外造成内存的开销,且文件大于 1TB 的情况非常少见.

(4) LogView 出现 id 重复

在第 3.5 节中提到,id 值的重复是由进程中断、机器宕机等导致的,而出现重复后则是直接更新为最新的值,这种方法会导致部分原日志数据的丢失.理论上可以缩小 3s 的持久化周期,但是文件的 IO 操作会对系统整体性能有影响.此处提出了新的解决方法,能够进一步降低 id 重复的频率.

考虑将 *index* 值的更新操作放到索引文件中,如图 11 所示,在二级 Segment 索引中设置 4K+1 个 Entry 空间,最后一个用于存储 *index* 值.在每次内存块更新到数据文件之后,需要更新索引文件时,同时更新最后一个

Segment 中最后一个 Entry 的值为当前最新的 *index* 值,即,更新索引文件的最后 8 个字节即可.此方案不需要在单独的文件中存储 *index* 值,并且在每次更新索引文件时就可更新 *index* 值,不仅减少了文件 IO 操作,且本方案可以有效降低 id 值重复的频率.之前通过单独文件存储 *index* 的方式在宕机后会造成上当前 3s 周期内所有 *index* 都会重复,而当前方案只会导致在内存块中的 LogView 对应 *index* 出现重复.

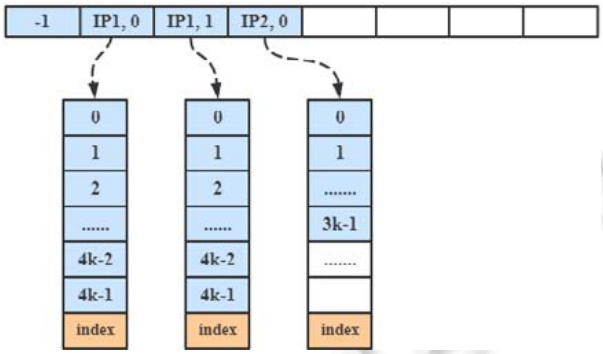


Fig.11 Design scheme of Index value

图 11 Index 值的设计方案

(5) 是否引入对 LogView 的缓存

理论上可以引入缓存机制,在内存中维护固定量的 LogView,并使用 LRU 策略进行更新.但是在第 1.3 节中提到了 LogView 的查询具有偶发性、离散性的特点,在较小时间段内,某一条 LogView 被多次请求的情况极少,即不存在热点数据,缓存机制无法有效的发挥作用,反而造成了额外的性能开销,所以此应用场景并不合适引入缓存机制.

6 总结与未来工作

本文不仅介绍了我们针对 CAT 这个 APM 框架而设计的日志存储系统的整体架构,而且详细讲解了存储的设计方案:使用内存块批量写入文件的方式提高写入效率、存储时采用了 Snappy 压缩算法节省磁盘空间,以及设计了两层索引来提供较高的日志查询效率.从最后的性能测试结果来看,本文的存储方案基本满足了前文提到的多项目标,且具有以下几个特点.

(1) 设计简单.在 LogView 的存储上,本文的两级索引结构与 hash 索引同样简捷、高效、易理解,而 B+树索引则相对复杂.在索引结构中,它仅需记录地址信息,而另外两种需要额外记录 id 标识,所以在代码的实现上也更容易,并且在索引构建时耗时相对更少.

(2) 性能稳定.存储方案中设计了更轻量的分布式系统架构,能够满足系统的整体读写性能;此外,两级的索引结构也提供了稳定、高效的查询效率.在第 4.1 节的性能结果中展示了线上真实运行数据,一条 LogView 查询过程的平均耗时在 32.29ms,满足第 2.3 节中提出的业务需求.

(3) 通用性.本文的存储方案原则上是可以通用的,只要对应的日志数据具有全局唯一的 id 标识即可,该方案也同样适用于前面所提到的 Tracing 类型数据的存储.

相对于已有的存储方案而言,除了上述几个显著特点以外,还存在其他设计优势:

(1) 该轻量级的分布式存储设计在 LogView 生成的时候就做了对应的“分库”操作,而其他现有的成熟数据库则需要在后期数据量庞大的时候做“分库、分表、同步”等操作,造成性能上的问题.

(2) 存放在同一个数据文件中的 LogView 数据来自于同一个应用实例,而来自于同一个实例的日志存在更多的相似数据,能够使得压缩算法具有更高的压缩率,节省更多的存储空间.

本文设计方案的创新性与贡献也同样在于上述特点与优势.

当前方案基本满足了第 2.3 节中针对存储方案提出的多项设计目标,且具有稳定、高效的性能.未来的工作主要从以下两个方面展开.

- 首先,需要实现第 5.2 节中针对不足所提出的解决方案.
- 其次,对当前的索引结构进行进一步的优化,例如将{*index*}值的持久化操作并入索引构建的过程中,可以减少磁盘 IO 以提高写入性能.

References:

[1] Ban QC. Research and implementation of Web log storage and analysis system based on hadoop [MS. Thesis]. Beijing: Beijing University of Posts and Telecommunications, 2018 (in Chinese with English abstract).

[2] Dianping/Cat. Meituan, 2020. https://github.com/dianping/cat/

[3] Lou JG, *et al*. Mining dependency in distributed systems through unstructured logs analysis. ACM SIGOPS Operating Systems Review, 2010,44(1):91−96.

[4] Chow M, Meisner D, Flinn J, *et al*. The mystery machine: End-to-end performance analysis of large-scale internet services. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2014). 2014. 217−231.

[5] Fonseca R, Freedman MJ, Porter G. Experiences with tracing causality in networked services. In: Proc. of the INM/WREN. 2010.

[6] Kaldor J, *et al*. Canopy: An end-to-end performance tracing and analysis system. In: Proc. of the 26th Symp. on Operating Systems Principles. 2017.

[7] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: Proc. of the 25th Symp. on Operating Systems Principles. 2015.

[8] Fonseca R, *et al*. X-trace: A pervasive network tracing framework. In: Proc. of the 4th USENIX Symp. on Networked Systems Design & Implementation (NSDI 2007). 2007.

[9] Sigelman BH, *et al*. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report, Google, 2010.

[10] DuBois P. MySQL. Addison-Wesley Professional, 2013.

[11] Mysql 5.6 reference manual: 11.3.4 the BLOB and text types. 2020. https://dev.mysql.com/doc/refman/5.6/en/ blob.html

[12] Chodorow K. MongoDB—The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly Media, Inc., 2013.

[13] Song YH. Massive logs in the decision-making assistance system based on MongoDB storage and analysis. Science & Technology Innovation and Application, 2019(33):5−8 (in Chinese with English abstract).

[14] Jose J, *et al*. Memcached design on high performance rdma capable interconnects. In: Proc. of the 2011 Int'l Conf. on Parallel Processing. IEEE, 2011.

[15] Redis. 2020. https://redis.io/

[16] Tikv. 2020. https://tikv.org/

[17] He HG. Design of mass log storage system based on key-value [MS. Thesis]. Shanghai: Fudan University, 2013 (in Chinese).

[18] Gormley C, Tong Z. Elasticsearch: The Definitive Guide: A Distributed Real-time Search and Analytics Engine. O'Reilly Media, Inc., 2015.

[19] Turnbull J. The Logstash Book. James Turnbull, 2013.

[20] Gupta Y. Kibana Essentials. Packt Publishing Ltd, 2015.

[21] Mitchell GR, Houdek ME. Hash index table hash generator apparatus. United States Patent 4215402, 1980-7-29.

[22] Jensen CS, Lin D, Ooi BC. Query and update efficient B+-tree based indexing of moving objects. In: Proc. of the 30th Int'l Conf. on Very Large Data Bases, Vol.30. VLDB Endowment, 2004.

[23] Zstandard: Real-time data compression algorithm. Facebook, 2020. https://facebook.github.io/zstd/#GUI

附中文参考文献:

[1] 班秋成.基于 Hadoop 的 Web 日志存储和分析系统的研究与实现[硕士学位论文].北京:北京邮电大学,2018.

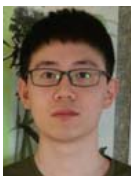
[13] 宋瑜辉.基于 MongoDB 存储和分析辅助决策系统中的海量日志.科技创新与应用,2019(33):5−8.

[17] 何海刚.基于 Key-Value 的海量日志存储系统设计[硕士学位论文].上海:复旦大学,2013.

尤勇(1987－),男,主要研究领域为分布式监控与告警.

汪浩(1998－),男,硕士生,CCF 学生会员,主要研究领域为智能运维.

任天(1997－),男,高级工程师,主要研究领域为分布式监控与告警.



顾胜晖(1995－),男,博士生,主要研究领域为实证软件工程.

孙佳林(1988－),男,主要研究领域为分布式监控与告警.