Apache IoTDB 的分布式框架设计

李天安^{1,2,3}, 黄向东^{1,2,3*}, 王建民^{1,2,3}, 毛东方^{1,2,3}, 徐毅^{1,2,3}, 袁骏^{1,2,3}

1. 清华大学软件学院, 北京 100084

2. 大数据系统软件国家工程实验室, 北京 100084

3. 北京信息科学与技术国家研究中心, 北京 100084

* 通信作者. E-mail: huangxdong@tsinghua.edu.cn

收稿日期: 2019-08-30; 修回日期: 2019-11-05; 接受日期: 2019-12-17; 网络出版日期: 2020-04-27

国家重点研发计划 (批准号: 2016YFB1000701) 和国家自然科学基金 (批准号: 61802224, 71690231) 资助项目

摘要 Apache IoTDB 是一个新型的开源时序数据库管理系统. 分布式数据管理系统不仅需要解决数据分区与多副本带来的节点间元数据同步问题, 还要支持高效查询请求处理. 本文针对节点间元数据同步问题提出了双层粒度元数据管理策略, 在此基础上基于一致性哈希分区方法和 Raft 协议设计了同时支持强一致性查询和最终一致性查询的分布式框架. 基于单机版 Apache IoTDB 进行了系统实现与实验测试, 测试结果表明: 双层粒度元数据管理策略与单层粒度管理策略相比, 其元数据内存资源占用更少且写入性能提升 5% ~ 10%, 并且分布式 Apache IoTDB 的读写性能随着集群规模的扩大而线性增长.

关键词 物联网数据库, 时间序列, 分布式框架, Raft 协议, 元数据, 强一致性, 最终一致性

1 引言

近年来, 随着信息技术的不断发展, 越来越多的机器设备产生的数据被收集并存储. 这些数据都包含一个时间戳来标识数据产生的时间, 因此被称为时间序列数据^[1] (简称时序数据). 时序数据具有以下特点^[2].

(1) 数据产生速度快. 例如国际风电标准 IEC 61400-25 规定, 每台风机每秒需采集 225 KB 的工况数据, 在极端工况下采集频率高达 8 KHz.

(2) 数据总量大. 机器设备长期处于 7×24 小时不间断采集数据的运行状态, 这对海量时序数据存储提出了要求.

(3) 数据种类多. 数据的来源和数据格式是多种多样的. 有的是单一的布尔值, 有的是浮点数, 有些特殊的传感器一个时间戳对应多个值 (例如 GPS 数据对应经度和纬度).

引用格式: 李天安, 黄向东, 王建民, 等. Apache IoTDB 的分布式框架设计. 中国科学: 信息科学, 2020, 50: 621–636, doi: 10.1360/SSI-2019-0189
Li T A, Huang X D, Wang J M, et al. The design of Apache IoTDB distributed framework (in Chinese). Sci Sin Inform, 2020, 50: 621–636, doi: 10.1360/SSI-2019-0189

分布式时序数据库系统是解决海量时序数据的存储和高效管理的途径之一, 实现分布式时序数据库系统的过程中会面临以下问题.

(1) 需要管理庞大的元数据. 机器设备管理涉及大量传感器, 每个传感器的元数据包括数据类型、压缩编码方式、权限等属性, 因此在时序数据的应用场景中通常需要管理庞大的元数据. 而在系统执行写入操作时, 需要通过元数据检查数据类型等的合法性; 系统执行查询操作时, 需要元数据进行查询路由转发等操作, 因此读写数据都需要获取相应时间序列的元数据. 在单机时序数据库系统中, 元数据保存在本地, 因此数据写入时可直接获取元数据. 但在分布式架构下, 元数据的管理策略将会影响数据写入的性能. 若在每个节点中存储全量的时间序列元数据, 由于元数据总量大, 过多冗余存储的元数据将会消耗大量的系统内存资源; 若将元数据划分为几部分由不同的节点进行管理, 则数据写入时需要节点间进行通信获取元数据, 从而降低了系统的写入性能. 因此实现分布式时序数据库系统时, 需要针对元数据总量大以及元数据对读写数据的影响来设计适当的元数据管理策略.

(2) 需要支持数据的多种副本一致性. 时序数据在一些应用场景下必须保证数据的强一致性, 否则系统将不可用. 例如在股票交易的场景中, 在任何时间点的查询结果要保证一致性. 而某些场景下, 只需保证数据读取的最终一致性即可.

(3) 节点间的冗余通信量会降低查询性能. 由于数据分区, 在分布式下查询通常需要从多个节点中获取数据, 节点间的通信量, 包括通信次数和通信的数据量, 将会影响查询性能. 因而设计查询引擎时要对查询进行适当的拆分来降低节点间的通信量.

本文的主要贡献有以下 3 方面.

(1) 提出了一种双层粒度的元数据管理方法 (dual-layer granularity metadata management, DMM). 该方法通过对时间序列进行分组将元数据分为两层, 一层为时间序列组元数据, 它在集群所有节点间同步; 另一层为时间序列元数据, 它仅与相应的数据一同在分区节点间同步. DMM 通过降低数据写入操作过程中元数据同步的代价来提高系统的写入性能, 同时占用更少的内存资源.

(2) 基于 Raft 协议^[3]在 Apache IoTDB 单机版系统上实现了按时间序列组进行分区且提供副本强一致的分布式框架. 该框架基于一致性哈希 (Hash)^[4]的思想, 将时序数据按所属时间序列组均匀地散列到分布式系统的各个节点上, 并通过加入虚拟节点的方法来避免出现数据倾斜^[5]问题. 通过将同一个数据分区的节点组成一个 Raft 组保证了分区内的副本一致性.

(3) 优化查询引擎. 在实现查询引擎时采用了基于时间序列组的过滤树剪枝方法, 有效地降低了节点间的数据传输量, 优化了查询速度. 同时, 查询引擎除了提供强一致性查询外还提供了最终一致性查询, 最终一致性级别的查询方式可以使查询性能提升一倍以上.

本文第 2 节介绍相关工作, 第 3 节介绍 IoTDB 系统中的基本概念, 第 4 节介绍双层粒度元数据管理方法, 第 5 节介绍 Apache IoTDB 分布式系统的实现, 第 6 节为实验部分, 第 7 节总结全文.

2 相关工作

2.1 分布式系统概述

目前分布式系统架构可以大致分为两大类, 一类是以 Hadoop^[6]为代表的主从架构^[7], 一类是以 Cassandra^[8]为代表的对等 (P2P) 架构^[9].

采用主从架构的 Hadoop 集群中有一个主节点, 一个备用主节点和若干个从节点. 在处理客户端请求时, 对于写入请求, 主节点决定哪些从节点来处理该请求; 对于查询请求, 需要主节点提供可以处

理该请求的从节点列表, 根据一定的策略选择一个从节点进行查询. 该架构的问题是容易出现单点故障和单点瓶颈, 因而主节点一般部署在支持 RAID^[10] 的机器上.

在 P2P 架构的 Cassandra 集群中, 将主节点的职责分散到集群中的各个节点. 当客户端发送一个写入请求时, 收到请求的节点会作为“协调者”, 通过一致性哈希计算该请求应该被集群中的哪些节点处理, 然后转发该请求. 根据用户设置一般当超过半数节点处理结束之后, “协调者”节点向客户端返回执行结果. 处理查询请求的方式类似. 该架构的问题是在查询时会触发数据一致性^[11] 的问题, 保证强一致性需要读取多个副本^[12] 的数据, 通过互相比较得出最新的数据.

2.2 时序数据库系统

Apache IoTDB¹⁾ 是一款高性能轻量级的时序数据管理系统, 具备高速数据写入 (百万数据点秒级写入)、快速查询 (TB 级数据毫秒级查询)、提供时序数据的增删改查和聚合功能.

目前已有的时序数据库系统有 3 个. InfluxDB^[13] 分布式版本收费且不公开源码; OpenTSDB^[14] 底层受限于 HBase^[15], 无法应用于高吞吐量的场景; KairosDB²⁾ 底层基于 Cassandra, 其分布式版的系统实现基于 Gossip^[16] 和 Quorum 协议.

2.3 数据分区与元数据管理

目前数据分区的方式主要有两种: 查找表分区和哈希分区. 一些分布式系统使用查找表记录数据和分区信息, 如 MongoDB³⁾, BigTable^[17] 和 HBase 等; 还有一些系统, 如 Dynamo^[18], Cassandra 等, 选择了一致性哈希的方法作为数据分区的策略. 此外, 还有一些结合了查找表分区和一致性哈希的混合方法^[19] 被提出. 由于使用查找表记录分区信息的方式会随着时序数据的不断写入, 分区信息的内存占用量不断增加, 从而耗费大量的系统资源, 因此对于可能拥有数千万条元数据记录的 Apache IoTDB 来说, 实现其分布式系统时不适合采用查找表进行分区, 故采用一致性哈希算法和虚拟节点机制进行数据分区.

目前元数据管理方法主要有两个: 一是有一些元数据节点来专门管理元数据, 如 OpenTSDB 中使用 Zookeeper^[20] 来管理元数据, 这种方法的元数据仅全量保存在部分节点中, 从而降低了内存资源消耗, 但因为读写操作需要和元数据节点进行交互, 会造成性能下降. 二是所有节点存储全量的元数据, 如 KairosDB 中使用 Gossip 协议将元数据在所有节点间进行同步, 这种方法下由于读写操作直接本地读取元数据因此保证了读写性能, 但元数据占用了极大的内存资源. 由于现有的方法都有相应的问题, 于是本文提出了一种双层粒度的元数据管理策略, 既能降低元数据的内存资源占用又能保证读写性能.

2.4 共识协议

分布式系统中通常以多副本的方式实现数据的可靠存储. 同一份数据的多个副本存储在不同的节点中且必须保证一致, 因此需要使用共识协议来保证数据的一致性. 比较著名的有 Paxos^[21] 和 Raft.

Raft 协议是由 Ongaro 等^[3] 于 2013 年提出的一致性协议, 它将一致性的问题拆分成了几个相对独立的子问题, 即领导人选举、日志复制、一致性保证等. Raft 协议中有以下重要的概念.

(1) Raft 组. Raft 组中有一个通过选举产生的领导人节点, 其他节点是跟随者. 当一个写入请求到来时, 首先要提交给领导人处理, 领导人先在自己的日志里面记录下这个写入请求, 然后将这条日

1) Apache IoTDB: <https://iotdb.apache.org/>.

2) KairosDB: <http://kairosdb.github.io/>.

3) MongoDB: <https://www.mongodb.com/>.

图 1 时间序列组织结构

Figure 1 Organization structure of time series

志同步给跟随者。当超过集群半数的跟随者收到日志后, 领导人执行这条日志并将结果返回给客户端。

(2) Raft 日志. Raft 通过日志的方式保证操作不会丢失, 日志中维护了一个 Commit 编号和 Apply 编号. 如果一条日志被 Commit, 就代表目前集群中超过半数的节点都收到并持久化了这条日志. 如果一条日志被 Apply, 就表示当前节点执行了这条日志. 当某些节点出现故障并重新恢复时, 该节点的日志就会落后于领导人的日志. 则在这个节点追上领导人的日志之前, 它不能向外界正常提供服务。

3 Apache IoTDB 系统基本概念

本节将介绍 Apache IoTDB 系统中的两部分元数据: 时间序列和存储组。

3.1 时间序列

在 Apache IoTDB 中, 一条时间序列的数据被定义为一个设备上的一个传感器采集到的数据, 这些数据是由若干 〈时间戳, 数值〉组成的集合. 传感器是最小的数据采集单元, 设备是一系列传感器构成的集合. 例如, 一台风力发电机在 Apache IoTDB 中认为是一台设备, 发电机上安装的测量叶片转速、风速、温度的各种采集单元在 Apache IoTDB 中认为是传感器。

3.2 存储组

Apache IoTDB 中时间序列名称的格式为 “root.sg1.device1.sensor1”, 其中 “root” 是保留字. Apache IoTDB 在存储时序数据时, 逻辑上按照树形结构存储, 如图 1 所示. 在这棵树上, 传感器代表叶子节点. 从根节点到某个传感器所在的叶子节点的一条路径代表一条时间序列。

在图 1 所示的结构层次中有一类特殊的节点, 叫存储组节点, 它包含多个时间序列. 在 IoTDB 中, 磁盘组织数据以存储组为最小单元进行划分, 同一个存储组的数据通常会被同时写入和查询, 将其存放在一起有助于提升写入和查询速度。

4 双层粒度元数据管理方法

元数据管理策略是 Apache IoTDB 的分布式设计中的要点. 在进行元数据管理策略设计时首先要考虑元数据在读写流程中的用途: 写入数据时需要元数据进行数据类型、权限等合法性检查, 查询数据时需要元数据进行查询路由. 同时, 由于时序数据场景中元数据庞大, 还需要考虑元数据对内存资

图2 集群分组 (5 节点 3 副本)

Figure 2 Cluster grouping (5 nodes 3 replicas). (a) Metadata group; (b) data group 1; (c) data group 2; (d) data group 4; (e) data group 5; (f) data group 6

源的消耗. 现有的元数据管理策略要么采用将元数据交由元数据节点专门管理的方式, 这种方法会降低读写性能; 要么采用在集群所有节点全量保存元数据的方式, 这种方式会消耗大量的内存资源.

为了解决上述问题, 本文设计了双层粒度元数据管理方法, 其核心思想是通过将元数据拆分为存储组和时间序列两层分别管理, 其中时间序列元数据中包含了数据写入时需要的数据类型、权限等信息, 存储组元数据包含了查询数据时的路由信息, 并通过存储组元数据在集群所有节点全量保存以及时间序列元数据在部分节点中保存来降低内存资源的消耗.

下面以 5 节点 3 副本为例进行详细描述. 将 5 个集群节点按哈希值放在哈希环中, 所有节点组成一个元数据组; 以每个节点为起始点, 在哈希环上顺时针寻找 3 个节点组成一个数据组. 如图 2, 该集群中包含一个元数据组和 5 个数据组, 图中虚线框中代表一个分组.

DMM 方法中包含元数据持有者和数据分区持有者, 元数据组中的每个节点称为元数据持有者, 数据组中的每个节点称为数据分区持有者, 采用 Raft 协议来保证每个持有者与同组的其他持有者的数据一致性. 该方法将元数据按存储组和时间序列两层粒度分别在元数据持有者和数据分区持有者中管理, 由于将时间序列元数据和数据一同在数据组内同步, 因此每次数据写入不需要进行元数据的检查与同步操作, 仅需要在修改时间序列元数据时进行存储组元数据的检查与同步操作, 从而提高系统性能. 例如创建一个时间序列并进行 50 万次数据写入的操作中, 元数据检查与同步操作从 50 万次降至 1 次.

4.1 数据分区

分布式系统实现中基于哈希环和环上查找算法将时序数据按照存储组进行分区. 首先将集群各个节点按哈希值放到哈希环上, 对于到来的一个时间序列数据点, 计算这个时间序列名称所对应的存储组的哈希值并放置到哈希环上, 在环上按顺时针方向进行搜索, 找到的第 1 个节点就是要插入的节点 (如图 3, 插入的是节点 1).

图 3 一致性哈希查找
Figure 3 Consistency hash lookup

图 4 虚拟节点
Figure 4 Virtual node

图 5 元数据持有者
Figure 5 Metadata holder

使用哈希环进行数据分区时, 容易出现两个节点的哈希值的差较小的情况, 如图 4 中左图所示, 此时大部分的数据全部被节点 1 接管保存, 会导致数据严重倾斜并影响系统性能. 因此在使用一致性哈希环的基础上引入虚拟节点, 具体做法是将每个物理节点虚拟成若干个, 并将这些虚拟节点按照哈希值放置到哈希环上, 此时图 4 中数据点本来要存储在物理节点 1 上, 引入虚拟节点后就被节点 2 的虚拟节点接管, 即被物理节点 2 接管, 在很大程度上避免了数据倾斜的情况, 使数据分布得更加均匀.

4.2 元数据持有者

元数据持有者负责管理和同步当前已创建的存储组信息, 分布式集群中的每个节点都是元数据持有者, 且元数据持有者所在组中的领导人称为 Manager. 元数据持有者的作用是使集群中的每个节点都保存了必要的“索引”信息. 对于查询“select * from root”, 和客户端连接的节点 P 应能够在本地将这种查询拆分为若干个针对不同存储组的查询, 然后根据一致性哈希计算出每个存储组的数据节点, 最后向这些节点发起查询.

采用 Raft 协议将所有元数据组持有者组成一个 Raft 组, 实现了元数据组内的数据同步. 元数据组可以处理设置存储组请求, 该请求作为一条日志记录并且持久化到磁盘中, 即元数据日志. 图 5 虚线圆框中表示元数据组, 5 个节点均为元数据持有者. 以节点 5 为例, 它包含本地状态机和日志两部分内容. 日志中记录了设置存储组的操作, 而本地状态机为 Apache IoTDB 单机版系统, 负责处理日志操作.

元数据持有者通过在每个节点上保存全量的存储组信息, 解决了集群中的各个节点仅保存部分元

图 6 数据分区持有者

Figure 6 Data partition holder

数据导致无法提供如全量查询等操作的问题。

4.3 数据分区持有者

在 Apache IoTDB 中数据以存储组为粒度进行分区, 数据分区之间不存在数据交叠, 一个数据分区中的数据保存在多个节点作为副本, 这些节点称为数据分区的持有者, 且数据分区持有者所在 Raft 组中的领导人称为 Leader。

对于一个数据分区, 使所有该分区的持有者组成一个 Raft 组可以保证所有持有者数据的一致性。这个 Raft 组可以创建/删除时间序列、写入数据和查询数据。创建/删除时间序列和写入数据操作都会作为一条日志持久化到磁盘上, 即数据日志。图 6 中虚线中的节点 1, 2, 3 是同一个数据分区的持有者。每一个数据分区持有者包括数据日志和本地状态机, 数据日志记录创建/删除时间序列和写入数据操作, 而状态机为 Apache IoTDB 单机系统, 负责处理数据日志中的操作。

时间序列元数据由数据分区持有者管理可以提高系统的效率和性能。若时间序列元数据由元数据持有者管理, 由于创建时间序列语句记录在元数据日志中, 插入时间序列语句记录在数据日志中, 两个操作记录在不同的 Raft 组中导致无法保证操作的执行顺序。

当数据分区持有者 Leader 检查到某个时间序列不存在时, 有两种情况: 一种是时间序列不存在, 用户没有创建过; 另一种是时间序列已经创建成功, 但是该节点还没有收到来自 Manager 同步的日志。在这种情况下, 数据点的插入操作需要 Leader 和 Manager 之间通信来保证执行顺序, 进而保证数据一致性, 造成了性能损耗。

因此, 采用数据分区持有者存储时间序列的元数据和时序数据的方式提供了更高效的性能, 同时由于每个数据分区持有者同时还是元数据持有者, 二者结合可以提供全量查询服务。

5 分布式系统实现

5.1 处理逻辑

在进行分布式 Apache IoTDB 实现时, 我们通过将单机版逻辑和分布式版逻辑进行解耦, 从而最大程度地降低了开发复杂度。如图 7 所示, 在单机版 Apache IoTDB 中, 用户调用接口进行读写请求时将依次通过 JDBC 服务、SQL 解析服务、物理计划生成服务, 最后由写入/查询引擎进行处理, 并且写入的数据最终会形成 tsfile 文件存储在磁盘中, tsfile 是 Apache IoTDB 底层数据组织文件格式。JDBC

图 7 (网络版彩图) 节点处理逻辑的变化
Figure 7 (Color online) The changes of node processing logic

服务负责系统和用户间的交互, SQL 解析服务负责对用户请求进行解析优化并生成逻辑计划, 物理计划生成服务负责逻辑计划与系统引擎接口的适配并生成物理计划。

Apache IoTDB 的分布式系统架构是在单机版的处理逻辑上封装了一层分布式处理逻辑, 如同步日志、分布式读写逻辑等。通过将分布式处理逻辑封装成单独的框架层, 达到了尽可能复用单机版的处理逻辑并且与单机版中的功能模块基本解耦。

5.2 数据写入

用户连接任意一个节点 P 并通过批量方式进行数据写入。节点 P 负责将批量的写入操作拆分成多个子任务, 每个子任务对应于一个数据分区, 并将这些子任务进行转发、结果合并并返回给用户。具体流程如下。

- (1) 节点 P 逐条解析写入操作并记录写入操作的顺序, 并根据存储组信息计算对应的数据分区, 对每个数据分区生成一个子任务, 每个子任务负责管理所有属于对应分区的写入操作。
- (2) 节点 P 向每个数据分区的持有者 Leader 发送子任务, 图中省略查询分区 Leader 的过程, 节点通过缓存数据分区 Leader 的方式提高效率。
- (3) 数据分区 Leader 收到任务后将任务中的多个操作记录成一条日志。Leader 等到该日志被数据分区上的大多数持有者收到以后, 开始依次执行数据写入操作, 并将最后结果返回给 P。
- (4) P 根据多个数据分区 Leader 返回的结果和 (1) 中记录的写入操作顺序按序合并结果并返回给用户。

5.3 基于过滤树剪枝的数据查询

在分布式系统中, 查询需要有个节点作为“协调者”, 这个节点负责将查询分发给各个分区节点并从这些节点中收集查询结果并汇总返回。在这种情况下, 如何在用户给定的查询任务下最小化节点间的通信消耗来保障系统的高性能是分布式查询的难点。

主要通过两种方式来解决: 一是采用分批返回结果集的形式来减少节点间的网络通信次数, 这种方法是在总传输数据量相同的情况下, 通过增大每次返回的数据量来减少通信次数; 二是通过对查询

语句进行一些预处理来减少传输无效数据,从而减少了传输的总数据量. Apache IoTDB 分布式系统的查询引擎中采用分批返回结果集和对查询语句过滤条件剪枝的两种方式减少节点间通信的次数和传输的总数据量,从而保障了查询的高性能.

5.3.1 数据查询

用户连接任意一个节点 P 并发送查询任务,节点 P 负责查询语句的拆分、向各个数据分区持有者批量获取数据、本地合并结果并返回给用户. 具体流程化如下.

- (1) 节点 P 根据本地保存的存储组信息,将用户的查询分解成对若干个存储组的查询.
- (2) 节点 P 根据一致性哈希得到每个存储组所在的数据分区,对每个数据分区进行查询过滤条件的剪枝.
- (3) 节点 P 在每个数据分区中选择一个数据分区持有者并创建索引值 idx 表示读取的次数,初始为 1.
- (4) 对每个数据分区,节点 P 将查询的内容、idx 和 fetch size (批量获取的数据量) 发送给已选择的数据分区持有者,并获取返回结果.
- (5) 对于数据分区持有者返回的批量数据,若为空,表示该分区已无满足查询条件的数据;否则, P 在本地进行缓存并进行结果合并最后返回给用户.
- (6) 用户调用 hasNext(), next() 方法继续获取结果数据,若批量获取的数据未遍历完,则直接返回下一个数据;否则,将 idx 加 1,跳转到步骤 (4).
- (7) 用户调用 close() 方法,节点 P 执行状态、资源清理,并返回清理结果.

5.3.2 过滤条件树剪枝

如图 8, 对于查询 “select A1, A2, B, C from root where (A1 > 10 and B ≤ 5) and (A2 < 100 and (Time < 100 or C > 10))”, 其中 “Time” 表示时间戳, A1, A2 属于存储组 root.sg1, B 属于存储组 root.sg2, C 属于存储组 root.sg3. 假设节点 P 只有存储组 root.sg1 的数据. 在该查询中, 需要根据过滤条件筛选有效的时间戳, 因此需要从 P 节点分别获取 “select A1 from root where A1 > 10” 和 “select A2 from root where A2 < 100” 的数据作为过滤条件树叶子结点中 “A1 > 10” 和 “A2 < 100” 的数据输入, 但通过对过滤条件树进行剪枝操作可以减少从 P 点获取的数据量.

首先把过滤条件转化为一棵二叉树 (如图 8 左上角). 针对 P 进行查询剪枝时把不属于存储组 root.sg1 的过滤条件都设置为 True (如图 8 右上角). 再向上简化整棵树即为 “where A1 > 10 and A2 < 100”. 因此在 P 上针对 A1 > 10 执行的查询语句为 “select A1 from root where A1 > 10 and A2 < 100”. 相比较不剪枝的方式, 该查询将避免对不满足 A2 < 100 的时间戳的 A1 数据进行传输, A2, B 和 C 的查询同理. 通过这种方式, 能够在一定程度上将过滤条件在查询的时候下推, 从而减少了节点间需要传输的数据总量, 提高了系统的查询性能和效率.

5.3.3 最终一致性查询

Apache IoTDB 分布式系统的查询引擎除了实现强一致性的查询外, 还实现了最终一致性查询. 当用户连接一个数据分区中的任何一个节点进行查询时, 若该节点是该数据分区的 Leader, 则直接查询并返回结果; 反之, 由于该节点不确定本地数据是否为最新数据, 因此需要和分区节点的 Leader 进行数据同步的操作. 具体做法是向 Leader 询问最新的 Apply 日志编号, 等到本地的 Apply 日志编号超过该值时即可确保本地数据最新. 因此每次查询均要和 Leader 节点通信并等待至本地数据为最新数

图 8 查询条件过滤树剪枝
Figure 8 Prune tree of query filter

表 1 实验环境配置
Table 1 Configuration of experimental environment

Configuration	Description
OS	Ubuntu 16.04.4 LTS
CPU	2.40 GHz × 8
Memory	16 GB
Disk	1 T HDD
Apache IoTDB version	0.7.0
Maven version	3.5.2
JDK version	1.8

据,从而保证了强一致性的查询.

当适当降低对查询的一致性要求时,可以将每次节点与 Leader 节点通信并等待本地数据为最新数据的过程去掉来获得更高的查询性能.由于任何一个分区节点中的数据最终都会和该分区的 Leader 节点中的数据保持一致,因此这种查询方式的级别为最终一致性查询.

6 实验

本节将通过实验来探讨在不同负载下 DMM 对系统资源和性能的影响、强一致性查询和最终一致性查询的性能比较以及与时序数据库系统 KairosDB 进行性能对比.

6.1 实验准备

6.1.1 实验环境

本实验使用青云平台创建多台云主机来搭建分布式集群,每台机器配置相同,机器配置如表 1.

6.1.2 实验数据

实验采用自研工具 IoTDB-Benchmark⁴⁾ 产生数据, 并且模拟客户端对分布式版 Apache IoTDB 系统和 KairosDB 系统进行读写操作. 产生的数据是 64 位浮点数, 按照正弦曲线的方式进行变化. 所有数据按照时间戳严格递增, 不存在乱序数据的情况, 相邻两个数据点时间戳相差 5 s. 在下面的实验中集群规模为 4 ~ 10 个节点, 副本数为 3, 在 Apache IoTDB 中创建 20 个存储组, 每个存储组中创建 5000 条时间序列.

6.1.3 测试指标

写入性能由平均写入速度 (AvgWriteThroughput, AWT) 衡量, 其计算方式为

$$AWT = \frac{\text{写入总点数}}{\sum \text{批量写入方法的执行时间}}.$$

查询性能由平均延迟 (AvgQueryLatency, AQL) 和 99% 分位数延迟 (99%QueryLatency, P99QL) 来衡量. P99QL 的含义是将所有查询延迟采样值从小到大排序, 处于 99% 位置的查询延迟值. AQL 的计算方式为

$$AQL = \frac{\sum \text{每次查询的延迟}}{\text{查询总次数}}.$$

6.2 双层粒度元数据管理策略 (DMM)

6.2.1 DMM 对系统资源的影响

双层元数据管理策略相对于单层元数据管理策略 (single-layer granularity metadata management, SMM) 而言, 将时间序列的元数据信息通过数据组而不是元数据组进行同步. 由于元数据组包含集群中的所有节点, 而任意一个数据组的规模为副本数, 因此集群节点需要在内存中维护的总的元数据信息量大大降低, 且集群中为同步时间序列元数据的节点通信次数也大大降低. 设集群规模为 n , 副本数为 m , 则使用 DMM 策略时集群节点的元数据内存占用总和与集群节点间的通信次数只有使用 SMM 策略的 $\frac{m}{n}$ ($n \geq m$). Apache IoTDB 中每条时间序列的内存占用约为 2 KB.

以 10 节点 3 副本为例, 若采用单层元数据管理策略, 则每条时间序列都要在系统中的每个节点中保存并维护, 则 10 个节点的元数据总内存占用为 1.91 GB; 若采用双层粒度的元信息管理策略, 那么每条时间序列仅需要在集群中对应的数据分区中的 3 个节点中保存并维护, 这种情况下的元数据总内存占用为 0.57 GB, 有效降低了内存占用.

同理, 假设系统运行正常, 每条时间序列的元信息通过一次通信即可完成同步. 若采用 SMM 策略, 每条时间序列元信息需要传输至集群中的每个节点通信, 因此 10 个节点的集群规模下共需要 100 万次节点间通信; 若采用 DMM 策略, 每条时间序列元信息仅需要传输至对应数据分区的 3 个节点, 因此 10 个节点的集群规模共需要 30 万次节点间通信.

如图 9 和 10, 由于 DMM 策略下元数据内存占用总大小和同步元数据信息所需的通信次数和副本数相关, 与集群规模无关. 因此随着集群规模的增长, DMM 策略下这两者的资源消耗不变, 而 SMM 策略下这两者的资源消耗将会线性增长.

因此, 使用 DMM 策略降低了元数据信息占用的总内存大小和同步元数据信息所需的通信次数.

4) IoTDB-Benchmark: <https://github.com/thulab/iotdb-benchmark>.

图 9 (网络版彩图) 不同集群规模下的元数据内存占用总大小

Figure 9 (Color online) Total memory occupancy of metadata in different cluster sizes

图 10 (网络版彩图) 不同集群规模下的元数据同步通信次数

Figure 10 (Color online) The number of metadata synchronization communication in different cluster sizes

图 11 (网络版彩图) DMM 性能测试对比

Figure 11 (Color online) Performance test comparison of DMM

6.2.2 DMM 对系统性能的影响

系统实现时若将时间序列元数据信息由元数据持有者管理, 则每次数据写入操作都要对相应的元数据信息进行检查和同步. 若采用 DMM 方法, 则在创建时间序列时对相应的存储组元信息进行检查和同步. 本小节通过实验对比上述两种策略的性能.

实验中写入采用批量的方式, 每批写入 1 万条数据, 20 个客户端共写入 5000 万数据点. 图 11 中对比了在集群规模为 4, 6, 8, 10 下采用 DMM 策略和采用 SMM 策略的系统写入速度. 从图中可以看出, 系统采用 DMM 策略相比采用 SMM 策略写入速度提高 5% ~ 10%.

在本框架中, 通过数据分区持有者管理数据分区数据, 元数据持有者管理数据分区, 因此元数据信息和数据信息分属于两个 Raft 组, 从而无法保证这两个 Raft 组的日志执行顺序, 在数据写入时只能通过每次元信息查询同步的额外操作来保证操作顺序. DMM 方法通过将元信息进行双层拆分, 将部分元信息即时间序列元信息同步于数据分区中, 使得数据写入操作不需要额外进行元信息数据同步, 仅仅需要在时间序列创建时额外进行时间序列组元信息同步. 因此, 将高密度的数据写入操作的元信息同步转化为低密度的创建时间序列的元信息同步, 提升了系统的写入性能.

图 12 (网络版彩图) Apache IoTDB 和 KairosDB 写入性能对比

Figure 12 (Color online) Write performance comparison between Apache IoTDB and KairosDB

6.3 基于 Raft 协议的系统性能

本小节将探讨基于 Raft 协议的强一致性查询和最终一致性查询的查询性能对比以及分布式 Apache IoTDB 系统与 KairosDB 系统的性能对比. 在分布式框架的性能对比中, 由于不同分布式系统的存储引擎的性能相差可能会比较大, 因此不能通过直接比较查询和写入的性能指标来比较分布式框架性能, 应该对比随着集群规模的扩展分布式系统的读写性能的增长速率.

6.3.1 写入性能实验

本小节实验采用批量写入的方式, 每批写入 1 万条数据, 20 个客户端共写入 5000 万数据点. 随着系统规模的扩展, 数据组的数量将线性增长, 因此每个节点中的负载强度将下降, 从而提升系统写入性能. 如图 12 所示, 随着集群规模的扩展, 分布式 Apache IoTDB 系统写入性能的增长速率大于 KairosDB, 说明分布式 Apache IoTDB 的分布式框架的写入扩展能力优于 KairosDB.

6.3.2 查询性能实验

本小节实验测试系统在使用强一致性查询和最终一致性查询下的性能对比以及与 KairosDB 系统查询性能的对比. 查询测试包括精确时刻查询, 时间范围查询以及带值过滤条件的时间范围查询. 查询测试中共有 20 个客户端并发查询, 每个客户端查询 200 次.

实验进行了在强一致性查询 (strong consistency, SC) 下和最终一致性查询 (eventual consistency, EC) 下的查询性能对比. 如图 13 所示, 在两项查询性能指标下, 最终一致性查询的性能比强一致性查询的性能提升了近 30%. 此外, 如图 14 所示, 随着集群规模的扩展, 分布式 Apache IoTDB 系统查询性能的增长速率略大于 KairosDB, 说明分布式 Apache IoTDB 的分布式框架的查询扩展能力略优于 KairosDB.

6.4 实验总结

本小节通过实验说明了 DMM 策略对系统资源和系统性能的影响: 元数据信息占用更少的内存资源, 元数据信息同步需要更少的节点间通信且提升了系统的写入性能. 此外, 查询实验说明了使用最终一致性查询会使查询性能提升. 最后, 通过和 KairosDB 系统的读写性能对比可以看出, 分布式 Apache IoTDB 系统的读写扩展能力优于 KairosDB.

图 13 (网络版彩图) 强一致性查询和最终一致性查询性能对比

Figure 13 (Color online) Performance comparison between strong consistency query and eventual consistency query

图 14 (网络版彩图) Apache IoTDB 和 KairosDB 查询性能对比

Figure 14 (Color online) Query performance comparison between Apache IoTDB and KairosDB

7 总结

本文基于一致性哈希分区方法和 Raft 协议设计了同时支持强一致性查询和最终一致性查询的分布式框架。在框架中,为解决在分布式场景下元数据信息维护与数据写入之间的矛盾,提出了双层粒度元数据管理策略。该策略将元数据分为两层:存储组和时间序列。元数据持有者管理存储组的元信息,数据分区持有者管理时间序列元信息和数据,实验表明使用该策略与单层粒度管理策略相比,其元数据内存资源占用更少且系统的写入性能提高了 5%~10%。

在框架中,采用了一致性哈希算法将时序数据按存储组均匀地散列到分布式系统的各个节点上,并加入虚拟节点来达到负载均衡;此外,为提高系统的查询性能,采用了基于存储组的过滤树剪枝方法来减少节点间通信次数和数据量,同时提供了强一致性查询和最终一致性查询。实验结果表明该框架下的读写性能随集群规模的扩大而线性增长。

本文设计的框架对采用 Raft 协议进行分布式框架的设计具有借鉴意义,同时双层元数据管理策略也为面临如何高效管理大量元数据问题的系统提供了一种高效的解决方法。未来的工作主要为 3 方面:查询引擎的进一步优化、集群弹性伸缩以及负载均衡的研究。在目前的分布式框架中,没有考虑集群弹性伸缩的处理逻辑。此外,将尝试在按存储组分区的基础上加入时间维度来进行数据分区,进一步达到均衡负载的效果。

参考文献

- 1 Fu T. A review on time series data mining. Eng Appl Artif Intell, 2011, 24: 164–181
- 2 Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. SIGMOD Rec, 2005, 34: 42–47
- 3 Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of USENIX Annual Technical Conference, 2014. 305–319
- 4 Karger D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Proceedings of ACM Symposium on Theory of Computing, 1997
- 5 Kwon Y C, Ren K, Balazinska M, et al. Managing skew in Hadoop. IEEE Data Eng Bull, 2013, 36: 24–33
- 6 Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system. In: Proceedings of IEEE 26th Symposium

- on Mass Storage Systems and Technologies, 2010. 1–10
- 7 Dubreuil M, Gagne C, Parizeau M. Analysis of a master-slave architecture for distributed evolutionary computations. *IEEE Trans Syst Man Cybern B*, 2006, 36: 229–235
 - 8 Lakshman A, Malik P. Cassandra: a decentralized structured storage system. *SIGOPS Oper Syst Rev*, 2010, 44: 35
 - 9 Stoica I, Morris R, Liben-Nowell D, et al. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans Netw*, 2003, 11: 17–32
 - 10 Chen P M, Lee E K, Gibson G A, et al. RAID: high-performance, reliable secondary storage. *ACM Comput Surv*, 1994, 26: 145–185
 - 11 Aguilera M K. Stumbling Over Consensus Research: Misunderstandings and Issues. Berlin: Springer, 2010. 59–72
 - 12 Stonebraker M. Retrospection on a database system. *ACM Trans Database Syst*, 1980, 5: 225–240
 - 13 Naqvi S N Z, Yfantidou S, Zimanyi E. Time Series Databases and InfluxDB. Brussels: Universite Libre de Bruxelles, 2017
 - 14 Prasad S, Avinash S B. Smart meter data analytics using OpenTSDB and Hadoop. In: *Proceedings of IEEE Innovative Smart Grid Technologies-Asia*, 2013. 1–6
 - 15 Vora M N. Hadoop-HBase for large-scale data. In: *Proceedings of International Conference on Computer Science and Network Technology*, 2011. 1: 601–605
 - 16 van Renesse R, Dumitriu D, Gough V, et al. Efficient reconciliation and flow control for anti-entropy protocols. In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008. 6
 - 17 Chang F, Dean J, Ghemawat S, et al. Bigtable: a distributed storage system for structured data. *ACM Trans Comput Syst*, 2008, 26: 1–26
 - 18 DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon’s highly available key-value store. *SIGOPS Oper Syst Rev*, 2007, 41: 205–220
 - 19 Chen Z, Yang S, Tan S, et al. Hybrid range consistent hash partitioning strategy—a new data partition strategy for NoSQL database. In: *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013. 1161–1169
 - 20 Hunt P, Konar M, Junqueira F P, et al. ZooKeeper: wait-free coordination for Internet-scale systems. In: *Proceedings of USENIX Annual Technical Conference*, 2010. 8
 - 21 Lamport L. Paxos made simple. *ACM Sigact News*, 2001, 32: 18–25

The design of Apache IoTDB distributed framework

Tianan LI^{1,2,3}, Xiangdong HUANG^{1,2,3*}, Jianmin WANG^{1,2,3}, Dongfang MAO^{1,2,3},
Yi XU^{1,2,3} & Jun YUAN^{1,2,3}

1. *School of Software, Tsinghua University, Beijing 100084, China;*

2. *National Engineering Laboratory of Big Data System Software, Beijing 100084, China;*

3. *Beijing National Research Center of Information Science and Technology, Beijing 100084, China*

* Corresponding author. E-mail: huangxdong@tsinghua.edu.cn

Abstract Apache Internet of Things Database (IoTDB) is a new open-source timeseries database management system. A distributed data management system not only needs to solve the problem of metadata synchronization between nodes caused by data partition and multiple replicas, but also needs to support efficient query request processing. To solve the problem of metadata synchronization among nodes, we propose a dual-layer granularity metadata management strategy. Based on the consistency hash partitioning method and Raft protocol, we designed a distributed framework that supports both strong consistency query and eventual consistency query. Based on the single-machine version of Apache IoTDB, we carried out the system implementation and experimental test. Compared with the single-level granularity management strategy, the test results showed that the two-level granularity metadata management strategy takes less memory resources and improves the write performance by 5% ~ 10%. Also, the results showed that the read and write performance of the distributed Apache IoTDB increases linearly with the extension of cluster size.

Keywords IoT database, timeseries, distributed framework, Raft protocol, metadata, strong consistency, eventual consistency

Tianan LI was born in 1996. He is an M.S. candidate at Tsinghua University. His research interests include distributed database and big data system management.

Xiangdong HUANG was born in 1989. He obtained his Ph.D. degree from Tsinghua University in 2017. He is currently an assistant researcher at Tsinghua University. His research interests include big data system management and modeling.

Jianmin WANG was born in 1968. He obtained his Ph.D. degree from Tsinghua University in 1995. He is currently a professor at Tsinghua University. His research interests include big data and knowledge engineering.

Dongfang MAO was born in 1995. He is an M.S. candidate at Tsinghua University. His research interests include distributed database and time-series data management.