Apache IoTDB 集群扩展机制研究

(申请清华大学工程硕士专业学位论文)

培 养 单 位：　软件学院

工 程 领 域：　软件工程

申　请　人：　李 天 安

指 导 教 师：　王 建 民　教　授

二〇二一年五月

Research on scalability mechanism of Apache IoTDB cluster

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the professional degree of

Master of Engineering

by

Li Tianan

(Software Engineering)

Thesis Supervisor: Professor Wang Jianmin

May, 2021

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

王爱成	高级工程师	军事科学院系统工程研究院
宋韶旭	副教授	清华大学

答辩委员会名单

主席	王爱成	高级工程师	军事科学院 系统工程研究院
委员	王建民	教授	清华大学
	王朝坤	副教授	清华大学
	张力	教授	清华大学
	刘强	副教授	清华大学
	刘英博	副研究员	清华大学
	宋韶旭	副教授	清华大学
秘书	马再超	助理研究员	清华大学

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）按照上级教育主管部门督导、抽查等要求，报送相应的学位论文。

本人保证遵守上述规定。

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

随着物联网的高速发展，海量时序数据的高效管理成为了研究热点。Apache IoTDB 是一个新型的时序数据库管理系统，其分布式框架能够提供容错能力，但是还不支持集群扩展功能。由于集群的可扩展性是提升分布式系统高可用性和性能的必要功能，因此本文研究的内容是在 Apache IoTDB 分布式框架上设计并实现集群扩展功能。

Apache IoTDB 分布式框架基于一致性哈希和 raft 协议，组成多个 raft 数据组管理数据。系统在集群扩展时会进行多数据组成员变更、数据分区信息的修改和数据迁移，因此集群扩展机制要满足以下需求：在多数据组成员变更期间保证成员变更的安全性；在修改分区信息期间保证数据的安全性，不能出现已写入数据丢失的情况；在数据迁移时提供高效的数据迁移策略，减少对系统资源的使用。

本文的贡献有以下三点:

1. 针对需要保证多数据组成员变更安全性和数据安全性的需求，提出了一种基于一致性哈希和 raft 协议的两阶段集群扩展方法（Two-stage method of cluster scalability，简称 TSM）。第一阶段将集群成员变更信息同步于所有数据组中并执行数据组的预成员变更；第二阶段执行数据组的正式成员变更和数据迁移。该方法避免了数据组出现脑裂的情况，保证了成员变更的安全性。此外，该方法保证了在数据迁移前集群中所有写入请求都会由新的数据分区持有组处理，从而保证了数据的安全性。
2. 针对需要提供高效数据迁移策略的需求，提出了基于哈希槽和时间片的文件级数据迁移方法（File-level data migration method，简称 FDM）。该方法采用哈希槽分区策略，每个节点以槽为单位管理数据。此外，该方法利用了存储引擎按照时间片进行数据管理使得每个数据文件的所有数据都不会跨越两个槽的特性，在进行数据迁移时直接对整个文件进行传输和加载，从而避免了数据查找和重建元信息等过程，减少了对系统资源的使用。
3. 在 Apache IoTDB 的分布式框架上实现了集群扩展机制，并设计实验对集群扩展机制进行功能和性能测试。首先对集群扩展机制的效率进行了测试，测试结果表明集群扩展的成员变更可以在秒级完成，并验证了文件级别的数据迁移方案的高效性。其次，测试并分析了集群扩展机制对系统性能的影响。

关键词：时间序列；分布式；集群扩展；数据迁移；Raft 协议

Abstract

With the rapid development of the Internet of Things, efficient management of massive time series data has become a research hotspot. Apache IoTDB is a new type of time series database management system. Its distributed framework can provide fault tolerance, but it does not support cluster scalability. Since the scalability of the cluster is a necessary function to improve the high availability and performance of the distributed system, the content of this paper is to design and implement the cluster scalability function on the Apache IoTDB distributed framework.

The Apache IoTDB distributed framework is based on consistent hashing and the raft protocol to form multiple raft data groups to manage data. The system will change the members of multiple data groups, modify the data partition information and migrate data when the cluster is expanded. Therefore, the cluster scalability mechanism must meet the following requirements: ensuring the security of member changes during the change of members of multiple data groups; ensuring the security of the data during the modification of the partition information without loss of written data; providing efficient data migration strategies during data migration to reduce the usage of system resources.

The contributions of this paper are as follows:

1. In order to guarantee the security of changing multiple data group members and data security, a two-stage method of cluster scalability (TSM) based on consistent hash and raft protocol is proposed. In the first stage, the cluster member change information is synchronized in all data groups and the pre member change of the data group is executed; in the second stage, the formal member change and data migration of the data group are executed. This method avoids split-brain in the data group and ensures the security of member changes. In addition, it ensures that all write requests in the cluster will be processed by the new data partition holding group before data migration, thus ensuring data security.
2. In order to provide an efficient data migration strategy, a file-level data migration method (FDM) based on the hash slot and time slice are proposed. In this method, a hash slot partition strategy is adopted, and each node manages data by slot. In addition, this method takes advantage of the storage engine's ability to manage data according to time slices so that all data in each data file does not cross two slots.

When data migration is performed, the entire file is directly transferred and loaded, which avoids the process of data search and meta information reconstruction, and reduces the use of system resources.

3. The cluster scalability mechanism is implemented on the distributed framework of Apache IoTDB, and experiments are designed to test the function and performance of the cluster scalability mechanism. Firstly, the efficiency of the cluster scalability mechanism is tested. The results show that the member changes of the cluster scalability can be completed in seconds. It also verifies the efficiency of the file-level data migration scheme. Secondly, the impact of cluster scalability mechanism on system performance is tested and analyzed.

Keywords: time series; distributed system; scalability; data migration; raft protocol

目 录

插图和附表清单

第 1 章　引言

1.1　研究背景与意义

随着互联网时代的高速发展，越来越多的设备不断地产生数据，这些海量数据被用于收集、存储、研究和分析，构成了物联网[1]应用的基础。由于这些数据往往伴随着时间戳来标识数据产生的时间，因此被称为时间序列数据[2]（简称时序数据）。时序数据有以下几个特点：

1. 数据产生速度快。例如国际风电标准 IEC 61400-25 规定，每台风机每秒需要采集 225KB 的工业数据，甚至在一些极端工况中采集的频率达到 8KHz。
2. 数据总量大。工业机器设备长期处于 7*24 小时不间断的高速采集状态，数据规模不断地增大。
3. 数据种类丰富。数据来源是多种多样的，包括股市、天气预报、医疗、化工等；数据类型包括浮点数、字符串、整数、布尔值和复合数据类型（如 GPS 的定位数据包括经度和纬度信息）等。

针对时序数据的这些特点，传统数据库已无法满足需求[3]，因此需要专门用于处理时序数据的时间序列数据库[4]（简称时序数据库）。目前比较有名的时序数据库有 Influxdb[5]、OpenTSDB[6]、KariosDB[7]和 Apache IoTDB[8]。Apache IoTDB 是一个新型的开源时序数据库管理系统，该系统在 2020 年 9 月成为 Apache 顶级项目。

随着业务复杂度和处理的数据量不断增加，Apache IoTDB 单机版系统已不能满足一些应用场景下的需求。因此，为了能够满足日益增长的业务需求以及高效扩展系统性能，Apache IoTDB 设计并实现了分布式框架[9]。该框架基于 raft 协议[10]和一致性哈希算法[11]，将集群划分为一个 raft 元数据组和多个 raft 数据组。元数据组包含集群中的所有节点，数据组包含副本数个节点。

在分布式系统中，当系统负载发生变化、节点硬件需要升级或者某些节点出现故障时，可以在不影响系统可用性的情况下动态地增删节点的功能称为分布式集群的可扩展性。但在当前的 Apache IoTDB 分布式框架中，集群初始化时每个节点通过读取本地的配置参数来获取集群的节点列表，集群节点间不交互集群节点的配置信息，且集群初始化后在运行过程中不支持动态地增删节点，因此不具备可扩展性。

由于支持集群的可扩展性是提升分布式系统高可用性和性能的必要功能，因此本文研究的内容是在 Apache IoTDB 分布式框架上设计并实现集群扩展功能。

1.2　研究内容

在 Apache IoTDB 分布式框架中，增删节点会导致集群多个 raft 数据组的成员变更。在集群中，以每个节点为起始点，在哈希环上顺时针找到副本数个节点即可组成一个数据组，即 N 个节点的集群共有 N 个数据组，每个数据组内使用 raft 协议来保证一致性。当加入新节点时集群中会增加一个以新节点为起始点的数据组，当删除旧节点时以被删除节点为起始点的数据组会从集群中删除。

增删节点还会导致系统数据分区的变化并造成数据迁移。系统在进行集群扩展时会进行数据分区信息的修改，每个数据组负责的分区会发生变化，当加入新节点时系统会让新增的数据组分摊其他数据组的负载；当删除旧节点时系统将被删除数据组的负载分配给其他数据组。此外，分区信息的变化会造成数据迁移，旧分区管理组会将数据迁移至新分区管理组。

综上，在设计集群扩展机制时主要进行以下几方面的研究：

1. 保证多数据组成员变更的安全性。集群进行一次增加节点或者删除节点操作时，由于数据组成员由一致性哈希环决定，因此会导致多个数据组的成员变更。Raft 组的成员变更可能会导致**脑裂** (脑裂是指 raft 组出现多 leader 的情况) 现象。因此集群扩展机制需要保证多 raft 组成员变更的安全性。
2. 保证数据的安全性。系统在修改分区信息的过程中，没有应用新分区信息的节点会将写入请求发送给旧分区管理组，应用了新分区信息的节点会将写入请求发送给新分区管理组。旧分区管理组在应用新分区信息前仍会处理该分区的写入请求，如果数据迁移时没有将这部分数据也迁移至新分区管理组，就会造成数据丢失，因此这期间集群扩展机制要保证数据的安全性。
3. 设计高效的数据迁移方案。当集群增删节点时，由于分区信息的变化，一些分区的数据会迁移至其他的数据节点。数据迁移的过程分为查找数据、传输数据和加载数据三步，其中每一步都会消耗系统的资源，因此数据迁移的效率将会影响系统的整体性能。

1.3　主要贡献

本文的主要贡献体现在以下几个方面：

1. 针对需要保证多数据组成员变更安全性和数据安全性的需求，提出了一种基于一致性哈希和 raft 协议的两阶段集群扩展方法（Two-stage method of cluster scalability，简称 TSM）。第一阶段将集群成员变更信息同步于所有数据组中并执行数据组的预成员变更；第二阶段执行数据组的正式成员变更和数据迁移。该方法避免了数据组出现脑裂的情况，保证了成员变更的安全性。此

外，该方法保证了在数据迁移前集群中所有写入请求都会由新的数据分区持有组处理，从而保证了数据的安全性。

2. 针对需要提供高效数据迁移策略的需求，提出了基于哈希槽和时间片的文件级数据迁移方法（File-level data migration method，简称 FDM）。该方法采用哈希槽分区策略，每个节点以槽为单位管理数据。此外，该方法利用了存储引擎按照时间片进行数据管理使得每个数据文件的所有数据都不会跨越两个槽的特性，在进行数据迁移时直接对整个文件进行传输和加载，从而避免了数据查找和重建元信息等过程，减少了对系统资源的使用。
3. 在 Apache IoTDB 的分布式框架上实现了集群扩展机制，并设计实验对集群扩展机制进行功能和性能测试。首先对集群扩展机制的效率进行了测试，测试结果表明集群扩展的成员变更可以在秒级完成，并验证了文件级别的数据迁移方案的高效性。其次，测试并分析了集群扩展机制对系统性能的影响。

1.4　组织结构

本文共分为八个章节，每个章节的内容如下：

第 1 章为引言部分。主要介绍时序数据的特点和 Apache IoTDB 时序数据库管理系统，并指出 Apache IoTDB 现有分布式框架不支持集群扩展功能，最后引出本文的研究内容和主要贡献。

第 2 章介绍了相关的研究工作。首先介绍了分布式系统的两种架构，其次对副本一致性协议进行了详细描述，然后介绍了集群扩展中的数据分区和数据迁移方法，最后对时序数据库 Apache IoTDB 进行了详细介绍，包括基本概念、TsFile、应用架构和系统架构。

第 3 章介绍了当前 Apache IoTDB 的分布式框架，包括双层粒度元数据管理方法和数据分区策略等。

第 4 章介绍了 Apache IoTDB 集群扩展机制的详细设计。首先分析了 Apache IoTDB 集群扩展机制设计上的要点和难点，其次详细地介绍了集群扩展机制的方案设计并进行了安全性证明，最后给出了集群扩展机制下系统增加节点、删除节点和读写流程。

第 5 章介绍了 Apache IoTDB 集群扩展机制的方案实现，包括两阶段方法、数据重分区以及数据迁移三部分的具体实现，给出关键实现的流程图并描述了异常情况下的处理。

第 6 章为实验部分。对集群扩展机制进行多方面的实验验证和分析，包括对集群扩展机制的效率测试、集群扩展机制对系统性能的影响测试以及和 KairosDB

的对比测试。

第 7 章对比了其他系统的集群扩展机制。通过和 Cassandra 系统、TiKV 系统的集群扩展机制进行对比，分析了本文设计的集群扩展机制的优势和不足。

第 8 章介绍了本文的工作总结和不足之处，并指出未来工作的主要方向。

第 2 章　相关研究综述

本章首先介绍了分布式系统架构和副本一致性协议，然后介绍了集群扩展相关的数据分区和数据迁移方法，最后详细介绍了 Apache IoTDB 系统。

2.1　分布式系统概述

随着物联网的兴起、计算资源的快速增长以及基础设施成本的不断下降，分布式系统开始广泛地应用在各个领域。分布式系统的架构主要有两种[12]：一种是以 HDFS[13]为代表的主从架构[14]；一种是以 Cassandra[15]为代表的 P2P 对等架构[16]。

在主从架构中，集群包括一个主节点和多个从节点。主节点负责管理所有的从节点以及数据的元信息，从节点负责具体的数据读写请求和数据管理。在处理用户的请求时，对于写入操作，客户端会首先将请求发送给主节点，主节点根据分区的策略将负责处理该请求的从节点信息发送给客户端，客户端再对给定的从节点进行请求完成写入操作；对于查询操作，客户端也会首先将请求发送给主节点，主节点在能够处理该请求的节点列表中选择一个合适的节点返回给客户端，客户端向给定的从节点发起请求。

因此，主从架构中主节点负责管理从节点和路由功能，随着负载和集群规模的不断增大，主节点的性能将会成为整个分布式系统的瓶颈。当一个主节点由于故障宕机或者网络不可达等原因导致不可用时，系统会重新选择一个新的节点作为主节点，这个过程一般由分布式协调服务来进行，如在 HBase[17]中采用 Zookeeper[18]来监控集群所有节点的状态并负责主节点的选举工作和主节点数据的热备份等功能。

在 P2P 对等架构中，所有节点是对等的，节点与节点之间通过流言（Gossip）协议[19]来获取集群中所有节点的状态以及集群成员的变更等信息。在主从架构中，客户端请求都需先向主节点询问相关信息再向对应的从节点发起读写请求；而在对等架构中，客户端可以将请求发送给集群中的任何一个节点，每个节点都可以对客户端请求进行处理。当处理客户端写入请求时，收到请求的节点会作为“协调者”节点对这些请求进行处理，通过分区算法（如在 Cassandra 中使用一致性哈希算法）得到处理该请求的副本数个节点并将请求转发给这些节点。当一定数量的节点（由用户配置）处理完该请求时，“协调者”节点就可以向客户端返回执行结果。

相较于主从架构，对等架构将主节点的职能分散到各个节点中，因此避免了

单点瓶颈的问题，但是却引入了共识问题。在主从架构中，由于主节点管理着所有的从节点，各个节点通过和主节点进行通信来保证对于集群的认知是一致的。但在对等架构中，保证所有节点对于集群的认知达到一致是困难的。比如当集群增删节点时，集群中部分节点已经感知到集群成员的变化，其他节点还没有感知到这个信息，此时向这两类节点发起请求会得到不一致的请求结果。此外，在进行查询请求时还会引发数据一致性[20]的问题，此时为保证强一致性就需要采用法团机制[21]，满足读取副本数与写入副本数之和大于副本数量的约束才能得到最新最全的数据。

2.2　副本一致性协议

在分布式系统中，通常引入多副本机制来保证系统的高可用性和数据的高可靠性。多副本机制就是将同一份数据备份在多个节点中，这样当某个节点不可用时，其他节点上保留着的副本数据还可供用户使用，保证了可用性的同时还极大程度地降低了数据丢失的风险。同一份数据要保存在集群中的不同节点上且要保持数据的一致性，这种一致性叫做副本一致性。需要注意的是，副本一致性与传统数据库中事务 ACID[22]的一致性不同[23]：副本一致性是分布式领域中 CAP 理论[24]中的一致性，具体指的是多个数据副本之间的一致性；事务 ACID 中的一致性指的是数据状态的一致性约束，比如银行账户余额不能为负数，一次转账交易中两个账户的金额总数不变等。

为保证副本的一致性，一些系统采用了保守的策略来完成，即一次数据写入只有当所有的副本都写入成功才算完成，比如 HDFS 中采用流水线式的写入方法来保证副本的一致性。这种方式虽然能保证副本的强一致性，但是等待全部副本写入成功的过程严重影响了系统的性能。另一些系统则在数据写入一定数量副本后就认为请求完成，其他副本中的数据写入将在后台进行。在这种方式下系统获得了高性能，但是引入了数据共识的难题：将一份数据复制到其他节点的过程中，由于节点故障或者网络延迟等原因这些节点什么时候可以完成数据的复制是不可确定的。在这种情况下同一份数据的多个副本之间产生了差异和冲突，分布式系统需要通过共识协议来保证各个副本对这份数据的共识是一致的。下面介绍比较常用的两个共识协议：paxos[25]协议和 raft 协议。

2.2.1　Paxos 协议

Paxos 是 Lamport 在 1990 年提出的共识协议，分布式锁服务 chubby[26]底层实现就是基于 paxos 协议，该协议将分布式节点划分为三个角色：提议者（proposers）、

决策者（acceptors）和学习者（learners），在工程实现中一个节点可以同时充当多个角色。提议者是提案的发起者，每个提案包括提案编号和提案内容，不同提议者提案的提案编号要保证不冲突，即每个提案的提案编号是全局唯一的。决策者是提案的接受者，一个提案被接受当且仅当超过半数的决策者接受了该提案。当有提案被接受时说明分布式系统就提案内容达成了共识，学习者角色就是将已达到共识的提案内容同步给其他未确定的决策者。整个协议的过程分为两个阶段：预提案阶段和正式提案阶段。

在预提案阶段，提议者生成一个提案编号，并向所有的决策者发起预提案请求，预提案请求不包含提案内容。决策者收到预提案请求时，如果提案编号小于收到过提案的最大编号，则拒绝该提案；否则，更新本地收到过提案的最大编号，接受该预提案并返回给提议者接受过的提案中编号最大的正式提案（如果没有则返回 null）。提议者如果没有收到超过半数决策者的接受响应，则重新生成一个更大的编号并重新发起预提案；否则进入正式提案阶段。

在正式提案阶段，决策者的返回结果中如果存在已被接受的正式提案，则正式提案的内容为所有返回结果中编号最大的已被接受的提案内容；如果不存在，那么提议者可以自己决定提案内容。决策者收到正式提案后，如果提案编号大于等于收到过的提案的最大编号，则接受该提案并更新本地收到过提案的最大编号，否则拒绝。当提议者收到了超过半数决策者的接受响应，那么提案内容就可以被正式确定，即集群对于提案内容达成了共识；如果没有则重新进入预提案阶段。

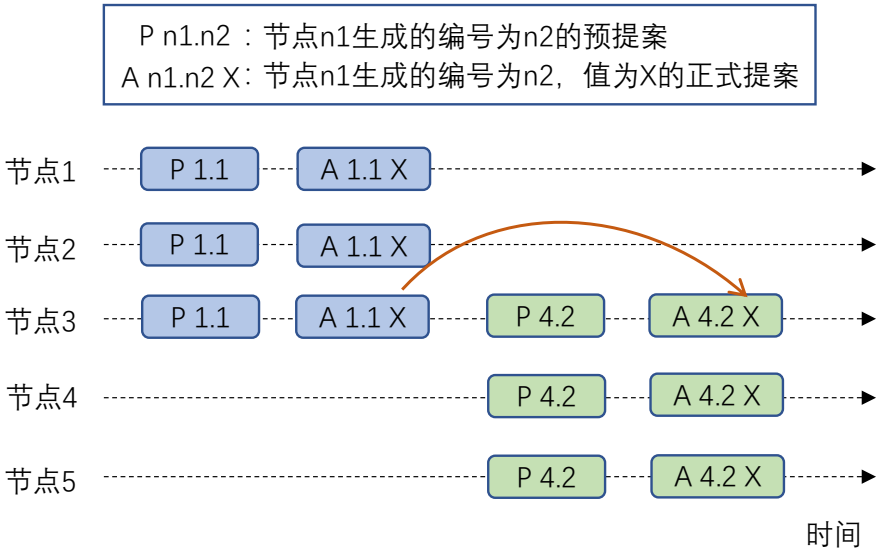


图 2.1　Paxos 算法示例流程

下面以图 2.1 中的示例进行说明，在 5 个节点的集群中，节点 1 首先发起了编号为 1 的预提案请求，被节点 1、2 和 3 接受，满足超过半数决策者接受请求的要

求。由于决策者的返回结果中不包含已被接受的正式提案，说明集群还没有对提案内容达成共识，因此节点 1 可以自己决定提案内容，比如 X。然后节点 1 发起了编号为 1 并且值为 X 的正式提案请求，被节点 1、2 和 3 接受，由于满足超过半数决策者接受请求的要求，因此集群对提案内容达成了共识，后面的所有提案内容都会被设置为 X。比如当节点 4 发起编号为 2 的预提案请求时，节点 3 会将本地已经接受的提案内容发给节点 4，因此节点 4 发起的编号为 2 的正式提案请求中值会被设置为 X。

整个 paxos 协议过程的核心在于保证了当集群中超过半数节点同意了某个提案内容后，已经被确定的提案内容不会被后面的提案所更新。在图 2.1 的示例中，当集群对于提案内容 X 达成共识后，后面提案的值都只会为 X，从而达到了共识的目的。在正式提案阶段，提案者可以自己决定提案内容当且仅当超过半数决策者不存在已被接受的正式提案，否则即使有新的提案其内容也会采用已达成共识的提案内容。

2.2.2　Raft 协议

Raft 协议是 Diego 等人在 2013 年提出的共识协议，它将共识问题拆分为日志复制、安全性和领导者选举三个子问题。在 raft 协议中将系统中的节点划分为三个角色：跟随者（follower）、领导者（leader）和候选者（candidate），每个节点在任意时刻只能扮演一个角色。Raft 将时间划分为一系列任期，任期开始时，候选者之间竞争领导者角色，选举过程保证至多产生一个领导者。当有候选者赢得选举时，它就成为了领导者，其他节点成为了跟随者。如果由于选票被平分导致无法选出领导者，那么将会开启下一个任期并重新进行选举。

Raft 通过心跳来触发选举阶段，集群正常运行期间有一个领导者和多个跟随者，领导者通过心跳来管理跟随者。当一个跟随者超过一定时间没有收到领导者的心跳就会触发新任期的选举，在整个选举阶段 raft 保证在每个任期中每个节点只会进行一次投票。投票的安全性原则是候选人的日志要比自己的日志新，当一个候选人收到了超过半数节点的投票时它就成为了领导者。

当一个领导者被选举出来后就可以处理客户端请求，领导者将客户端请求作为一条日志加入到本地的日志列表中，然后将这条日志分发给所有的跟随者。当日志被一半以上的跟随者收到后，领导者将这条日志提交并返回执行结果。每一条日志由任期、有序编号和请求内容组成，领导者的职责就是使所有跟随者的日志和自己完全一致。

在实际使用中，当日志列表长度超过一定大小时，raft 协议会在某个检查点上对系统进行快照，由于快照中包含了检查点之前的日志，因此快照完成后可以将

检查点之前的日志删除。当领导者和跟随者进行同步时，如果跟随者日志落后太多，那么领导者会直接将最新的快照发送给跟随者，这个过程叫做 catch up。

Raft 协议通过将复杂的问题拆分为多个相对简单的子问题来解决并强化了领导者在集群中的作用和地位，因此相较于 paxos 协议更容易理解和实现。Raft 协议目前已有比较多的开源实现和项目应用，如 TiDB[27]、Etcd①和 Apache IoTDB 等。

2.3　集群扩展

在分布式系统中，可扩展性指的是系统可以动态增删节点来提升或者降低性能而不影响系统可用性的特性，可分为垂直扩展性和水平扩展性。垂直扩展性指的是集群中单个节点的硬件可以进行升级或者降级，包括磁盘容量、内存和网络带宽等。水平扩展性也称为集群扩展性，指的是在不中断集群服务的情况下，通过增删集群节点的方式来提升或者减少系统提供服务的规模和性能。

2.3.1　数据分区

集群扩展功能与数据分区策略密切相关，因为它依赖于分区算法将数据分配到各个节点中去，当增删节点后分区算法也需要决定数据分区的重分配情况，因此数据分区策略的选择将影响集群扩展的性能。数据分区的方式主要分为两种：

1. 范围分区。范围分区由于数据分区和节点信息直接映射，因此需要通过查找表记录分区信息，主要应用有 BigTable[28]、HBase 和 MongoDB②等系统。
2. 哈希分区。哈希分区需要通过哈希函数计算哈希值然后在哈希值与节点之间再进行映射，具体实现包含节点取余、一致性哈希和哈希槽等，Cassandra、Dynamo[29]等系统就选择了一致性哈希的方式作为数据分区的策略。此外，还有一些混合分区方法[30]被提出。

2.3.1.1　范围分区

范围分区是按照数据的某个特征划分为多个区间，每个节点负责一个或多个数据区间。每个区间的大小是不固定的，因为有些区间的数据往往比较集中，这些区间的数据叫做热点数据。范围分区的优势在于可以根据负载情况进行动态调整。如图 2.2 所示，将数据按照选定的特征进行范围分区后，由于数据区间 (666,1000] 的数据比较集中，因此节点 2 的负载要比其他节点更大。这种情况下，当增加节点 3 时分区 (666,1000] 会分裂成两个区间，可以达到负载快速均衡的目的。

由于范围分区策略中每个节点负责的分区数和分区大小不是固定的，因此需

① Etcd. https://etcd.io/
② MongoDB. https://www.mongodb.com/

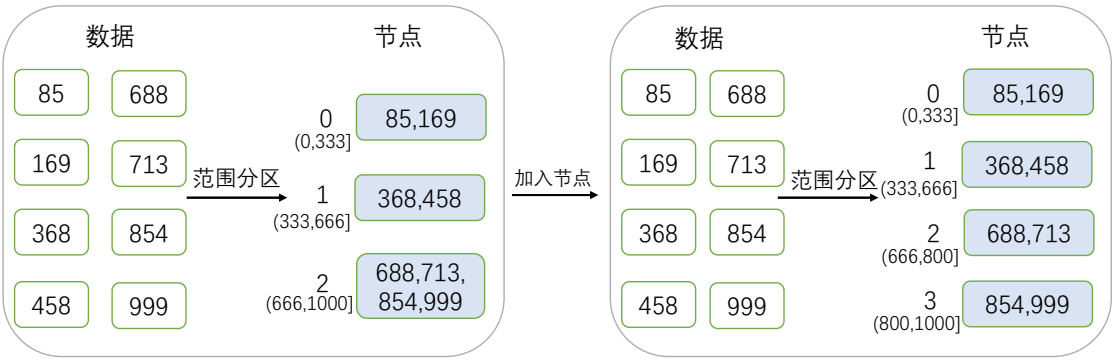


图 2.2　范围分区方法加入节点

要通过查找表记录分区信息，维护数据分区与数据节点之间的映射关系，分区数量越多查找表占用内存量越大。当增删节点导致分区区间的分裂和合并时，查找表记录的分区信息也要进行相应的更新和修改，因此范围分区的元数据管理比较复杂。

2.3.1.2　节点取余

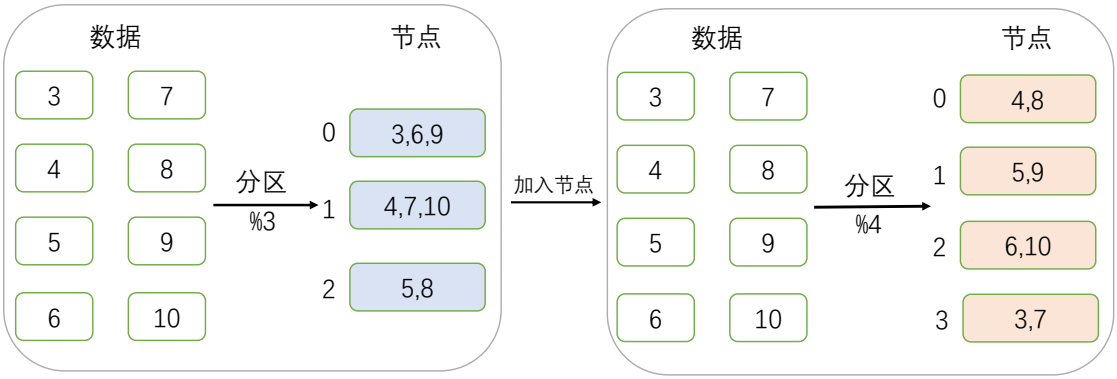


图 2.3　节点取余分区方法加入节点

传统哈希的方式是通过对数据的某个特征计算哈希值，然后将哈希值和集群中的某个节点建立映射关系。节点取余策略就是建立映射时采用对哈希值取模的方式，模数取节点数。这种映射方式的优势是需要记录的分区信息少，只需要知道哈希函数和集群节点的个数即可实现数据的分区。

节点取余分区方法有一个明显的缺点就是集群扩展性差，当增加或者删除一个节点时，由于集群中节点的数量发生了变化，因此要重新计算数据与节点的映射关系，从而导致大部分数据需要进行数据迁移。如图 2.3，当集群节点数由三个增加至四个会导致集群中所有数据的存储位置都发生了变化，因此所有的数据都要进行数据迁移。实际应用中在扩容时通常采用翻倍扩容的方式来减少迁移的数据量。

2.3.1.3　一致性哈希

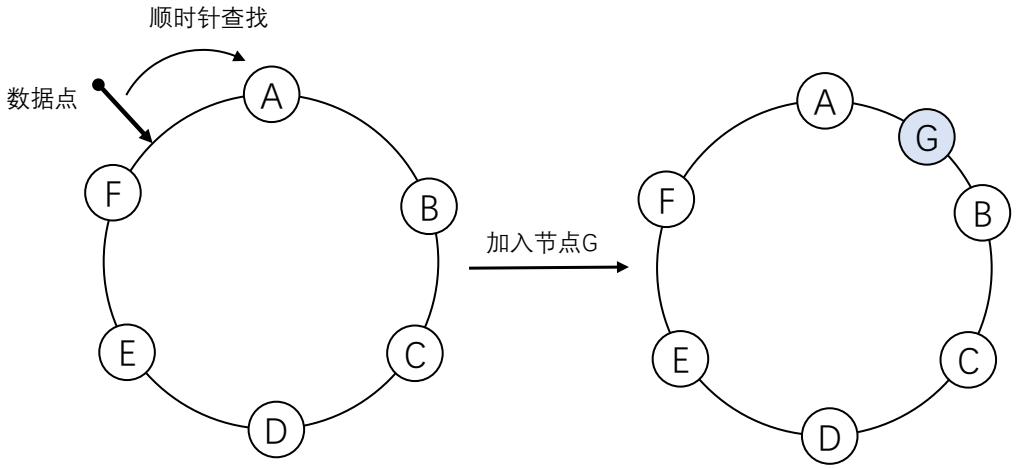


图 2.4　一致性哈希环分区方法加入节点

目前分布式系统采用的哈希分区策略普遍基于一致性哈希分区方法。一致性哈希通过哈希函数将节点和数据映射到相同的地址空间中，地址空间形成一个首尾相连的封闭环，即地址空间的最大值是最小值的前驱。如图 2.4 左侧所示，在哈希环上数据路由的方式是每个数据点先通过哈希函数计算出哈希值，然后找到环上相应的位置，最后沿着环按顺时针找到的 n 个节点保存该数据，n 为副本数。因此，每个节点在环中负责的数据范围为 (第 n 个前驱节点的位置，该节点的位置]，比如以三副本为例图中 B 节点负责的数据范围为 (E,B]。

一致性哈希分区方法的优势是在集群扩展时仅影响哈希环中后续副本数个节点，迁移数据量大幅度降低，不需要像节点取余分区那样大规模迁移数据。如图 2.4 右侧所示，以三副本为例，G 节点将代替 B 节点管理 (E，F] 区间的数据、代替 C 节点管理 (F,A] 区间的数据、代替 D 节点管理 (A,G] 区间的数据。

一致性哈希方法有以下两个缺点：

1. 由于集群节点在哈希环中的位置通过哈希函数确定，因此哈希函数的选择将严重影响集群的负载均衡和性能。不理想的映射结果会导致节点在环中分布不均匀，从而产生数据倾斜，导致某些节点负责管理大量的数据，进而成为性能瓶颈节点[31]。如图 2.5 左侧所示，由于 A、B、C 三个节点在哈希环中的位置分布不均匀，导致节点 B 负责的数据范围远小于节点 A 负责的数据范围，从而造成负载的不均衡。
2. 一般集群中的节点存在异质性，即既存在高配置机器，又存在低配置机器，但普通的一致性哈希算法并不能对集群节点的异质性进行考虑。

为了解决上述问题，一些分布式系统如 Cassandra、Dynamo 等采用了虚拟节

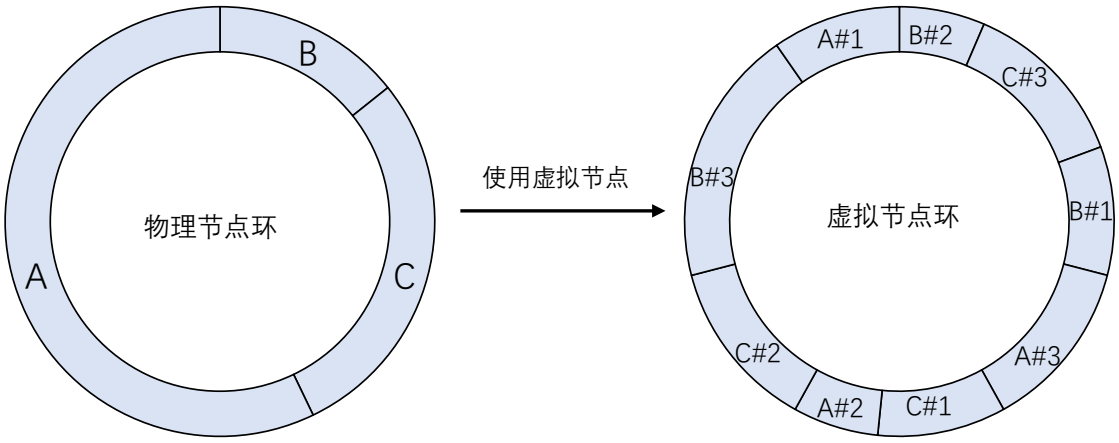


图 2.5　一致性哈希使用虚拟节点

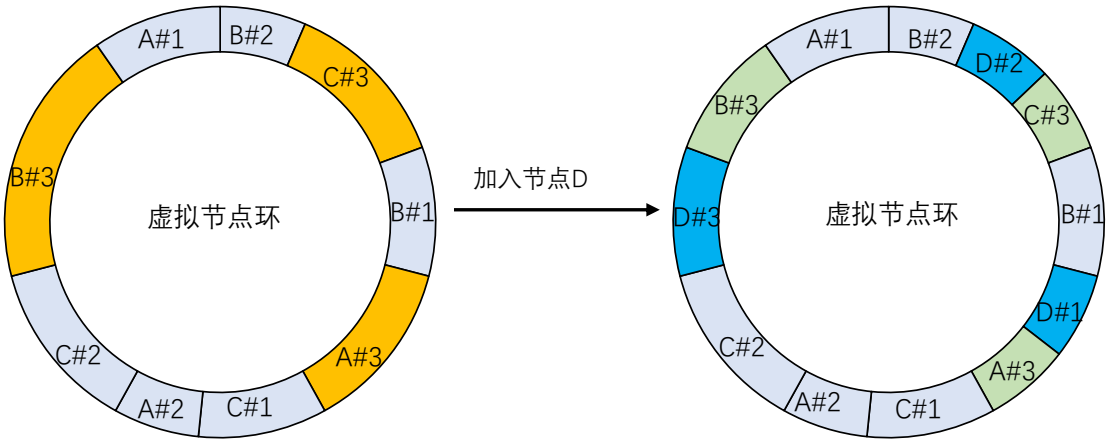


图 2.6　使用虚拟节点方法加入节点

点技术。该方法的基本思想是将一个物理节点虚拟成若干个节点并全部散列在哈希环上，虚拟节点的数量可以根据机器的性能而定。如图 2.5 右侧所示，将每个节点虚拟出 3 个虚拟节点后，哈希环上的节点分布更加均匀。在数据写入和查询时首先在哈希环上找到对应的虚拟节点，然后再向对应的物理节点进行请求。它有以下两点优势：

1. 数据分配得更加均匀；删除节点时该节点可以将负载均匀地分散到集群的各个节点中，同样当增加节点时该节点将从集群中的其他节点均匀地获取数据，如图 2.6 所示当加入节点 D 时，D 将分别接管 A#3、B#3、C#3 的部分区间数据。
2. 考虑了节点间的性能异质，对高性能的节点可以将虚拟节点的数量设置得更多，从而处理更多的负载请求。

但加入虚拟节点后的一致性哈希分区方法还是存在两个问题：

1. 由于每个节点负责的分区范围随着增删节点而变化，因此每个节点进行数据

存储时不能将各个分区的数据分开管理，导致数据迁移时节点需要先对数据进行扫描找出待迁移的数据然后再进行数据传输。由于分布式系统中的数据量比较大，因此对数据进行扫描将消耗很多的系统资源并增加了数据迁移的时间，增大了集群扩展的代价。

2. 该方法不适用于集群节点较少的情况，在集群节点较少时增删节点将导致大范围影响一致性哈希环中现有数据的映射关系。

2.3.1.4　哈希槽

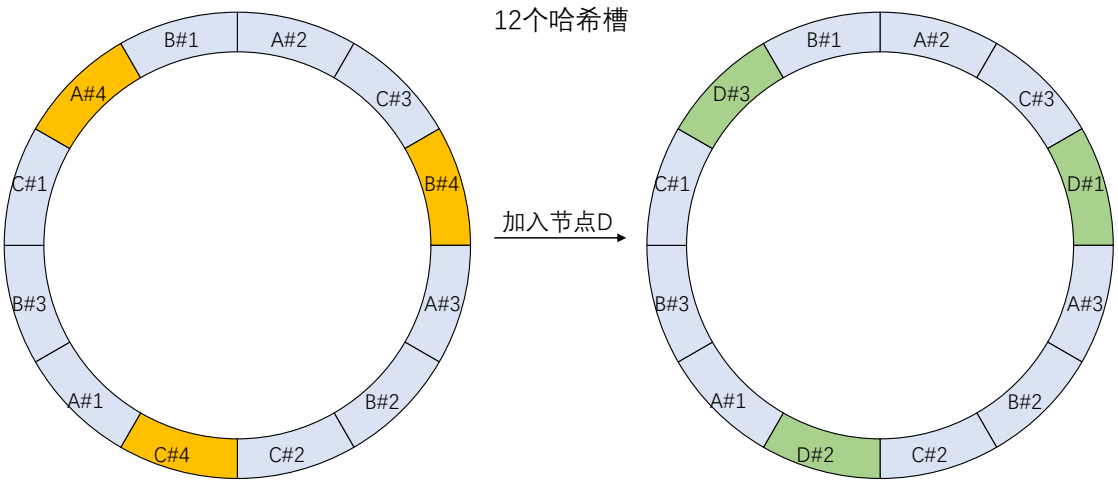


图 2.7　哈希槽分区方法加入节点

由于一致性哈希分区方法有上述缺点，在 Dynamo 和 Redis[①]系统中使用了另一种改进方法，该方法将哈希环等分为 Q 个区间，每个区间称为槽。集群中的各个节点以槽为单位管理数据，为了减少数据的倾斜性槽数 Q 远大于集群中的节点数。假设集群节点数为 N，则各个节点管理的槽数为 Q/N，当集群删除节点时将负责的槽均匀地分配给集群中的其他节点；同理当新增节点时将均匀地从其他节点获得一定数量的槽。如图 2.7 所示，将哈希环等分为 12 个哈希槽，3 个节点的集群中每个节点负责 4 个哈希槽。当增加一个节点时，新节点从集群中的每个节点获取一个哈希槽，因此集群中每个节点负责 3 个哈希槽。哈希槽分区方法的优势有以下三点：

1. 哈希环中的每个槽的范围不会随着集群的扩展而变化，因此数据可以直接按照槽为单位分开进行存储，这样当集群扩展时不需要经过扫描即可直接定位出需要迁移的数据。
2. 解决了一致性哈希分区方法在集群节点较少时的负载不均衡问题。

① Redis. https://redis.io/

3. 当集群出现热点数据时，槽分区方法可以根据负载情况和机器性能动态地调整各个节点管理的槽数来达到均衡。

2.3.2　数据迁移

对于采用存储计算分离架构的分布式系统，集群扩展计算节点不会带来存储集群的数据迁移，因此只需要修改数据的元信息与节点之间的映射关系即可。例如 HBase 底层数据存储在分布式存储系统 HDFS 上，集群扩展时只需要在分区信息中修改数据元信息和节点之间的映射关系。

对于其他的分布式系统，集群扩展会带来必要的数据迁移。数据迁移的过程包括数据的查找、节点间数据的传输和数据的加载，由数据分区策略选定迁移的数据后，选定的数据在节点间进行数据传输的效率会影响集群的性能。节点间数据传输的效率主要取决于数据传输策略是否充分利用了数据的存储格式以及数据传输的并行化。

采用增删节点会影响分区大小的分区策略时，如范围分区、一致性哈希等，数据迁移时需要对数据进行扫描来查找待迁移的数据。如在 Cassandra 中，数据迁移时首先需要执行全局 flush 操作，将所有 MemTable 的数据刷到磁盘中，然后对系统中所有 SSTable 的数据块进行扫描，将需要迁移的数据块流式传输到目标节点中。目标节点收到数据流后，将数据写入磁盘并重新构建索引、元数据等信息，最后加载到引擎中。

采用分区大小固定的分区策略时，如哈希槽分区，将数据和哈希槽进行绑定，存储引擎可以直接以槽为单位管理数据。通过将数据存储与哈希槽相结合，数据查找的代价为 0。比如在 Redis 系统中，节点中的数据按照槽为单位进行管理，数据迁移时将槽整体迁移，但是 Redis 采用的是流水线方式批量迁移数据，目标节点收到数据后还要重新写入存储引擎，不够高效。

2.4　时序数据库 Apache IoTDB

本节首先对 Apache IoTDB 的基本概念进行介绍，包括时间序列、存储组、TsFile 以及 Flush 和 Merge 操作。然后介绍应用架构和系统架构。

2.4.1　基本概念

2.4.1.1　时间序列

Apache IoTDB 是专门用于管理时间序列的时序数据库，每台设备的每个传感器都被视为一个时间序列，设备和传感器的名称构成时间序列名，使用一个由点

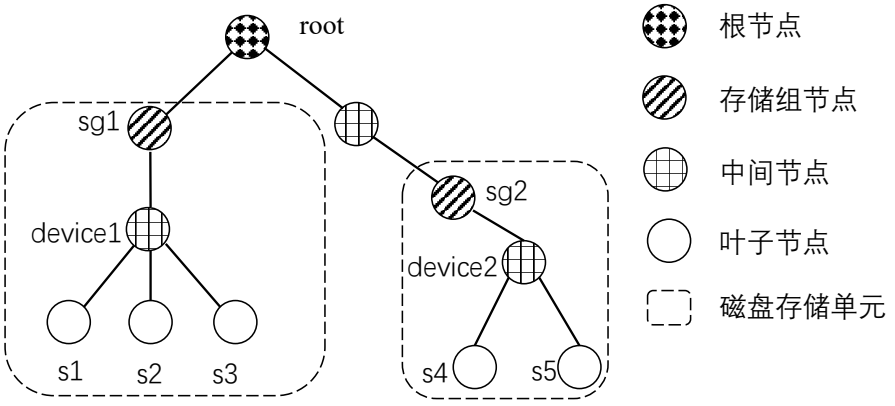


图 2.8　时间序列树

连接的路径来表示，如“root.sg1.device1.s1”。设备的传感器在不停地采集数据，每个数据点都会打上一个时间戳标记数据产生的时间，因此时间序列的一条数据采用 < 时间戳，值 > 的二元组来表示。

时间序列是按照层级来划分的，如时间序列“root.sg1.device1.s1”被划分为了四层，所有的时间序列构成了一棵时间序列树。如图 2.8，所有时间序列的根节点是保留的关键字“root”，从根节点出发达到叶子节点的任意路径代表一条时间序列。在进行数据写入前，需要首先创建相应的时间序列，如“create timeseries root.sg1.device1.s1 with datatype=BOOLEAN,encoding=PLAIN”表示创建数据类型为布尔值，编码方式为 PLAIN 的时间序列“root.sg1.device1.s1”。此外，Apache IoTDB 还支持自动创建 schema 的功能，在该功能开启时，系统对没有注册的时间序列会自动进行创建。

2.4.1.2　存储组

时间序列树由根节点、存储组节点、叶子节点和中间节点构成。叶子节点表示传感器，中间节点主要用于拓展名称使用，叶子节点的上一层表示设备。存储组节点是一类特殊的节点，每个存储组管理多个设备且有着独立的系统资源，这样可以提高系统的并行度并减少锁的竞争。在系统没有开启自动创建 schema 功能时，创建时间序列前需要创建相应的存储组，如“set storage group to root.sg1”表示创建存储组 root.sg1。

2.4.1.3　Flush 和 Merge

Apache IoTDB 的写入引擎基于 LSM-Tree[32] 结构，数据写入时首先存储在内存中的 MemTable 和磁盘上的写前日志中（write ahead log，简称 WAL）。当 MemTable 的大小超过阈值时，它会被刷到磁盘中的 TsFile 文件中，并且删除写前日志；当文

件达到一定大小时，文件会被封口，新的数据会写到新的 TsFile 文件中。用户可以通过调用 Flush 命令强制将当前正处于工作状态的 MemTable 刷到磁盘中，即强制数据落盘。

在时间序列的应用场景中，由于网络延迟等原因时序数据可能会乱序到达。Apache IoTDB 的写入引擎采用将顺序数据和乱序数据分开管理的方法，数据存储采用统一的格式 TsFile，后台定时调用 Merge 操作异步进行顺序数据和乱序数据的合并。用户可以通过调用 Merge 命令强制立即执行合并操作，并同步等待合并结果。

2.4.2　TsFile

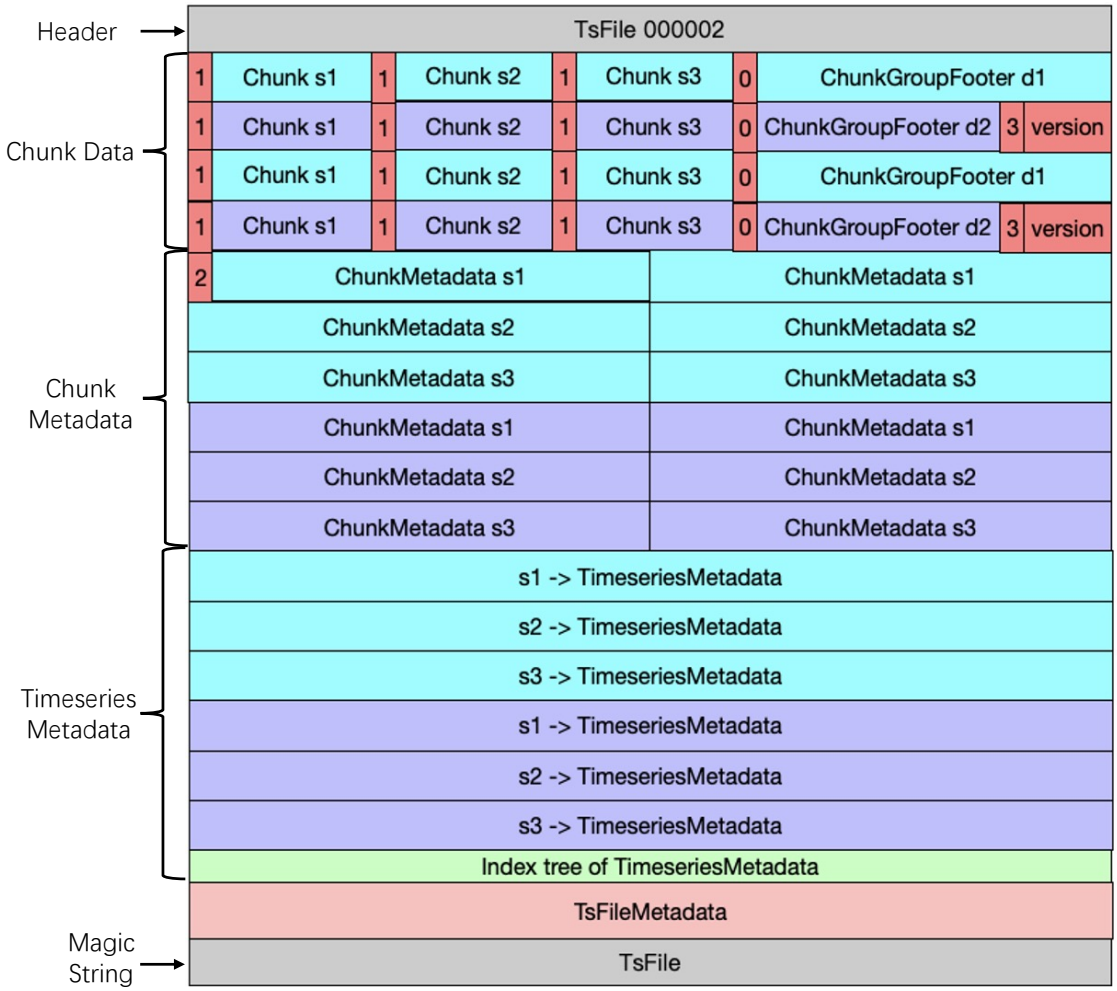


图 2.9　TsFile 文件格式

TsFile 是 Apache IoTDB 底层数据的存储文件，采用了一种专门为时序数据而设计的文件格式。如图 2.9，此文件包括两个设备 d1 和 d2，每个设备包含三个测点 s1、s2 和 s3，共有六个时间序列。TsFile 的文件头以六个字节的“Magic String”（TsFile）和版本号组成，文件尾部以“Magic String”（TsFile）作为结束，

中间包括数据和元数据。

TsFile 数据部分由多个 chunk group 组成，chunk group 存储的是一个设备的多个测点在一段时间内的数据。一个 chunk group 中包含多个 chunk 和一个 footer，每个 chunk 是按照时间递增的顺序存储某个测点在这段时间内的数据。

TsFile 元数据分为三部分：chunk metadata 记录了每个 chunk 在文件中的位置、数据类型和统计信息；timeseries metadata 记录了每个时间序列的所有 chunk metadata 在文件中的位置、数据类型和统计信息；TsFile matadata 由元数据索引树、版本映射信息和布隆过滤器等组成。

在 TsFile 文件中以查询时间序列 d2.s3 为例，查询流程如下：

1. 反序列化 TsFile metadata，得到 d2.s3 的 timeseries metadata 的位置。
2. 反序列化 timeseries metadata，得到 d2.s3 的所有 chunk 的 chunk metadata 的位置。
3. 根据 d2.s3 的每一个 chunk metadata，读取每个 chunk 的数据。

当 TsFile 文件达到一定阈值后会进行封口，封口后会生成一个 TsFile 的元信息文件，即.resource 文件。该文件中会记录 TsFile 文件内所有设备的起始时间和终止时间，用于在数据查询时进行文件快速过滤。

2.4.3　应用架构

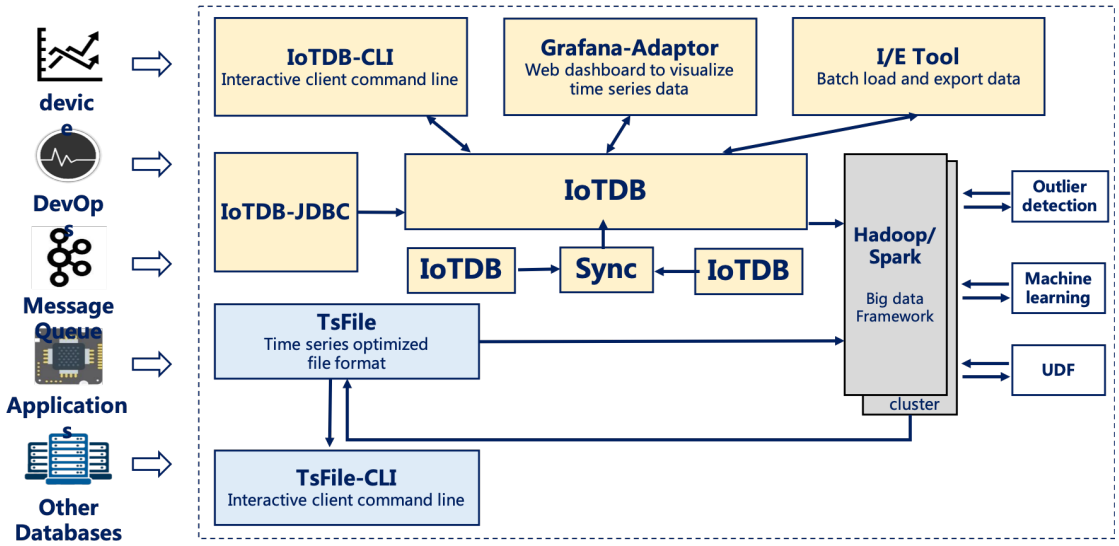


图 2.10　Apache IoTDB 应用架构

Apache IoTDB 的应用架构由若干组件构成，覆盖了数据的采集、存储、查询、可视化、分析等全生命周期的管理。如图 2.10 所示，主要包括以下组件：

1. IoTDB 引擎：IoTDB 引擎负责解析处理请求，包括元数据管理，读写数据和权限管理等。

2. IoTDB-JDBC 和 IoTDB-CLI 连接层：用户可以通过 JDBC 或者 CLI 连接到 IoTDB 引擎层进行元数据管理、请求读写数据等。
3. Grafana 连接层：Grafana 是开源的指标量监测和可视化工具，用户可以使用 Grafana 连接层对 IoTDB 的数据进行可视化展示，通过自定义可视化内容可以检测数据的读写速率等信息。
4. TsFile：IoTDB 的底层数据文件，是一种专门为存储时序数据而设计的列式文件格式，详细设计见 2.4.2 节。
5. I/E 工具：可以将其他类型的数据文件导入或者导出 IoTDB。
6. Sync 工具：可以在多个 IoTDB 实例之间进行高效的数据同步。
7. Hadoop/Spark 连接器：IoTDB 借助 TsFile 文件格式和 Hadoop/Spark 生态进行集成，支持将 TsFile 导入 Hadoop/Spark 中，同时用户也可以通过 Hadoop/Spark 操作 TsFile 文件的数据。

2.4.4　系统架构

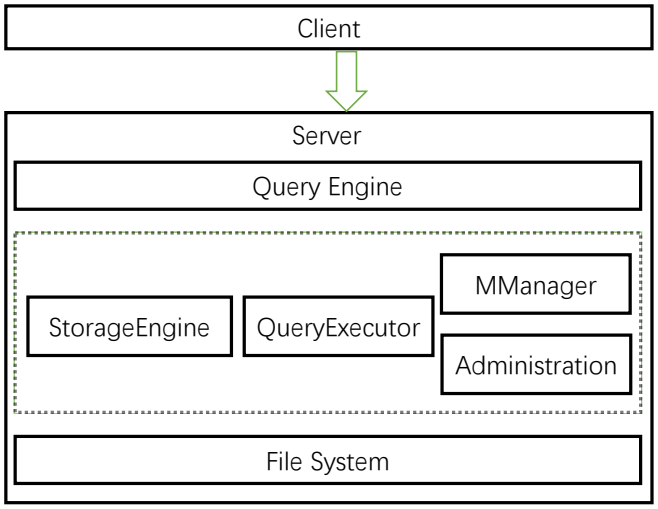


图 2.11　Apache IoTDB 系统架构

如图 2.11 所示，Apache IoTDB 系统主要包括以下几部分：

1. 查询引擎：负责对用户的各类请求进行处理和解析，并分发到其他组件进行进一步处理。
2. 存储引擎：对数据进行存储，包括写入、删除和修改。
3. 查询执行器：负责执行用户的各类查询请求。
4. 元数据管理模块：负责管理元数据，包括存储组和时间序列。
5. 权限模块：负责管理用户权限。
6. 文件系统：负责数据的落盘持久化。

2.5　本章小结

本章首先介绍了分布式系统的两种架构并进行了对比。其次描述了副本一致性，并详细介绍了 paxos 和 raft 这两个共识协议。然后介绍了集群扩展的相关内容，包括数据分区方法和数据迁移方案。最后对时序数据库 Apache IoTDB 的基本概念、应用架构和系统架构进行了介绍。

第 3 章　Apache IoTDB 现有分布式框架

本章介绍了当前 Apache IoTDB 的框架设计，包括双层粒度元数据管理方法和数据分区策略等。

3.1　分布式框架

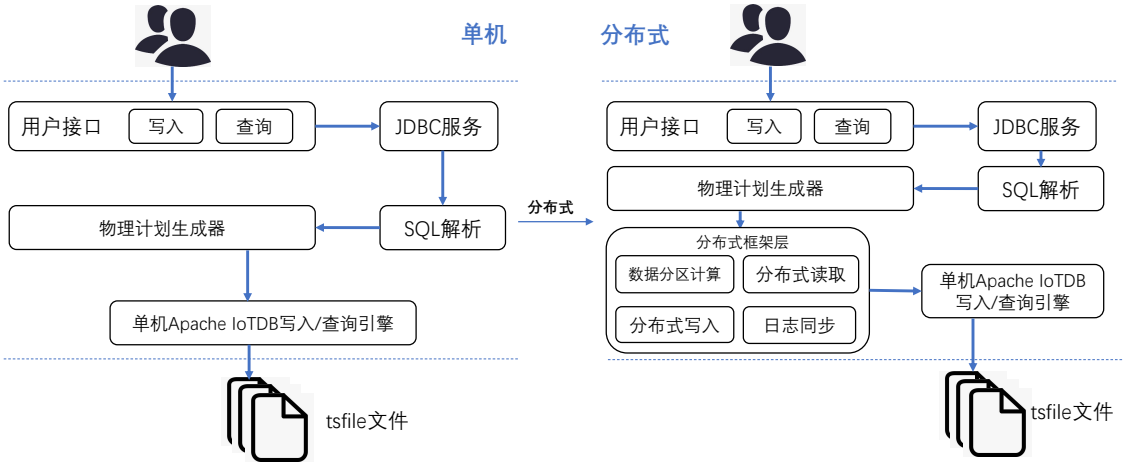


图 3.1　分布式框架

在进行 Apache IoTDB 分布式实现时，通过将单机版逻辑和分布式版逻辑进行解耦可以最大程度地降低开发复杂度。如图 3.1 所示，在单机版中，用户调用接口进行读写请求将依次通过 JDBC 服务、SQL 解析服务、物理计划生成服务，最后交由写入/查询引擎进行处理。JDBC 服务负责系统与用户间的交互，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架构层，达到了尽可能复用单机版逻辑的目的，并且与单机版中的功能模块基本解耦。此外，该框架还同时支持强一致性查询和最终一致性[33]查询。

3.1.1　双层元数据管理方法

元数据管理策略是 Apache IoTDB 的分布式框架设计中的要点。元数据在系统的读写流程中都有重要的作用：数据写入时需要元数据进行数据类型、权限等合法性的检查，查询数据时需要元数据进行查询路由。因此，分布式框架采用了双

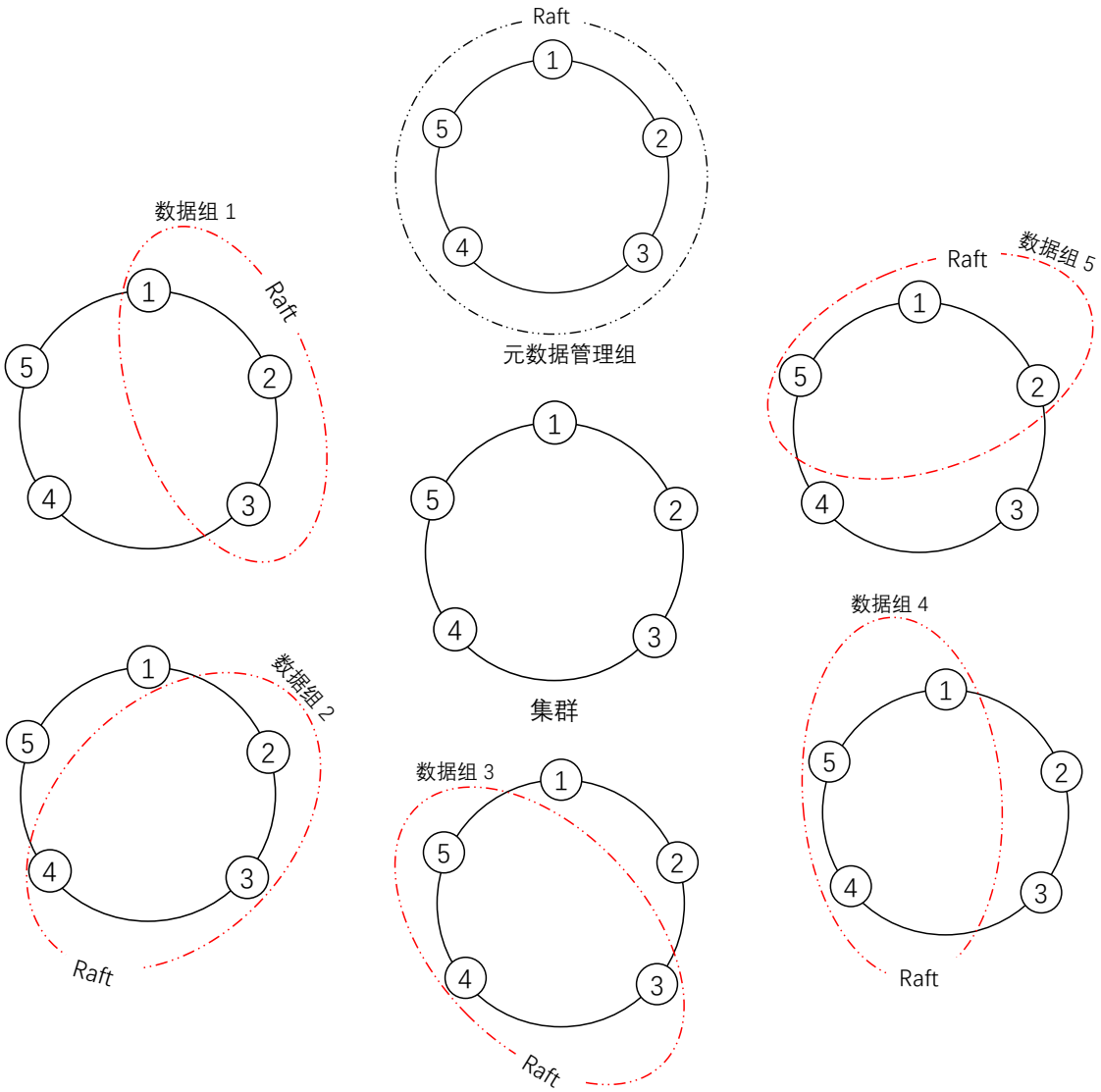


图 3.2　集群分组（5 节点 3 副本）

层粒度的元数据管理策略，其核心思想是将元数据拆分为两部分管理，分别是存储组元数据和时间序列元数据。其中时间序列元数据包括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对内存资源的消耗。

下面以 5 节点 3 副本为例进行详细描述，将 5 个集群节点按照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 3.2，该集群中包括一个元数据组和 5 个数据组，图中虚线框中代表一个分组。

双层元数据管理方法中包含元数据持有者和数据分区持有者，元数据组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副本一致性。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。

3.1.2　数据分区

Apache IoTDB 的分布式框架基于一致性哈希和环上查找算法对时序数据按**存储组加上时间片**进行分区。时间片指的是一段时间，具体的时间间隔可以由用户指定，可以设定为一小时、一天或者一周等。系统进行读写数据请求的路由时，首先根据一致性哈希算法计算 < 存储组，时间片 > 的哈希值，然后放置到哈希环上，顺时针寻找到第一个节点，该节点作为首节点的数据组负责管理该时间序列。节点在处理时间序列元数据的相关请求时，由于请求中不包含时间戳，因此创建/删除时间序列时是以存储组为粒度进行分区路由。

如果读写数据请求的路由也采用存储组作为分区粒度，当某个存储组成为热点数据时，高负载的压力将始终由管理该存储组的副本数台机器承担，不能很好地将热点数据的高负载分摊在集群中的各个节点中。因此读写数据请求的路由采用存储组加上时间片的方式，可以将同一个存储组不同时间片的负载分布在集群的不同节点上。

3.1.3　元数据组

集群中所有节点构成元数据组，创建和删除存储组的操作都交由元数据组进行管理。元数据组的作用是使得每个节点上都保留了必要的“索引”信息，由于数据分区的粒度为存储组加上时间片，因此在处理查询请求“select * from root”时，协调者节点可以在本地将这些查询拆分为多个针对不同存储组的不同时间片的子查询，然后根据一致性哈希计算哈希值就可以将请求转发给相应的数据组进行处理。

3.1.4　数据组

集群中每个节点作为起始点，在哈希环上顺时针找到副本数个节点即可组成一个数据组，因此 N 个节点的集群共有 N 个数据组。各个数据组之间的数据不重叠，每个数据组内使用 raft 协议来保证副本的一致性。数据组负责管理时间序列的元数据和时序数据，当进行数据路由时由于本地有相应时间序列的元数据，因此不需要和元数据组进行同步，只有当增删时间序列时需要和元数据组进行同步。

3.2　本章小结

本章介绍了当前 Apache IoTDB 的分布式框架，包括双层粒度元数据管理方法和数据分区策略等。当前实现不支持集群扩展功能，当集群初始化时每个节点通过读取本地的配置参数来获取集群的节点列表，在整个系统运行期间不能再增加或者删除节点。而在实际的分布式系统应用中，由于节点故障、业务需求变化等原因需要系统能在保证可用性的情况下进行集群的扩缩容，因此这个功能很有必要提供支持。

第 4 章　集群扩展机制设计

本章首先对集群扩展机制的设计难点进行分析，然后详细地介绍解决思路并引出集群扩展两阶段方法和数据迁移策略，最后给出处理流程。

4.1　设计难点

设计 Apache IoTDB 的分布式集群扩展机制时，要解决以下三个难点：

1. 如何保证多 raft 数据组成员变更的安全性

Raft 数据组成员变更的安全性指的是在集群增删节点的处理过渡期间组内不能存在多个 leader 的情况出现。Raft 协议本身提供了两种集群节点变更的处理方法，其中更简单可靠的方式是每次只允许用户向集群中增加或者删除一个节点，当需要增删多个节点时可通过一系列的单节点增删操作来实现。这种方式下新旧配置的多数节点至少包含一个重叠节点，同时由于 raft 节点对同一个任期只能投一票，因此只能有一个节点满足收到大部分节点投票的特性成为 leader，从而保证了成员变更的安全性。

但是 Apache IoTDB 中基于一致性哈希的数据组分配方法在集群扩展时会造成多个数据组的成员替换，使集群扩展变得更复杂。集群中包含多个数据组，每个数据组是由一致性哈希环中连续的副本数个节点构成。这种方式下在集群增删节点时不仅会存在数据组的增加或者删除，还会存在一些数据组进行了成员替换。

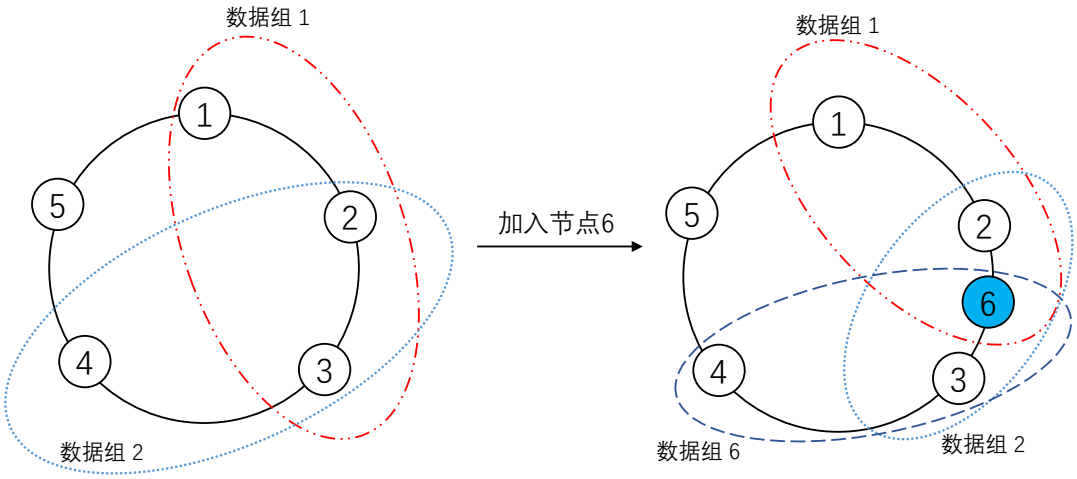


图 4.1　加入节点后的成员变更

如图 4.1，在 5 节点 3 副本的集群中加入节点 6 后除了增加一个数据组 6（数据组成员 {6，3，4}）外，数据组 1 和数据组 2 发生了成员替换。数据组 1 中新加

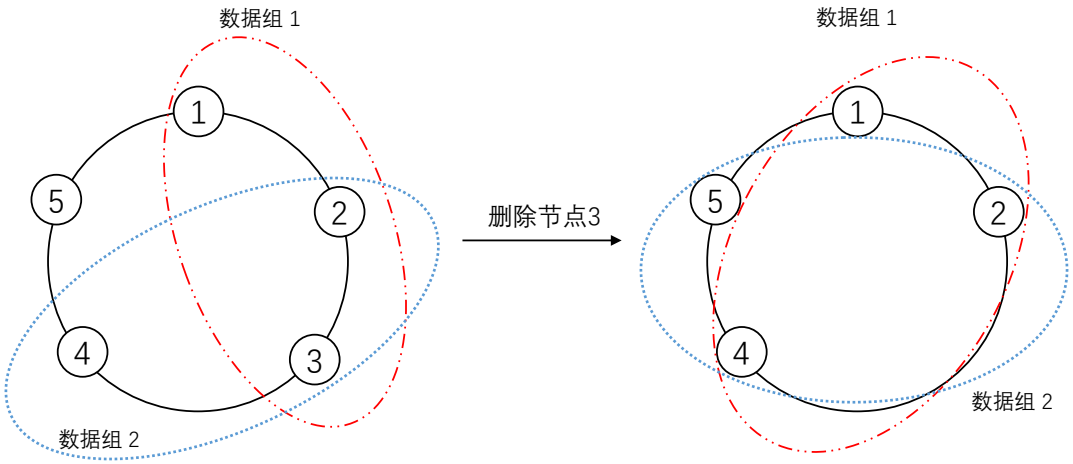


图 4.2　删除节点后的成员变更

入的节点 6 替换了节点 3，数据组成员由 {1，2，3} 变成了 {1，2，6}；数据组 2 中新加入的节点 6 替换了节点 4，数据组成员由 {2，3，4} 变成了 {2，6，3}。

如图 4.2，在 5 节点 3 副本的集群中移除节点 3 后除了删除了一个数据组 3（数据组成员 {3，4，5}）外，数据组 1 和数据组 2 发生了成员替换。数据组 1 中节点 4 替换了节点 3，数据组成员由 {1，2，3} 变成了 {1，2，4}；数据组 2 中节点 5 替换了节点 3，数据组成员由 {2，3，4} 变成了 {2，4，5}。

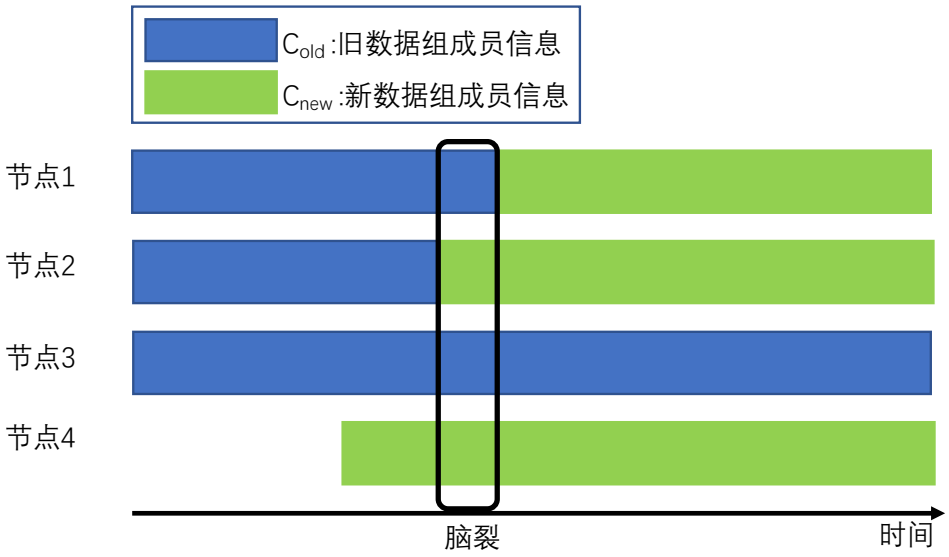


图 4.3　数据组成员替换

Raft 组的成员替换不可以直接执行，否则可能会产生脑裂。图 4.3 中直接用节点 4 替换节点 3，C_{old} 表示旧的数据组成员信息，用蓝色标识；C_{new} 表示新的数据组成员信息，用绿色标识。在图中标记处的位置，由于节点 1 和节点 3 采用旧数据组成员信息，节点 2 和节点 4 采用新数据组成员信息，均满足超过半数选票，因

此会产生脑裂。成员替换可以视为连续两次单节点成员变更操作，例如 raft 组成员由 {1，2，3} 变成了 {1，2，6}，可以通过先向 raft 组中加入节点 6 再从 raft 组中删除节点 3 实现，也可以通过先从 raft 组中删除节点 3 再向 raft 组中加入节点 6 实现。在 Apache IoTDB 的分布式框架下，由于存在一次集群成员变更但是有多个数据组需要两次成员变更来实现的情况，因此设计方案中需要保证这些数据组成员替换的安全性。

2. 如何保证数据的安全性

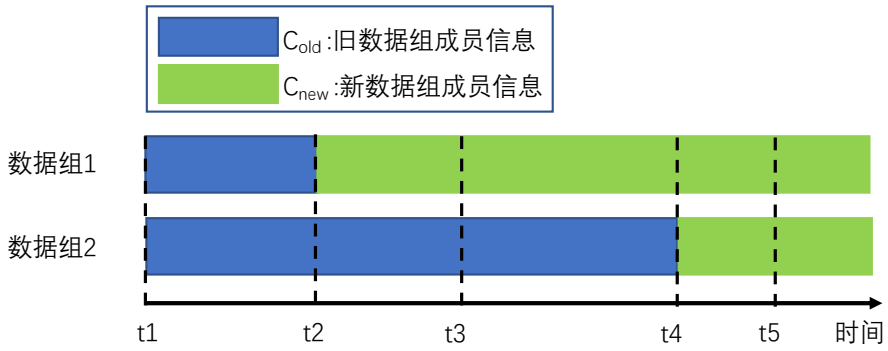


图 4.4　集群扩展期间可能出现的数据丢失问题

由于集群扩展期间不同节点更新本地的数据分区信息的时间是不固定的，数据迁移完成的时间也是不固定的，因此用户连接不同的节点进行读写请求时路由结果也可能是不同的。如果数据迁移期间还有数据写入旧分区管理组，那么这些数据会丢失，下面将举一个例子来说明。

图 4.4 中 t1 时刻开始进行集群扩展，t2 时刻数据组 1 开始应用新的分区信息，t4 时刻数据组 2 开始应用新的分区信息，t5 时刻所有数据组都应用了新的分区信息。对于某个特定的分区，数据组 1 是管理该分区的新数据组，数据组 2 是管理该分区的旧数据组，因此数据组 2 要将该分区的数据迁移至数据组 1，t3 时刻数据迁移结束。在 t3-t4 时间段内，如果客户端连接了一个节点作为协调者节点进行该分区的数据写入请求，这个协调者节点使用的是旧分区信息，这种情况下协调者节点会将写入请求转发给数据组 2 进行处理。数据组 2 收到请求后判断应当管理该分区，因此会直接执行写入请求。由于该分区的数据迁移已经在 t3 时刻完成，因此数据组 1 在 t3 时刻后会一直判定本地已经包括了该分区的所有数据，因此 t3 时刻后对该分区发起查询请求，数据组 1 会直接返回本地的查询结果，从而导致在 t3-t4 时间段内写入数据组 2 的数据丢失且无法被查询。因此设计方案中需要解决这类情况下数据丢失的问题。

3. 如何进行高效的数据迁移

集群扩展时，由于分区信息的变化，一些分区的数据会迁移到新的管理组。在

2.3.2 节中介绍过，Cassandra 系统首先通过全局 flush 将内存中的数据落盘，然后查找出所有需要进行数据迁移的数据块传输给目标节点，最后目标节点将数据块写入磁盘并重构索引、元数据等信息。在这种方法中，查找符合条件的数据块和重构元数据等过程不仅耗时还会增加对系统资源的消耗，因此如果设计的数据迁移方法可以减少或者避免这些消耗就能提高数据迁移的效率。

4.2　设计方案

由于 raft 协议将更复杂的成员变更操作转化为一系列的单增删节点操作来实现，因此本文设计的集群扩展机制同样采用逐节点扩展的实现方法。本节详细地介绍集群扩展机制的设计方案，来解决上一节提出的设计难点。

4.2.1　两阶段方法

4.2.1.1　方法内容

集群扩展时，有的数据组发生了成员替换，为了保证 raft 组成员替换的安全性不能直接执行替换的操作，需要拆分成两个步骤分别进行单节点增删变更，后文描述时统一采用先增后删的顺序。针对集群扩展期间可能发生的数据丢失问题，需要保证的是管理每个数据分区的新数据组在开始进行数据迁移后该分区的所有写入操作不会写入该分区的旧管理组，否则就会造成数据的丢失。要保证这一点，需要旧管理组也应用新的数据分区信息，这样收到数据写入请求时，旧管理组 leader 会将请求转发给新的数据组。这就要求所有分区的数据迁移要在集群中所有的数据组都采用了新的数据分区信息后再进行，因此集群扩展期间要有个明确的时间点来标识所有的数据组已应用新的数据分区信息。

基于以上思路设计了两阶段方法来保证安全性，具体内容如下：

1. 第一阶段

用户的增删节点操作转发给元数据组 leader 来处理，处理过程分为三步：

(1) 元数据组 leader 首先进行安全性检查，确保之前的集群扩展已经完成。检查满足要求后，元数据组 leader 对数据进行重分区并将结果加入增删节点的日志中，然后将这条日志分发给所有元数据组的 follower，等到超过半数的 follower 收到这条日志执行第二步。

(2) 元数据组 leader 将这条日志发送给所有的 raft 数据组。每个数据组收到日志后按照 raft 流程进行处理，不同点是所有数据组成员在收到集群增删节点的日志时直接提交这条日志（包括 leader），即更新本地的集群成员信息和数据分区信息并进行**预成员变更**（预成员变更是对发生成员替换的数据组执行第

一步增加节点的变更操作）。每个数据组 leader 按照预成员变更后的组内成员信息进行日志分发。

(3) 当所有的数据组完成增删节点请求后，第一阶段结束并返回给用户集群增加/删除节点成功的消息。若集群扩展操作为增加节点，新加入的节点会正式启动并提供服务。

2. 第二阶段

元数据组提交增删节点日志，处理过程分为四步：

(1) 对本地所有数据组进行**正式成员变更**（正式成员变更是对发生成员替换的数据组执行第二步删除节点的变更操作）。

(2) 当由于成员替换导致本节点成为某个数据组的新成员时，本地创建并启动该数据组的实例提供服务；当由于成员替换导致某个数据组成员不包含本节点时，本地关闭并删除该数据组的实例。

(3) 对所有的数据组按照数据重分区情况启动异步的数据迁移任务。当所有数据组的数据迁移完成后，用户就可以进行下一个集群扩展请求。

(4) 如果是删除节点操作，元数据组 leader 需要通知被删除的节点执行删除流程。因为第二阶段时元数据组的成员已经不包含被删除的节点，不会和这个节点进行心跳，因此需要主动通知它执行删除流程。

图 4.5　连续增加节点时组内成员日志长度

由于设计的集群扩展机制采用逐节点扩展的方式，因此多节点的集群扩展将转化为一系列单节点集群扩展操作来实现，而且每次集群扩展开始前需要检查上次集群扩展是否完成，如果没有则拒绝此次集群扩展请求。检查上次集群扩展是否完成分为两部分：一方面要检查上次集群扩展造成的数据迁移是否完成；另一方面要检查需要进行成员变更的数据组是否存在可能阻塞日志提交的节点，该节

点还未追上 leader 的日志。如图 4.5，数据组一开始的成员为 {1，2，3}。假设第一次集群扩展时加入了节点 4，并将替换组内成员 3。这种情况下预成员变更时组内成员为 {1，2，3，4}，由于 1、2、3 的日志是和 leader 同步的，因此满足了超过半数节点的要求，正式成员变更后数据组成员为 {1，2，4}。假设在节点 4 没有追上 leader 日志前第二次集群扩展加入了节点 5，并将替换组内成员 2。这种情况下预成员变更时组内成员为 {1，2，4，5}，由于节点 4 和节点 5 都没有追上 leader 的日志，因此一直不能满足超过半数节点的要求直至有一个节点追赶上了 leader 的日志。在这期间数据组会一直阻塞后续的请求，导致不可用，因此为了避免这种情况发生，在进行集群扩展前要进行检查。

在第一阶段中数据组收到集群增删节点的日志时直接提交日志。只有当 raft 组内超过半数节点已经提交了上一次成员变更，该 raft 组才可以进行下一次成员变更操作。如果对于增删节点的日志也像其他类型的日志一样，即数据组成员只有在 leader 提交了这条日志后才会提交日志，那么数据组 leader 将很难知道什么时候数据组中超过半数节点已经采用了新的集群成员信息和数据分区信息，这种情况下就需要添加一些其他的机制来保证；反之，如果在收到增删节点的日志后直接提交日志，那么数据组 leader 返回执行结果时就可以安全地进行下一次成员变更操作。此外，在集群扩展前检查上次集群扩展是否完成的步骤也保证了下一次预成员变更之前上一次的正式成员变更已完成，因此该方法**保证了 raft 数据组成员变更的安全性**。

在两阶段方法中，仅当第一阶段结束时集群中所有的数据组都已经采用了新的数据分区信息后，第二阶段中第三步所有数据组才会启动数据迁移，因此该方法严格限制了变更数据分区信息和数据迁移的执行顺序，确保了第一阶段结束后所有数据写入请求都只会由新的数据组来处理，不会写入旧的数据组，从而解决了数据可能丢失的问题，**保证了数据的安全性**。当用户在第二阶段查询某个分区的数据时，只要将管理该分区的旧数据组的查询结果和新数据组的查询结果合并，就可以返回正确且完整的查询请求结果。

4.2.1.2　接口设计

下面介绍两阶段方法中主要的一些操作接口：

boolean checkPreviousChangeMembership()：

此方法是元数据组 leader 在开始处理集群增删节点请求前的安全性检查工作，一次集群扩展完全结束后才可以安全地进行下一次集群扩展操作，在第一阶段的第一步中调用。

void preAddNode(Node *node*)：

该方法用于在集群增加节点时对数据组进行预成员变更，输入参数 node 是加入集群的节点，通过新节点计算得到的哈希值在哈希环上的位置判断数据组是否需要进行成员替换，如果需要则将节点插入合适的位置。

void preRemoveNode(Node *node*)：

该方法用于在集群删除节点时对数据组进行预成员变更，输入参数 node 是待删除的节点，如果数据组成员中包含这个节点，那么将哈希环上最后一个成员节点后面的那个节点加入数据组中。

void addNode(Node *node*)：

该方法用于在集群增加节点时对数据组进行正式成员变更，输入参数 node 是加入集群的节点，如果数据组中已经包含了新节点，那么将数据组中最后一个成员节点移除。

void removeNode(Node *node*)：

该方法用于在集群删除节点时对数据组进行正式成员变更，输入参数 node 是待删除的节点，如果数据组成员中包含该节点，那么从数据组中将该成员节点移除。

4.2.2　数据重分区

4.2.2.1　方法内容

在两阶段方法中，集群成员变更的请求由元数据组 leader 来处理，因此可以由元数据 leader 先进行数据分区的修改，然后将修改后的数据分区信息（即分区表）放入增删节点日志中同步给其他节点，这些节点提交这条日志时加载新的数据分区信息，这样可以实现数据重分区的统一管理。

分区表需要版本控制。当集群已经进行过多次集群扩展时，raft 组的日志列表中会存在多个集群扩展日志，每个集群扩展日志中都会包含一个分区表。当新加入 raft 组的节点和 leader 进行日志同步时会依次执行这些日志并加载历史分区表，这时通过版本可以在加载分区表时判断是否为历史分区表，如果是则直接跳过加载阶段。

数据的重分区可以有多种策略来执行，最基础的策略是将哈希槽的数量均分到所有的数据组中，这种方式适合数据比较均匀的情况，每个哈希槽的数据量都

大致相同；但当出现了热点数据时，有些哈希槽的数据量会远多于其他槽的数据，此时的数据重分区策略更适合按照各个节点的实际负载指标来进行分配，需要搭配负载均衡器使用，比如可以按照数据量、写入速率等，使得选定的负载能尽量均分到各个数据组中。

4.2.2.2　接口设计

数据重分区阶段主要使用以下三个接口：

PartitionTable moveSlotsToNew(Node *node*)：

该方法在集群增加节点时用于数据重分区，输入参数 node 是加入集群的节点，数据组管理数据是以哈希槽为单位进行的，新加入的节点会以它为首节点形成一个新的数据组，这个数据组会从其他数据组中获取哈希槽来接管。返回结果是修改后的数据分区信息表。

PartitionTable retrieveSlots(Node *node*)：

该方法在集群删除节点时用于数据重分区，输入参数 node 是待删除的节点。以待删除的节点为首节点的数据组会从集群中删除，这个数据组管理的数据会分摊到其他数据组中。返回结果是修改后的数据分区信息表。

void loadPartitionTable(PartitionTable *table*)：

节点调用该方法加载分区表，输入参数是新的数据分区信息表。

4.2.3　数据迁移

4.2.3.1　分区方法改进

在 2.3.1.3 节中介绍过，加入虚拟节点后的一致性哈希方法不适用于集群节点较少的情况，且不利于集群扩展时数据的高效迁移，因此改进后采用哈希槽分区方式。将一致性哈希环划分为一定数量相等大小的槽（默认为 10000），然后将这些哈希槽分配给数据组。

为了适配哈希槽分区方法，在 Apache IoTDB 的存储引擎中，每个存储组也按照时间片分开进行数据的管理。这样存储组的每个时间片内所有 TsFile 文件的数据都属于同一个槽，每个槽管理多个存储组的多个时间片数据，因此在进行数据迁移时可以以时间片为单位进行整体迁移。

4.2.3.2　方法内容

在两阶段方法中保证了在开始数据迁移后数据不会写入旧数据组，因此旧数据持有者可以执行 flush 操作强制将内存 MemTable 中的数据落盘。由于每个 TsFile 文件中的数据都属于同一个哈希槽，因此可以直接以 **TsFile 文件级别**进行数据传输和加载，避免了数据查找和重建元数据的过程，**降低了系统资源的消耗**。

数据迁移的内容分为两部分，一部分是时序数据，即一系列 TsFile 数据文件；另一部分是时间序列的元数据。旧数据组的集群扩展日志不需要迁移，因为历史的集群成员变更不会影响当前系统的运行状态。旧数据组节点和新数据组节点之间的数据迁移流程如下：

1. 每个新数据组节点选择一个旧数据组节点，并向旧数据组节点请求接管的哈希槽数据。
2. 旧数据组节点对所有需要迁移的存储组时间片数据执行 flush 操作，然后收集需要迁移的 TsFile 文件列表和属于该分区的时间序列元信息，最后向新数据组节点返回结果。
3. 收到结果后，新数据组节点加载时间序列元信息，并将文件列表中的所有文件从旧数据组节点拉取到本地并加载。所有文件加载完毕后，通知旧数据组的所有节点一个副本的数据迁移完成。
4. 当旧数据组的各个节点收到副本数个通知后，说明新数据组的所有副本都完成了数据迁移，此时可以将本地已完成迁移的数据进行删除，数据迁移结束。

整个数据迁移流程中有以下几点细节：

1. 旧数据组节点在收集需要迁移的时间序列元信息时，如果时间序列数量超过一定阈值，那么将这些时间序列元信息写入本地文件中，并将文件名返回给新数据组节点；否则，直接将时间序列元信息列表返回给新数据组节点。新数据组节点在处理时间序列元信息时，如果收到的是时间序列元信息列表，那么直接执行加载过程；否则，先从旧数据组节点将时间序列元信息文件拉取到本地，然后再依次加载文件中的时间序列元信息。
2. 和 Cassandra 的旧分区持有者在数据迁移开始时需要进行全局 flush 操作不同，由于系统中每个 TsFile 文件中的数据都属于同一个哈希槽，因此旧数据组节点在进行 flush 操作时仅 flush 需要迁移的存储组时间片的 MemTable 即可，减少了不必要的 flush 操作。
3. 由于集群中每个数据组都是由 n 个节点组成的 n 副本 raft 组，因此新数据组的 n 个节点都要进行数据迁移，如果全都和旧数据组的 leader 执行数据迁移流程，那么旧数据组 leader 的资源消耗会很大。为了均摊数据迁移的负载，新

数据组节点和旧数据组节点按照一对一的方式进行数据迁移，可以最大化数据迁移效率。因此在第一步选择旧数据组节点时，新数据组节点按照自己在组内的位置去找旧数据组中相同位置的节点，如新数据组成员为 {1，2，3} 且旧数据组成员为 {4，5，6}，那么节点 1 选择节点 4、节点 2 选择节点 5、节点 3 选择节点 6 进行数据迁移。这种方法下旧数据组 follower 需要和 leader 同步保证数据是最新的。

4. 在进行数据迁移时，新数据组的节点有可能同时也是旧数据组的成员，以新数据组成员是 {1，2，3} 且旧数据组成员是 {3，4，5} 为例，由于节点 3 本地已有相应的数据，这种情况下不需要进行数据迁移流程，直接通知旧数据组的所有节点一个副本的数据迁移完成；此外，执行第四步骤时节点 3 作为旧数据组成员由于同时也是新数据组成员，因此本地不能将相应的数据删除。

4.2.3.3　接口设计

数据迁移阶段主要用到以下四个接口：

PullSnapshotResp requestSnapshot(List<Integer> *requiredSlots*)：

该方法用于新数据组节点向旧数据组节点请求待迁移的数据，输入参数为需要迁移的哈希槽列表，返回结果包括 TsFile 文件列表和时间序列元信息列表。

void installTimeseriesSchema(Collection<TimeseriesSchema> *schemas*)：

该方法用于新数据组节点注册返回结果中的时间序列元信息列表 schemas。

void pullRemoteTsFile(String *path*, Node *node*)：

对于返回结果中的 TsFile 文件列表，调用该方法从目标节点 node 拉取数据文件，path 是该 TsFile 的文件路径。

void loadFile(String *path*)：

每个从旧数据组节点拉取到的 TsFile 数据文件，调用该方法加载到存储引擎中提供服务。

4.3　处理流程

4.3.1　加入节点

集群扩增节点时，启动待加入集群的节点前，需要先在配置项中指定种子节点列表，然后执行加入节点的脚本来启动新节点，如图 4.6 所示，加入节点的流程

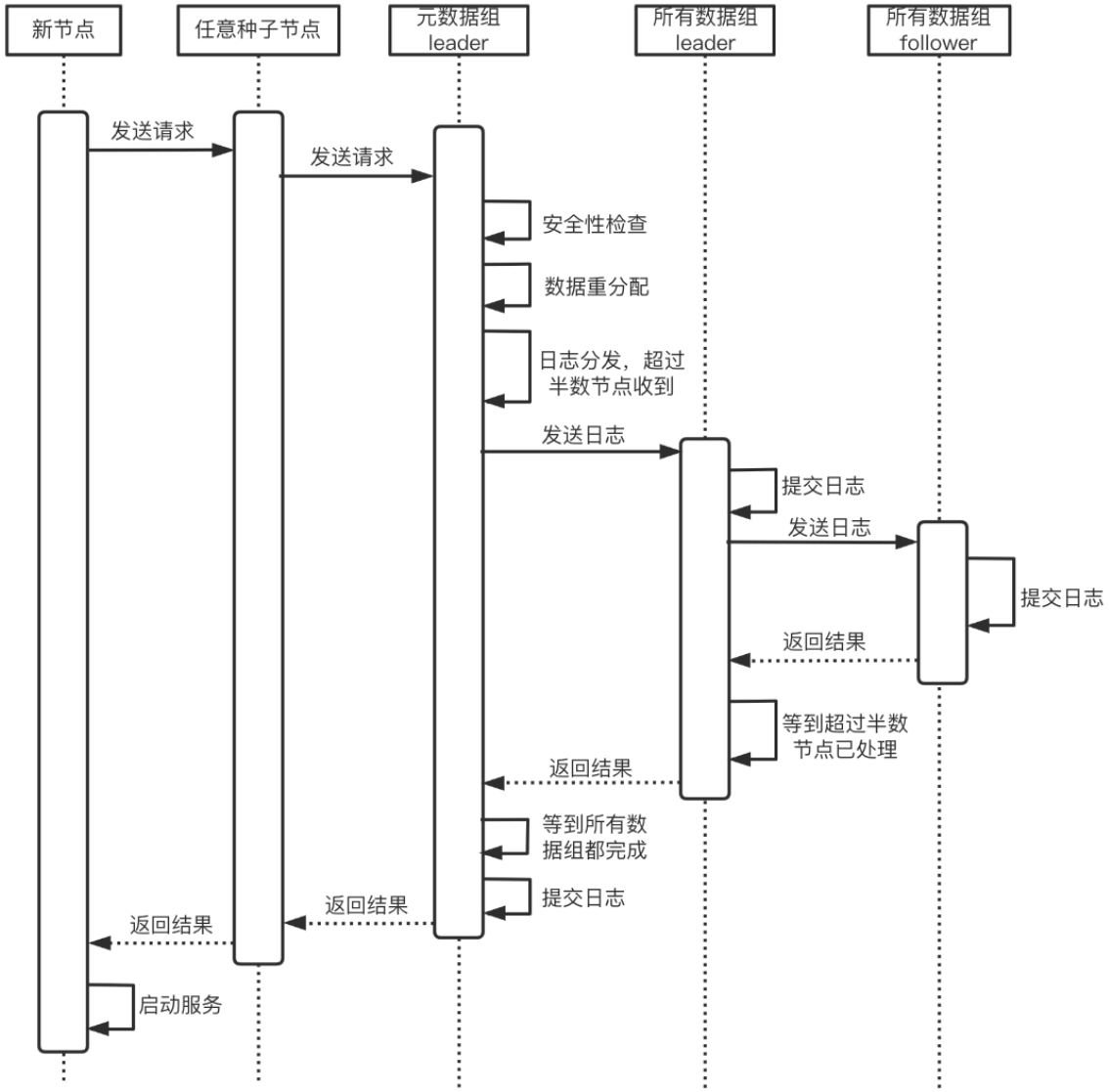


图 4.6　加入节点时序图

如下：

1. 新节点从种子节点列表中选择一个节点，并向该节点发送加入集群的请求。
2. 被选定的种子节点接到请求后，将请求转发给元数据组 leader。
3. 元数据组 leader 收到请求后，首先进行安全性检查，包括用户配置项如副本数、存储引擎分区间隔等是否一致、之前的集群扩展是否已经完成。然后进行数据重分区，并将修改后的数据分区信息加入日志中。最后将日志分发给元数据组的成员。
4. 超过半数的元数据成员收到后，元数据组 leader 将日志发送给所有的数据组 leader。
5. 每个数据组 leader 收到日志后直接提交日志，包括更新分区表、修改集群成员信息和组内预成员变更。然后将日志发送给数据组的所有成员，成员收到

日志后同样直接提交并返回结果。当超过半数数据组节点处理完日志后返回结果。

6. 等到所有数据组都完成后，第一阶段结束，元数据组 leader 提交日志并返回结果。
7. 新节点收到返回结果后，创建并启动元数据组和数据组，开始提供服务。

4.3.2　删除节点

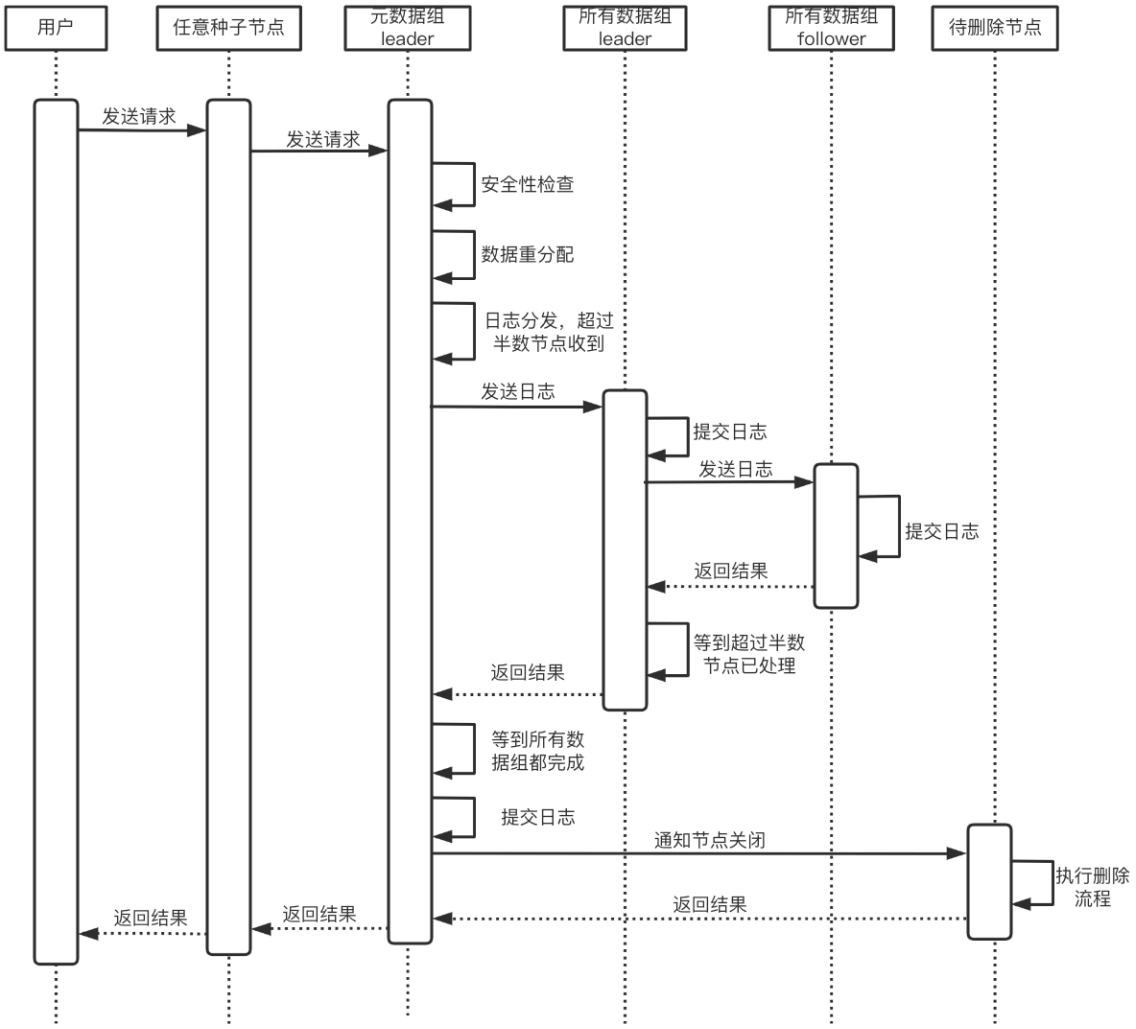


图 4.7　删除节点时序图

集群移除节点时，用户可以在集群中任意一个节点中启动删除节点脚本并指定要删除的节点，如图 4.7 所示，删除节点的流程如下：

1. 删除节点脚本启动后，首先从种子节点列表中选择一个节点，然后向该节点发送从集群中删除目标节点的请求。
2. 被选定的种子节点接到请求后，将请求转发给元数据组 leader。
3. 元数据组 leader 收到请求后，首先进行安全性检查，包括待删除节点是否在

集群中、之前的集群扩展是否已经完成。然后进行数据重分区，并将修改后的数据分区信息加入日志中。最后将日志分发给元数据组的成员。

4. 超过半数的元数据成员收到日志后，元数据组 leader 再将日志发送给所有的数据组 leader。
5. 每个数据组 leader 收到日志后直接提交日志，包括更新分区表和修改集群成员信息。然后将日志发送给数据组的成员，成员收到日志后同样直接提交并返回结果。当超过半数数据组节点处理完日志后返回结果。
6. 等到所有数据组都完成后，第一阶段结束，元数据组 leader 提交日志并通知待删除节点关闭。
7. 待删除节点收到通知后，执行删除流程，关闭元数据组和所有的数据组并停止服务，返回结果给用户。

4.3.3　数据写入

加入集群扩展功能后，由于集群增删节点会导致写入数据过程中协调者节点的本地集群分区信息可能不是最新的，从而将请求转发给了管理该分区的旧数据组。在这种情况下需要让协调者节点重定向至管理该分区的新数据组，因此在每个数据组 leader 处理写入请求时要先判断自己是不是管理该数据所对应分区的数据组。如图 4.8 所示，写入流程如下：

1. 用户选择集群中的任何一个节点作为协调者节点，建立连接并发送数据写入请求。
2. 协调者节点先解析写入操作中的时间序列所属存储组，并计算时间戳对应的时间片，然后根据一致性哈希算法计算出 < 存储组，时间戳 > 所对应的哈希槽，最后确定管理该哈希槽的数据组为数据组 1。
3. 协调者节点将写入请求转化的物理计划发送给数据组 1 的 leader。
4. 数据组 1 的 leader 收到请求后，根据本地的数据分区信息检查自己是否管理该数据所对应的分区。如果不是，则返回真正管理该分区的数据组，即数据组 2，协调者节点将请求重新发送给数据组 2 的 leader。
5. 管理该数据的数据组分发日志并等到超过半数数据组成员收到后，数据组 leader 提交日志，最后将结果返回给用户。

4.3.4　数据查询

加入集群扩展功能后，在各个数据组中查询某个分区的数据时，将分为以下两种情况：

1. 该分区数据在该数据组中处于**稳态**。稳态指的是该分区的数据不处于数据迁

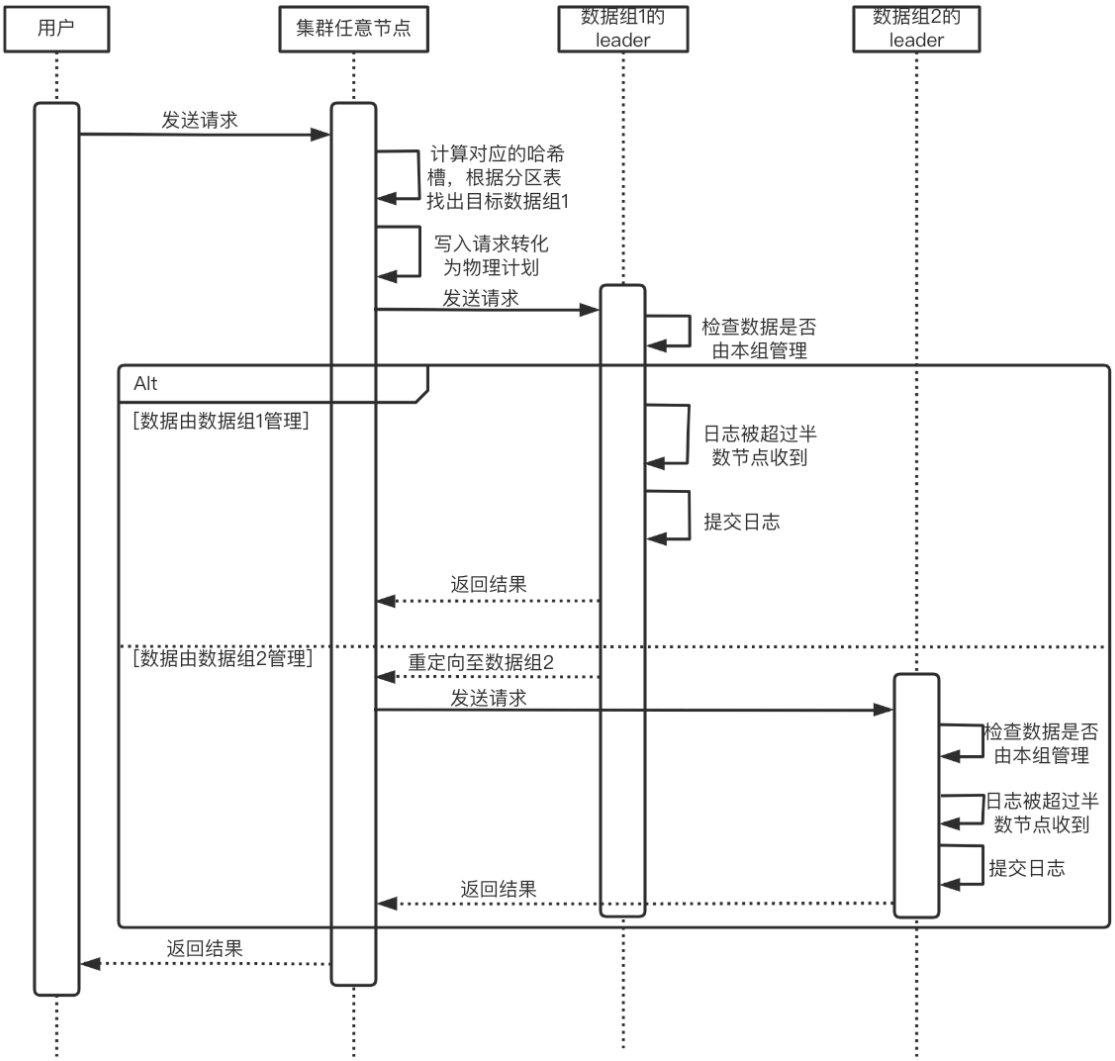


图 4.8　数据写入时序图

移的流程中，本地含有该分区的所有数据，在这种情况下查询走常规的 raft 流程。

2. 该分区数据在该数据组中处于**过渡态**。过渡态指的是由于数据分区信息的变化导致的数据迁移正在进行中，在这种情况下本地仅包含部分该分区的数据，旧的数据组中包含该分区的其他数据，因此查询数据时需要同时查询管理该分区的旧数据组和新数据组，并将结果进行合并后返回。

由于管理分区的新数据组在数据迁移完成前分区数据是不完整的，因此查询结果有可能需要合并管理该分区的旧数据组数据。如图 4.9 所示，查询流程如下：

1. 用户选择集群中的任何一个节点作为协调者节点，建立连接并发送数据查询请求。
2. 协调者节点解析查询请求，并拆分为多个数据分区的子查询，形成对多个数据组的子查询请求。图中示例拆分为对数据组 1 和数据组 2 的两个子查询。

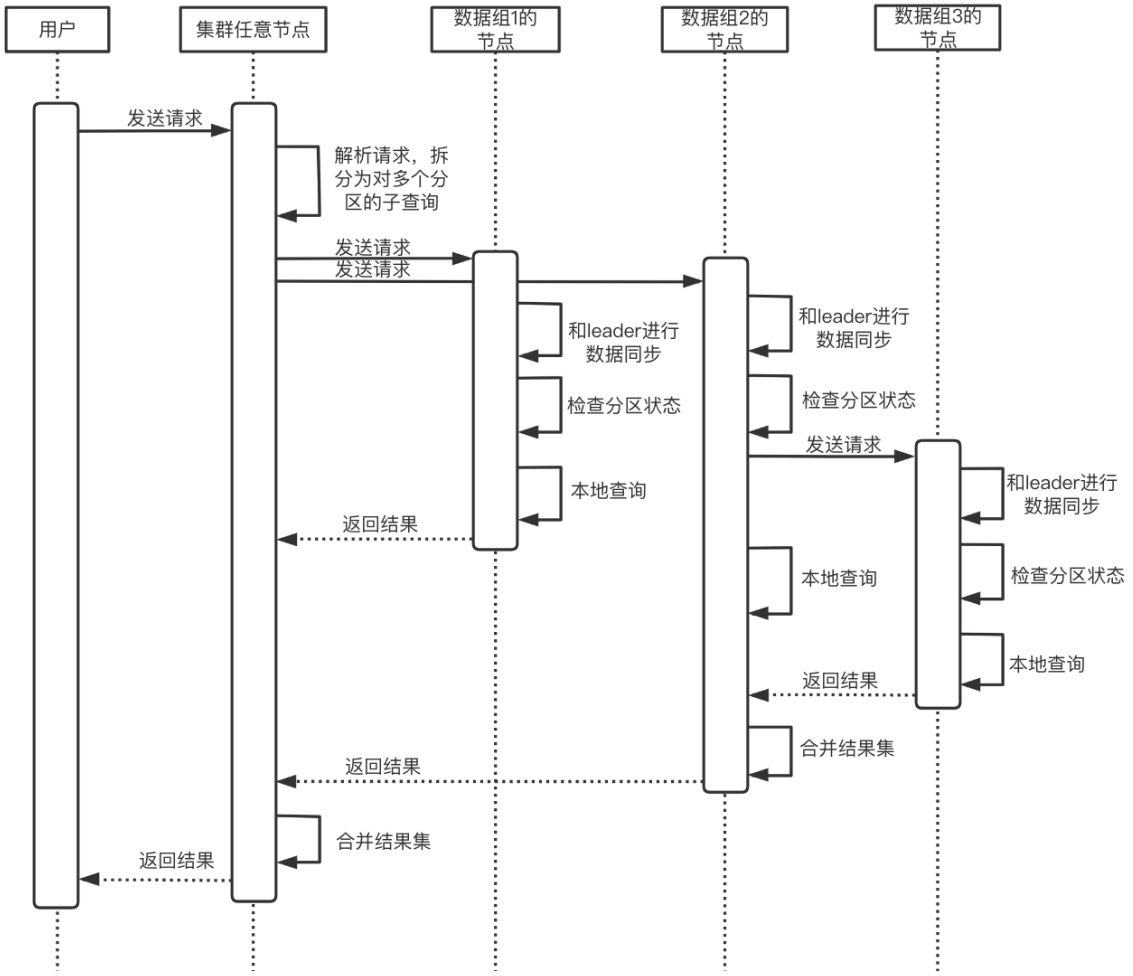


图 4.9　数据查询时序图

3. 协调者节点在各个数据组中任选一个节点并发送查询请求。
4. 各个数据组节点和 leader 同步日志保证本地数据的完整性。这里需要注意当集群扩展时产生成员替换的数据组会有新节点的加入，这个新节点的日志和 leader 相差很大。这种情况下新节点收到查询请求时直接拒绝本节点查询，协调者会将查询请求重新发送给数据组中其他日志和 leader 已同步的节点。
5. 各个数据组节点根据分区的状态进行查询请求处理，数据组 1 节点待查询分区处于稳态，直接本地查询；数据组 2 节点的待查询分区由于处于过渡态，因此需要和管理该分区的旧数据组（数据组 3）节点的查询结果合并。最后各个数据组节点将查询结果返回给协调者节点。
6. 协调者节点将各个数据组返回的查询结果进行合并，最后将结果返回给客户端。

4.4　本章小结

本章首先详细分析了设计集群扩展机制时的难点，然后详细介绍了解决方案和接口设计，采用元数据组 leader 统一负责数据重分区信息并同步到各个节点的方式来实现数据分区的分配；采用集群扩展两阶段方法既保障了多 raft 组成员变更的安全性，还解决了数据可能丢失的问题；采用基于哈希槽和时间片的文件级别的数据迁移方法在数据组间高效传输数据。最后介绍集群扩展机制下的处理流程，包括增删节点和读写数据。

第 5 章　集群扩展机制实现

在上一章设计中将集群扩展机制分为两阶段方法、数据重分区和数据迁移三部分。本章将介绍这三部分的具体实现，给出关键实现的流程图，并对异常情况下的处理进行描述。

5.1　两阶段方法

5.1.1　功能实现

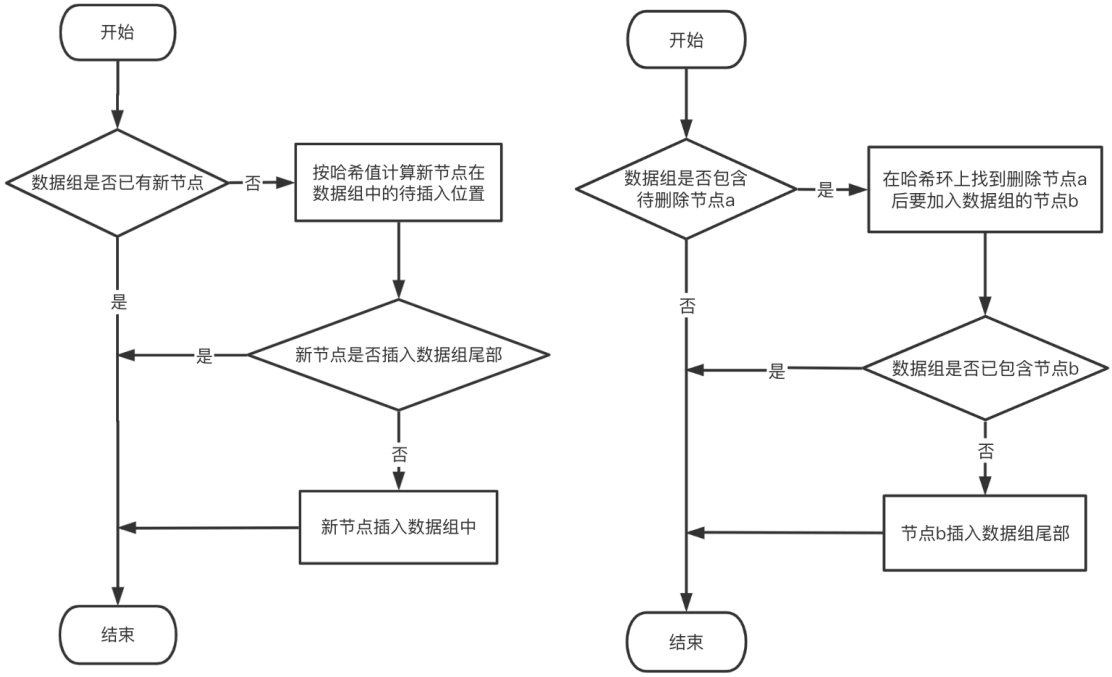
两阶段方法中最主要的操作步骤是每个数据组的预成员变更和正式成员变更。在实现这两个功能时，需要考虑并解决预成员变更和正式成员变更操作的顺序问题：在系统运行过程中集群大部分节点会先进行预成员变更再进行正式成员变更，但是存在一些数据组节点先进行了正式成员变更再进行预成员变更。这是因为正式成员变更是在第二阶段元数据组提交增删节点日志时触发的，第一阶段结束时保证了所有数据组超过半数节点进行了预成员变更，而那些没有进行预成员变更的数据组可能会出现先进行正式成员变更的情况。

解决上述问题最简单直观的方法是数据组在正式成员变更前先检查有没有执行预成员变更，如果没有则一直等待。这种方法的弊端是可能导致某些节点处于长时间等待状态。由于预成员变更是在数据组提交日志时执行的，正式成员变更是在元数据组提交日志时执行的，这种方法强制串行化了两个 raft 组的日志提交，等待的过程实际上阻塞了一些元数据组成员的日志提交过程。

为了解决上述弊端，在实现时直接将预成员变更设计成幂等操作，这样在进行正式成员变更操作时不管数据组有没有进行预成员变更操作，都再执行一遍预成员变更操作，并且保证在正式成员变更操作后再进行预成员变更操作的正确性。

图 5.1 描述了预成员变更的流程：

- 增加节点时，首先判断数据组是否已经包括了新节点。若数据组包括新节点，则说明数据组已经执行过预成员变更，这种情况下应直接返回；若数据组不包括新节点，则根据哈希值找到新节点在哈希环上的位置。如果新节点位于数据组最后一个节点沿着哈希环顺时针到数据组首节点的区间，说明新节点的加入不会引起本数据组的成员变更，否则将节点插入数据组中并返回。
- 删除节点时，首先判断数据组是否包含待删除节点。如果数据组不包含待删除节点，则会有两种情况：一是这个节点的删除本身不会引起本数据组的成



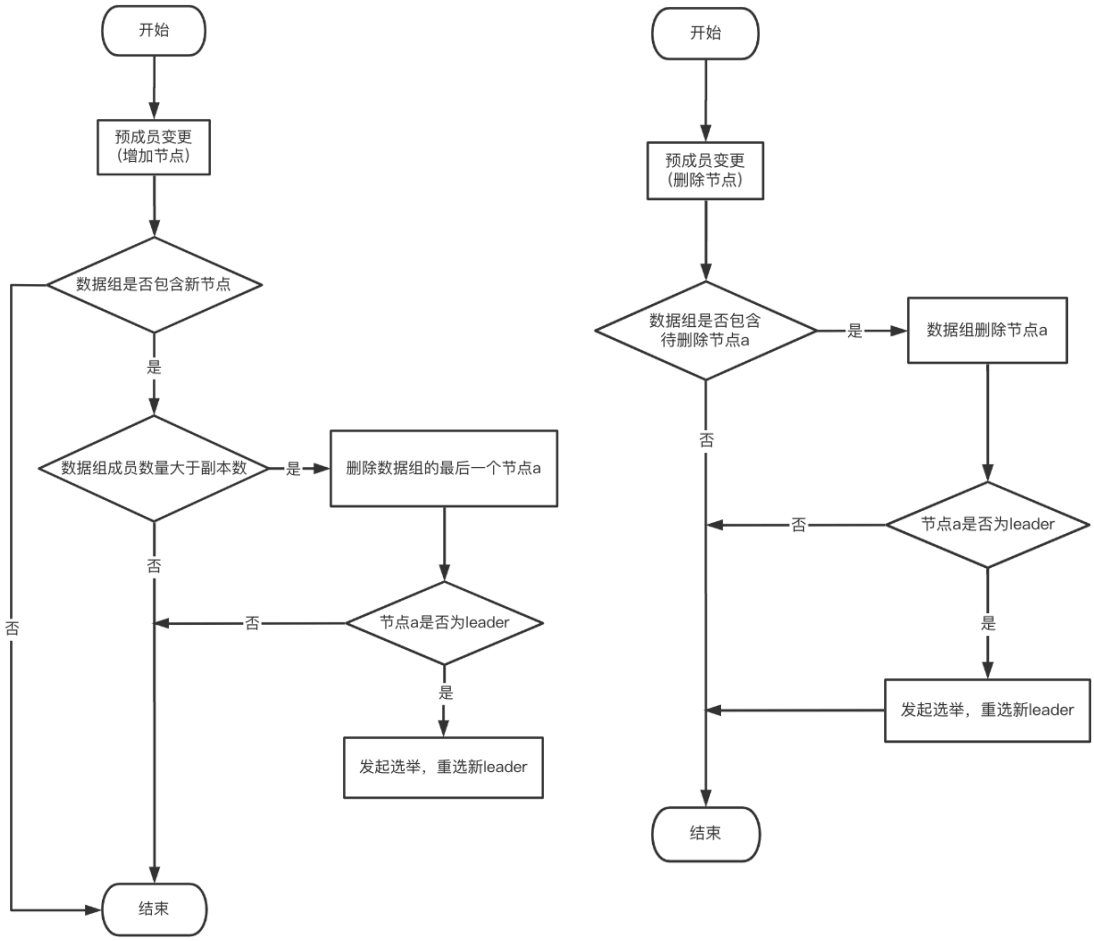
(a) 增加节点　　(b) 删除节点

图 5.1　数据组预成员变更流程图

员变更，二是数据组已经执行过正式成员变更并已经将这个节点删除，这种情况下应直接返回。然后要判断数据组是否已经包括了替换节点，如果包括，则说明数据组之前已经执行过预成员变更，直接返回；如果不包括，就将替换的节点加入数据组并返回。

集群成员变更时，在一些特殊情况下增删节点操作不成功，用户可能会多次触发同一个增删节点操作，因此正式成员变更也要保证幂等性，图 5.2 描述了正式成员变更的流程：

- 增加节点时，首先进行预成员变更，然后判断是否包括新节点，如果不包括，则说明新节点的加入不会引起本数据组的成员变更，直接返回。其次，判断数据组成员数量是否大于副本数，如果不是，则说明数据组已经执行过一次正式成员变更操作，直接返回。最后删除数据组的最后一个节点，如果该节点是 leader，那么应该立刻发起选举选出新 leader。
- 删除节点时，首先进行预成员变更，然后判断数据组是否包含待删除节点，如果数据组不包含待删除节点，则有两种情况：一是这个节点的删除本身不会引起本数据组的成员变更，二是数据组已经执行过正式成员变更并将这个节点删除，这种情况下应直接返回。最后删除这个节点，如果该节点是 leader，那么应该立刻发起选举选出新 leader。



(a) 增加节点　　(b) 删除节点

图 5.2　数据组正式成员变更流程图

5.1.2　异常处理

在两阶段处理的过程中每个阶段都有可能产生异常，下面对各种异常情况下的处理进行介绍：

- 如果第一阶段第一步中元数据组 leader 一直没有收到超过半数节点的回复，超过一定时间后用户会收到超时的响应。此时用户会重新发送集群扩展请求，系统可能会执行多次预成员变更和多次正式成员变更，由于在实现中保证了操作的幂等性，因此这种情况下不会出现问题。
- 如果第一阶段第二步中某些数据组处理增删节点日志超时，元数据 leader 会不断地重试直至完成，因此会出现数据组执行多次预成员变更操作，同样由于在实现中保证了操作的幂等性，因此提供了正确性的保障。
- 如果第一阶段第一步完成后，元数据组更换了 leader，新 leader 直接进行第二阶段会产生问题。因此在实现时将第一阶段第二步放在元数据组日志提交逻辑中。元数据组成员日志提交时，如果本节点是元数据组 leader，那么先

将日志分发给所有的数据组。这样保证了即使更换了元数据组 leader，第一阶段第二步还是会正常完成。

5.2　数据重分区

在 4.2.3.1 节介绍了数据分区策略采用哈希槽分区方式，集群扩展后由元数据组 leader 进行数据重分区，并将修改后的分区表随着增删节点日志一起同步给其他节点实现统一管理。为了实现以上功能，分区表类 SlotPartitionTable 包含的属性如表 5.1 所示。

表 5.1　SlotPartitionTable 属性说明

属性名	说明
List<Node> **allNodes**	记录集群中的所有节点，按哈希值顺序排列
Node[] **slotNodes**	记录管理每个哈希槽的数据组首节点
long **version**	分区表的版本号
SlotBalancer **slotBalancer**	槽分配器，用于在集群增删节点时对数据进行重分区
Map<Node, Map<Integer, Node>> **previousHolder**	记录数据组获得的所有新 slot 的原有持有组，外层的 key 是新分区管理组的首节点，内层的 key 是 slot，value 是旧分区管理组的首节点

在分区表中，数据组都用首节点来表示，完整的数据组成员可以直接在哈希环上找到首节点后顺序查找即可。在分区表的属性中，版本号用于区分不同的集群扩展操作导致的数据重分区结果。当一个新节点加入集群时，作为元数据组的新成员，follower 要和元数据组的 leader 进行日志同步并依次提交所有的历史集群增删节点操作，此时通过版本号可以直接将这些旧操作跳过。具体实现中采用**元数据组的日志编号**作为版本号。

分区表的作用主要是数据路由和增删节点时对数据进行重分区，为了支持这些功能，分区表类 SlotPartitionTable 包含的接口如表 5.2 所示。增删节点的操作主要通过 SlotBalancer 进行数据重分区，槽分配器主要包含两个接口，如表 5.3 所示。当前实现中采用的是哈希槽均匀分配策略：当增加节点时，首先用总槽数除以节点数算出各个节点应当管理的哈希槽数，然后从各个节点将多余的槽交由新节点管理；当删除节点时，待删除节点的槽均匀地分配给其他节点。

表 5.2　SlotPartitionTable 接口说明

接口名	说明
List<Node> **route**(String *sgName*, long *time*)	给定存储组和时间戳，返回管理该存储组下的给定时间片的数据组
boolean **judgeHoldSlot**(Node *node*, int *slot*)	判断节点 node 是否管理槽 slot 的数据
void **addNode**(Node *node*)	集群增加节点时，更新分区表的相关属性并用槽分配器进行数据重分区
void **removeNode**(Node *node*)	集群删除节点时，更新分区表的相关属性并用槽分配器槽进行数据重分区
ByteBuffer **serialize**()	序列化分区表，当元数据组 leader 进行分区表更新结果同步和将分区表持久化到磁盘时使用
void **deserialize**(ByteBuffer *buffer*)	反序列化分区表，当节点提交分区表更新结果和节点初始化时使用

表 5.3　LoadBalancer 接口说明

接口名	说明
void **moveSlotsToNew**(Node *node*)	当加入节点时，根据一定策略从其他节点接管一些槽，当前实现采用均分哈希槽策略
void **retrieveSlots**(Node *node*)	当删除节点时，根据一定策略将待删除节点管理的槽分配给其他节点，当前实现采用均分哈希槽策略

5.3　数据迁移

两阶段方法的第二阶段中，元数据组提交集群增删节点日志后，新旧分区管理组要开始进行数据迁移，整个数据迁移流程主要分为获取快照、文件传输和文件加载三个步骤，下面将详细介绍每个步骤的实现细节。

5.3.1　功能实现

5.3.1.1　获取快照

旧分区持有者收到新分区持有者获取某些分区的数据快照请求后，首先会检查本地数据是否完整，因为数据组的非 leader 节点的日志是有滞后性的，数据迁移时需要保证数据的完整性。如果本地数据完整，则执行获取快照流程。

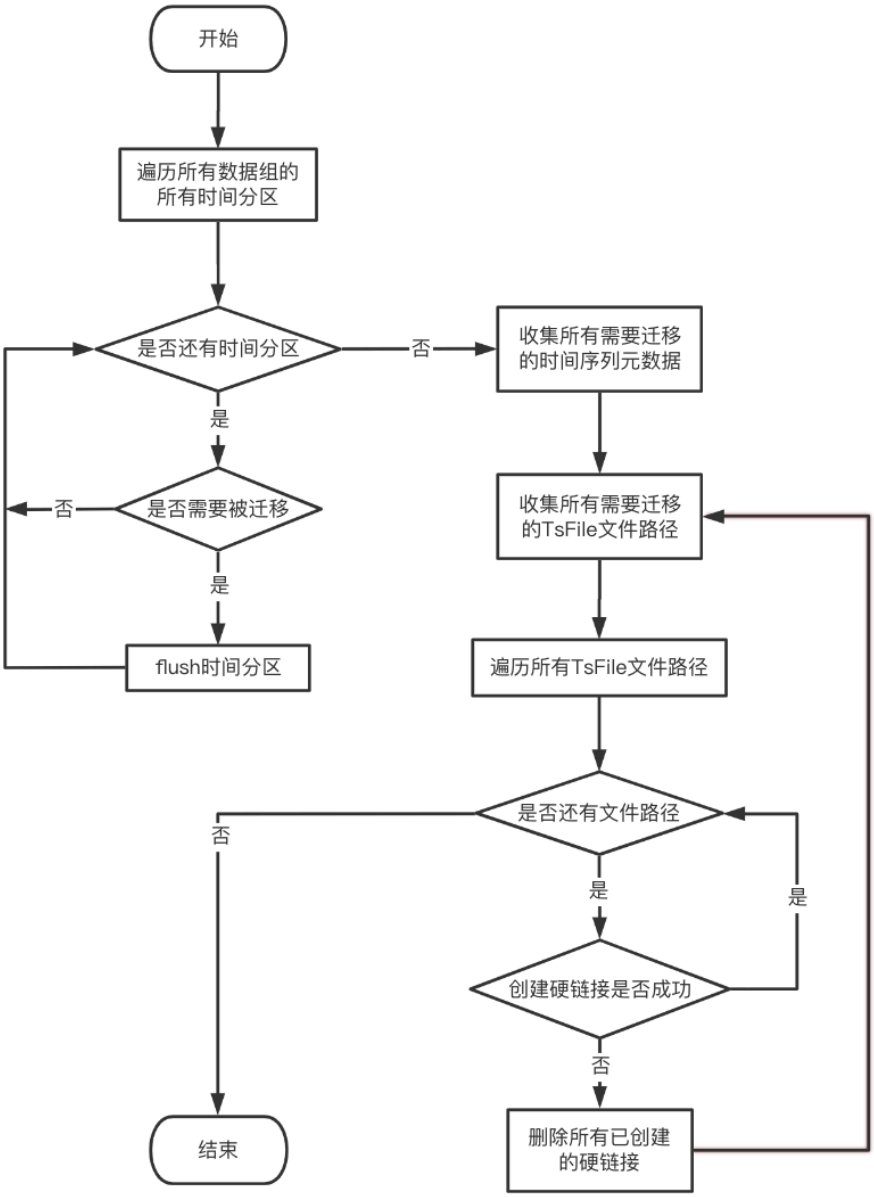


图 5.3　数据迁移获取快照流程图

获取快照首先要对所有需要迁移的存储组时间分区执行 flush 操作实现内存中的数据落盘，由于在此前已经保证了本地数据完整，即数据组日志最新并且本地的分区表信息最新，因此数据组不会再处理新的属于该分区的数据写入请求，保证了所有需要传输的数据都在磁盘上的 TsFile 文件中。

旧分区持有者需要返回的内容分为两部分：一部分是时间序列元数据，另一部分是所有待迁移文件的路径。由于存储引擎存在 merge 操作，新分区持有者拉取文件时，可能会由于 merge 操作已经将文件删除，从而导致数据迁移的不完整性。为了解决这个问题，可以采用对所有文件创建硬链接的方式来保证快照请求者拉取文件时本地文件存在。

如图 5.3，获取文件数据快照首先收集所有需要迁移的 TsFile 文件路径，然后全部创建硬链接，当且仅当所有文件的硬链接创建成功才算完成；否则，说明存在文件已经因为 merge 操作而被删除，这种情况下需要删除所有已创建的硬链接并重新执行流程。当新分区持有者拉取完所有文件后再将硬链接全部移除。

5.3.1.2　文件传输

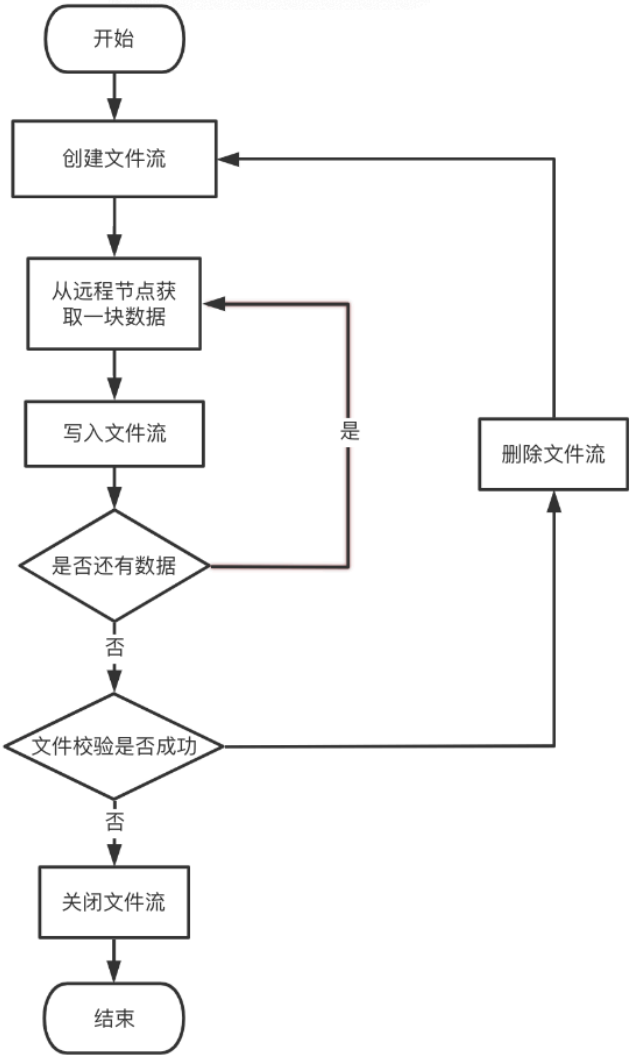


图 5.4　拉取单个远程文件流程图

新分区持有者收到快照后，首先直接加载快照中的元数据注册时间序列，然后依次拉取远程 TsFile 文件进行加载。如图 5.4 所示，拉取远程文件时采用分块传输的方式，拉取完文件的所有数据后需要进行文件校验，主要有两个原因：一是由于网络原因传输的数据可能会出错；二是旧分区持有者的 merge 操作可能会对文件进行修改导致拉取的文件不正确。

5.3.1.3　文件加载

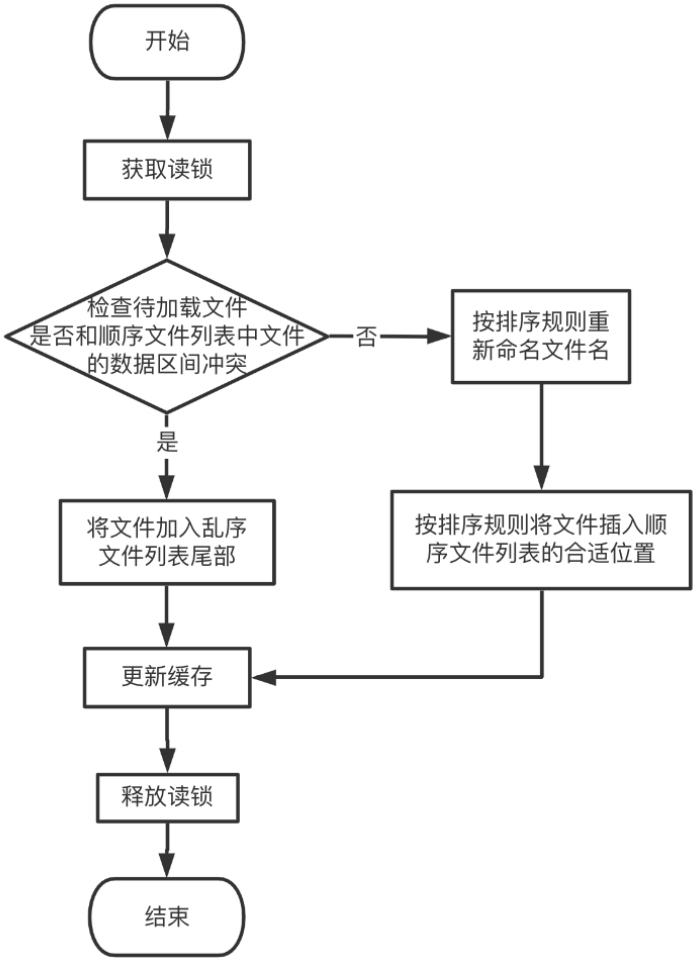


图 5.5　存储引擎文件加载流程图

新分区持有者拉取完一个 TsFile 文件后需要将文件加载到存储引擎中。Apache IoTDB 的存储引擎将顺序数据和乱序数据分开进行管理形成顺序文件列表和乱序文件列表，每个 TsFile 的文件名由文件产生时的时间戳、版本号以及 merge 次数组成。当系统启动时，会通过文件名来对文件进行排序，从而保证文件列表中文件的有序性，因此加载文件时要对文件名进行必要的修改。

存储引擎加载文件时首先要获取写锁，然后根据待加载 TsFile 的.resource 元信息文件中所有设备的起始时间和终止时间来判断是否和顺序文件列表中的文件数据产生冲突。如果产生冲突，就将文件加入乱序文件列表尾部等待合并；否则，按照排序规则对文件进行重命名后再插入顺序文件列表的合适位置。文件重命名具体的方式是将 merge 次数设置为 0，时间戳和版本号设置为待插入位置的前后文件时间戳和版本号的平均值。最后存储引擎更新相关缓存信息并释放写锁完成文件的加载。

5.3.2　异常处理

在数据迁移的过程中，如果文件传输时发生网络故障或者文件校验失败，新分区持有者会多次重试去拉取文件。如果新分区持有者获取快照失败或者拉取文件时超过一定次数后仍未成功，新分区持有者会视为和当前选定的旧分区持有者的数据迁移失败，这种情况下会重新选择新的旧分区持有者重新开始数据迁移流程。

5.4　本章小结

本章介绍了集群扩展机制的实现方法，包括两阶段方法、数据重分区以及数据迁移。两阶段方法中介绍了预成员变更和正式成员变更的流程以及异常情况下的处理，数据重分区中介绍了分区表的设计，数据迁移中介绍了获取快照、拉取文件和加载文件的流程以及异常情况下的处理。

第 6 章　实验与测试

本章将通过实验对集群扩展机制进行功能和性能测试。首先介绍实验环境、测试工具和测试数据，然后对集群扩展机制进行多方面的实验测试和分析，并和 KairosDB 系统进行对比，最后进行总结。

6.1　实验准备

6.1.1　实验测试环境

本实验采用 11 台相同配置的阿里云服务器，其中 1 台机器运行测试工具模拟多个客户端进行操作请求。每台服务器的配置如表 6.1 所示。

表 6.1　实验硬件配置

配置项	描述
操作系统	Alibaba Cloud Linux 2.1903 LTS 64 位
CPU 和内存	4 核 16 GB
实例规格	ecs.g6e.xlarge
网络信息	万兆专有网络
云盘	ESSD 云盘 PL1 400GB
Java 版本	OpenJDK 11.0.10

6.1.2　测试工具

本实验采用 IoTDB-Benchmark[34] 模拟客户端。IoTDB-Benchmark 是一个专门为时序数据库而设计的测试工具，提供了多种数据生成方式，包括方波、正弦波和锯齿波等，同时还提供了乱序数据的生成来模拟各种物联网场景的负载状况。

6.1.3　测试数据

实验中分布式系统的副本数统一设置为 3，由于系统的分布式层包含 raft 日志，因此在测试过程中将关闭存储引擎中的写前日志（WAL）。实验数据类型包括 boolean、int32、int64、float、double 和 text，各类型数据比例为 1:1:1:1:1:1。没有特殊说明的话，本章的所有实验采用的其他参数配置统一如表 6.2 所示。

表 6.2　实验参数配置

配置项	描述
客户端并发数量	20
存储组数量	20
总设备数量	200
总时间序列数量	1 万
batch 大小	100
总数据点数	5 亿
编码方式	PLAIN
乱序数据比例	10%

6.2　实验内容

从用户角度来看，集群扩展分为集群成员变更和数据迁移两个步骤。本实验从两方面进行：一方面是对集群扩展机制的效率测试，包括集群成员变更效率和数据迁移效率；另一方面是集群扩展机制对系统性能的影响测试以及对比测试。

6.2.1　集群成员变更测试

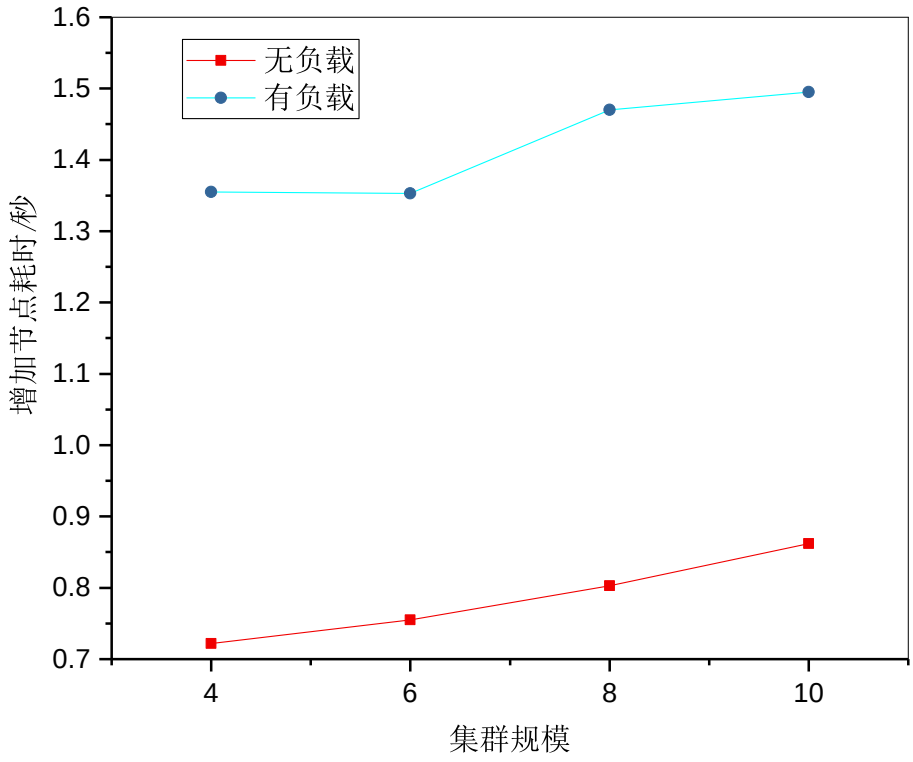


图 6.1　增加节点成员变更效率

集群成员变更的效率指的是在系统允许增删节点时，用户发起增删节点请求到请求成功的耗时。当增加节点时成员变更效率越高表示新节点可以越快分摊系统负载，处理用户请求。当删除节点时成员变更效率越高表示被删除节点可以越快停止提供服务。

如图 6.1 所示，实验测试了在无负载和有负载情况下增加节点成员变更的效率。从图中可以看出 4 节点集群增加至 10 节点的过程中增加节点的耗时会随着集群规模的增大而增大，无负载下增加节点耗时在 700ms~900ms 之间，有负载情况下增加节点耗时在 1.3s~1.5s 之间。

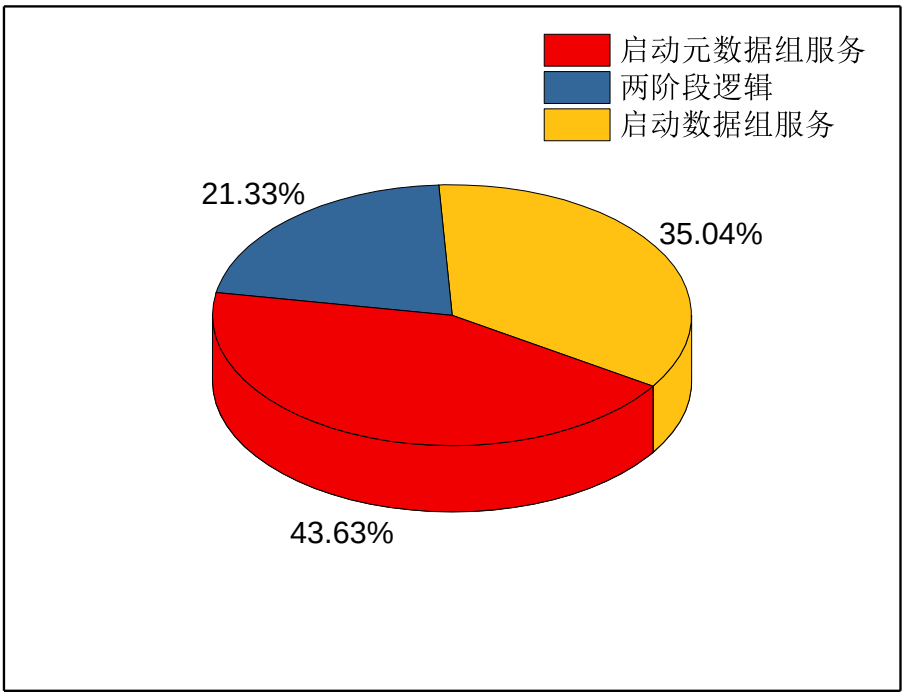


图 6.2　4 节点 3 副本场景下增加节点成员变更耗时分布

新节点启动后，首先启动元数据组服务，然后从配置项中任选一个种子节点发起加入集群请求，最后在系统执行两阶段流程结束后新节点启动所有的数据组服务，新节点加入集群完毕。以 4 节点 3 副本为例，图 6.2 表示了各个阶段的耗时分布，两阶段逻辑的耗时最少。随着集群规模的增长，启动元数据组服务的耗时和启动数据组服务的耗时不会增加，因为每个节点只会启动固定的 1 个元数据组和 3 个数据组，两阶段流程的耗时会随着集群规模的增大而增大。

两阶段流程中的耗时主要分为两部分：

1) 元数据组 leader 将增删节点日志在元数据组中同步，随着集群规模的增加，超过半数的节点数也会增加，因此这部分的耗时会随着系统规模的增大而增大。
2) 元数据组 leader 将增删节点日志发送给所有的数据组，随着集群规模的增大

需要发送的数据组数量也会越多。但是由于发送的过程是异步进行的，因此这个阶段的耗时取决于处理请求最慢的数据组。当系统有负载时，增删节点的日志要和数据写入的日志进行争抢，因此第二部分中数据组的处理效率相较于无负载会明显降低，导致耗时会增加很多。

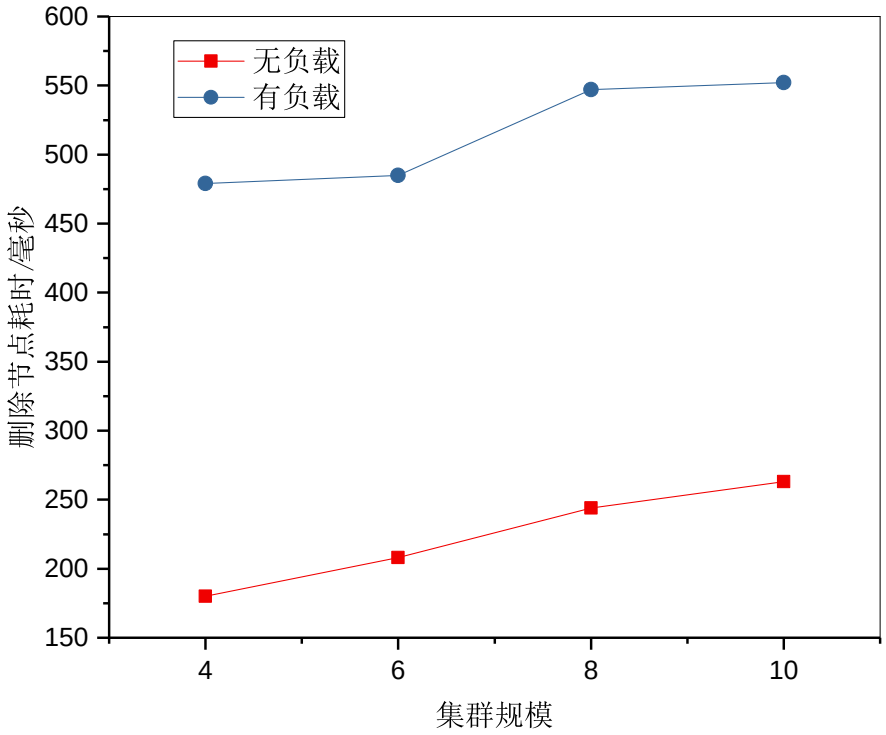


图 6.3　删除节点成员变更效率

如图 6.3 所示，实验测试了在无负载和有负载情况下删除节点成员变更的效率。从图中可以看出，虽然删除节点时比增加节点多了通知被删除节点执行删除流程的步骤，但是由于没有启动元数据组和数据组服务的耗时，删除节点的效率比增加节点高。无负载下删除节点耗时在 180ms~270ms 之间，有负载情况下增加节点耗时在 460ms~560ms 之间。删除节点期间两阶段的耗时变化规律和增加节点时的情况相同。

6.2.2　数据迁移测试

成员变更结束后，随着元数据组成员提交增删节点日志，系统开始进行数据迁移。数据迁移的过程中对系统资源的占用主要来自于数据的传输和数据的加载，在本文的设计中需要迁移的数据包括时间序列元数据和时序数据点。因此数据迁移的测试会从两方面进行：固定时间序列数量时的迁移效率和固定总点数的数据迁移效率。

如图 6.4 所示，实验采用固定的 10 万条时间序列写入不同数据点，从图中可

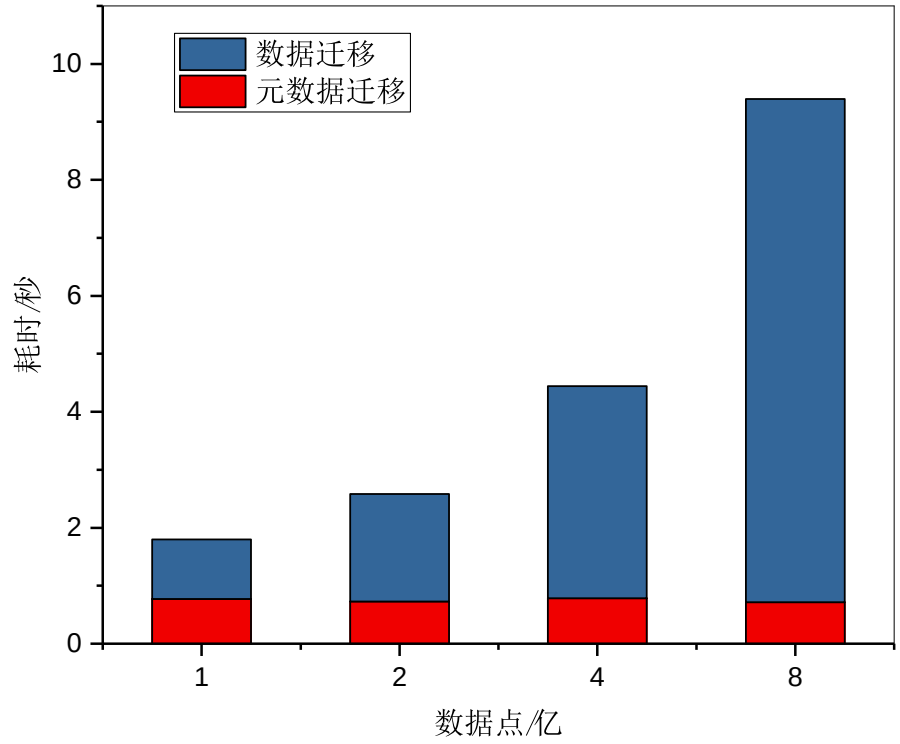


图 6.4　固定时间序列数据量的数据迁移效率

以看出时间序列元数据的迁移耗时不变，每秒可以注册大约 13 万条时间序列，数据迁移的耗时随着数据点的增加而线性增加。由于本文设计中采用的是文件级别的数据传输方式，在固定时间序列的数量下存储引擎产生的数据文件大小和写入的数据点数成正比，因此数据部分的迁移耗时会呈线性关系，每秒约 1 亿点的迁移效率。

如图 6.5 所示，实验固定写入 1 亿点数据，变化时间序列数量测试效率。从图中可以看出时间序列元数据的迁移耗时随时间序列的数量增加而线性增长，每秒可以注册大约 13 万条时间序列。此外，数据迁移的耗时也随着时间序列的元数据的增加而增加，下面对于这个实验结果进行分析。

数据写入时首先将数据写在内存中的 MemTable 中，当 MemTable 达到一定阈值后再 flush 持久化到磁盘中的 TsFile 中。由 2.4.2 节可知，每条时间序列 flush 后的数据组成一个 chunk 并且包含一个 chunk header，每个设备 flush 后的数据组成一个 chunk group 并且包含一个 chunk group footer。当时间序列数量增大时，由于 MemTable 中将使用更多的内存来表示时间序列，因此 flush 时的 MemTable 里包含的数据点会减少，flush 后的 chunk header 和 chunk group footer 等元信息会增加，因此相同的总数据点下时间序列越多，生成的 tsfile 文件会越大，如图 6.6 所示。

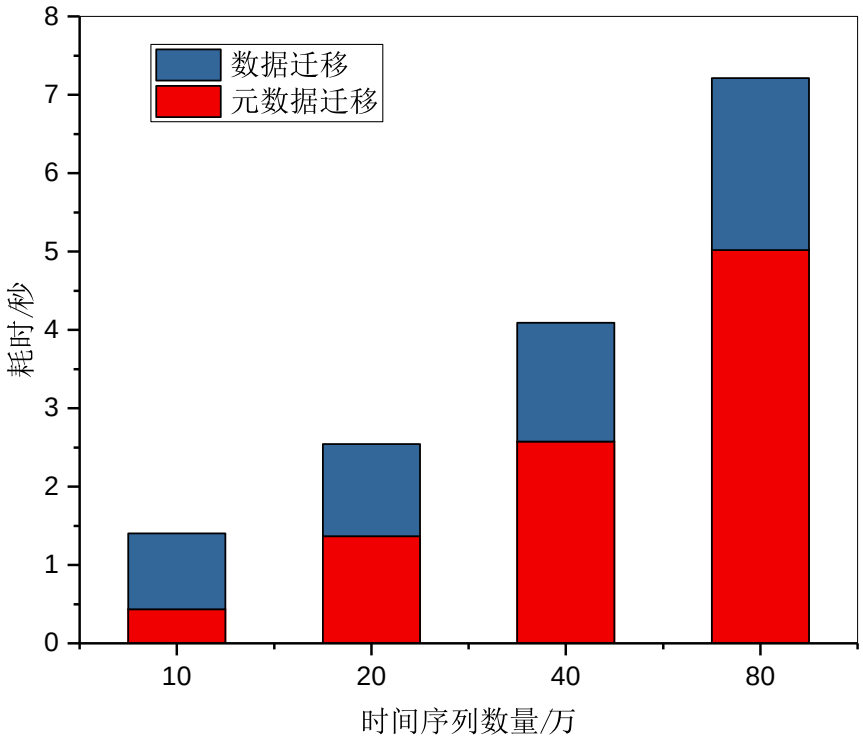


图 6.5　固定总点数的数据迁移效率

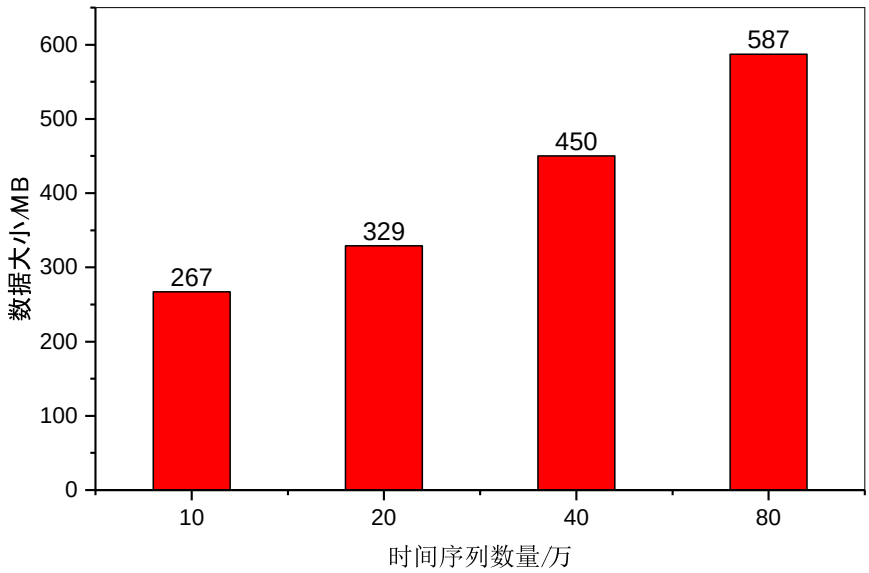


图 6.6　不同数量时间序列相同总点数下的数据大小

6.2.3　系统性能测试

由于目前 Apache IoTDB 的分布式查询功能并不稳定和完善，因此下面以 4 节点 3 副本为例测试增删节点对于系统整体写入性能的影响并进行分析。实验首先使用 IoTDB-Benchmark 向 4 节点 3 副本的集群中写入 5 亿个数据点，然后增删节点观察集群性能变化。

6.2.3.1　增加节点

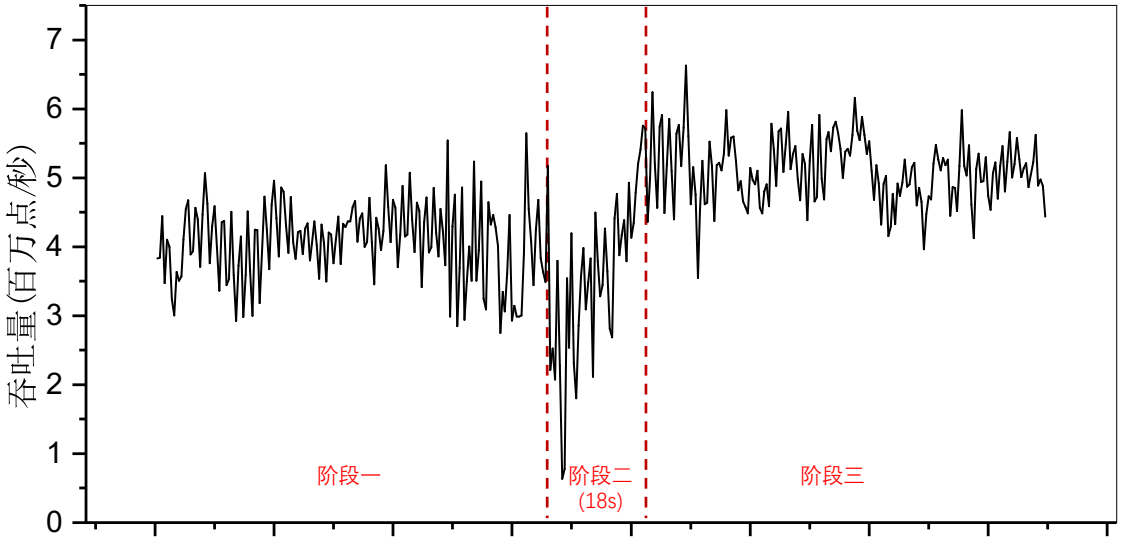


图 6.7　5 亿数据点下集群增加节点写入吞吐量变化

如图 6.7 所示，阶段一是集群规模为 4 时的系统写入性能曲线。阶段二是增加节点时系统规模由 4 变到 5 所经历的一个过渡阶段，即新节点发送加入集群请求到集群中所有数据组的数据迁移完成的过程。阶段三是系统稳定后集群规模为 5 时的写入性能曲线。从这张图中可以分析出以下结论：

1) 阶段二中系统开始增加节点时性能会有明显的下降，主要原因是数据组成员变更有三分之一的概率会将数据组的 leader 进行替换，这种情况下数据组需要进行 leader 的选举过程，在新的 leader 选出来之前所有需要该数据组处理的写入请求都会被拒绝。然后由于多个数据组之间会进行数据迁移，系统资源的消耗也对性能产生了影响。随着数据迁移的逐渐完成以及新的数据组的加入分摊了系统的写入压力，系统性能呈现逐渐上升的过程。整体上看过渡的阶段二相较于阶段一的吞吐量下降约 10%，持续时间为 18s。
2) 阶段三中系统稳定后相较于阶段一的写入吞吐量提升了将近 20%。在集群规模为 4 时每个数据组负责 5 个存储组的数据写入，当集群规模为 5 时每个数据组负责 4 个存储组的数据写入。新加入的数据组分摊了其他数据组的写入压力，从而导致整体写入性能的提升。

6.2.3.2　删除节点

如图 6.8 所示，阶段一是集群规模为 4 时的系统写入性能曲线。阶段二是删除节点时系统规模由 4 变到 3 所经历的一个过渡阶段，即用户发送删除节点请求到集群中所有数据组的数据迁移完成的过程。阶段三是系统稳定后集群规模为 3 时

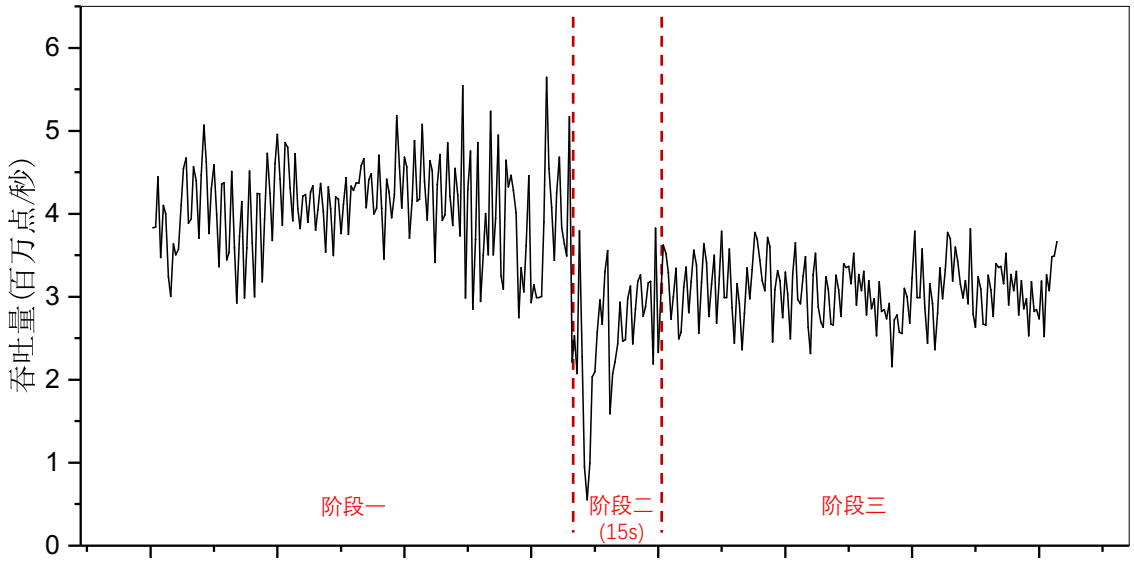


图 6.8　5 亿数据点下集群删除节点写入吞吐量变化

的写入性能曲线。从这张图中可以分析出以下结论：

1) 阶段二中系统开始删除节点时性能会有明显的下降，主要原因是数据组成员变更有三分之一的概率会将数据组的 leader 进行替换，这时数据组需要选举出新 leader，在新的 leader 选出来之前所有需要该数据组处理的写入请求都会被拒绝。然后由于多个数据组之间会进行数据迁移，系统资源的消耗也对性能产生了影响。随着数据迁移的逐渐完成，系统性能呈现逐渐上升的过程。由于被删除的数据组将写入负载平分给了其他数据组，因此整体上看过渡的阶段二相较于阶段一的吞吐量下降得比较多，约 25%，持续时间为 15s。
2) 阶段三中系统稳定后相较于阶段一的写入吞吐量降低了约 20%。在集群规模为 4 时每个数据组负责 5 个存储组的数据写入，当集群规模为 3 时有两个数据组要负责 7 个存储组的数据写入，一个数据组要负责 6 个存储组的数据写入。被删除的数据组将写入负载分配给了其他数据组，从而导致整体写入性能的降低。

6.2.4　KairosDB 对比测试

本小节将对分布式 Apache IoTDB 和 KairosDB 系统在不同集群规模时的写入性能进行测试，对比两个系统的扩展性能力。KairosDB 系统底层基于 Cassandra，实验中采用默认配置。

如图 6.9 所示，分布式 Apache IoTDB 和 KairosDB 的写入性能都随着集群规模的增长而线性增长，其中 Apache IoTDB 的写入性能在每秒 400~700 万点，KairosDB 的写入性能在每秒 30~50 万点，整体性能相差一个数量级。此外，随着集群规模的

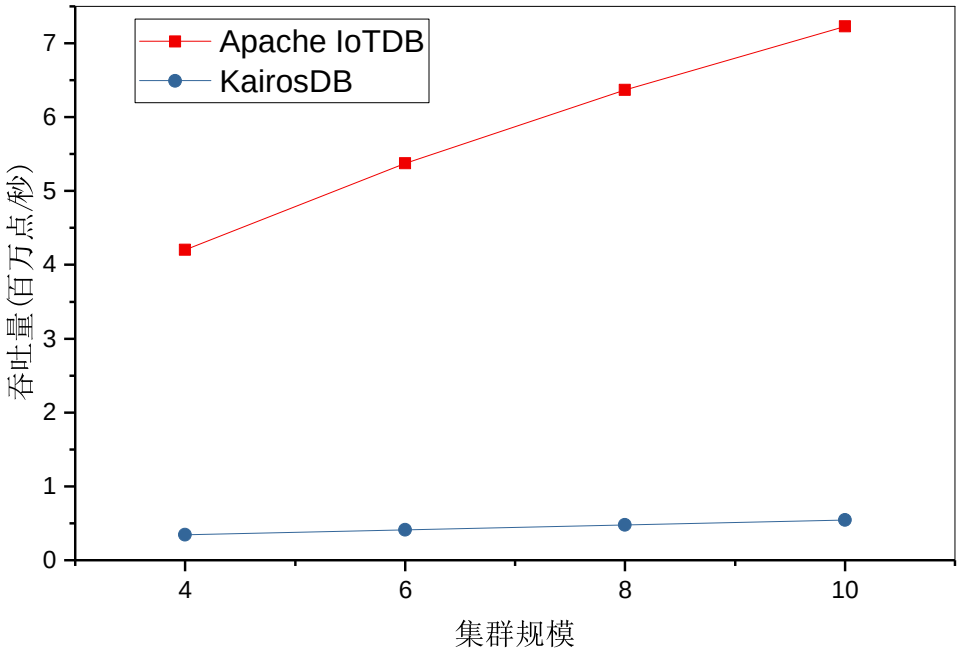


图 6.9　Apache IoTDB 和 KairosDB 在不同集群规模下的写入性能对比

增长，分布式 Apache IoTDB 系统写入性能的增长速率大于 KairosDB，说明 Apache IoTDB 的分布式扩展能力优于 KairosDB。

6.3　实验总结

本章从两个方面对集群扩展机制进行了测试，实验首先表明增删节点的成员变更可以在秒级完成，其次论证了文件级别的数据迁移方案的高效性，然后测试并分析了集群扩展机制对系统性能的影响，最后和 KairosDB 系统进行对比说明了 Apache IoTDB 有更好的分布式扩展能力。

第 7 章　集群扩展机制对比

本章通过和其他系统的集群扩展机制进行对比，分析本文设计的集群扩展机制的优势和不足。由于 Cassandra 系统是采用 Gossip 协议的去中心化系统，TiKV 系统采用主从架构且 TiKV 系统的副本之间同样采用 raft 协议来保证一致性，因此下面将通过这两个典型的系统进行对比和分析。

7.1　Cassandra 集群扩展机制

Cassandra 系统采用一致性哈希的方式进行数据分区，并通过加入虚拟节点的方式将数据分配得更加均匀，每个虚拟节点在哈希环上的位置称为 token，因此每个节点对应哈希环上的多个 token。系统的节点状态信息和数据分区信息通过 Gossip 协议分发在集群所有节点中，每个节点通过一致性哈希算法和集群中各节点的 token 位置就可以对数据的写入和查询请求进行路由和处理。Cassandra 采用法团协议来保证查询的正确性，当集群节点的副本数为 3 时，读写副本数都会为 2。

下面以在 3 副本集群中增加节点为例介绍 Cassandra 集群扩展的流程：

1. 新启动的节点开启 Gossip 服务，并等待一段时间来获取当前的集群信息，包括每个节点的状态和数据分区信息。
2. 新节点根据当前的集群信息计算如果自己加入集群后会接管哪些 token 区间的数据，此时不更新分区信息。
3. 对要接管的每个 token 区间，新节点会从当前持有该区间数据的三个副本中选择一个节点并启动数据迁移。
4. 数据迁移结束后，新节点更新分区信息，并用 Gossip 协议通知其他节点本节点正式加入集群。

由于 Gossip 协议维护的集群状态是最终一致性的，因此 Cassandra 在集群扩展过程中可能会产生查询结果不一致甚至数据丢失的情况，下面将举例说明。由于数据写入时不一定所有副本全都写入成功，假设某个分区的数据由节点 1、2、3 来管理，节点 1 和节点 3 数据写入成功，节点 2 写入失败，但是由于满足法团协议的要求因此写入请求会返回成功。此时集群加入节点 4，节点 4 会代替节点 3 管理这个分区数据，这种情况下会产生以下几个问题：

1. 节点 4 选择了节点 2 进行数据迁移，由于节点 2 的数据不是最新的，因此节点 4 加入集群后，三个副本 1、2、4 中只有副本 1 的数据是最新的，这种情

况下进行数据查询时会有三分之一的概率不能读到最新的数据。

2. 在节点 4 数据迁移的过程中，由于新节点还没有正式加入集群，因此新的数据写入还是会写向节点 1、2、3，如果仍旧只有节点 1 和节点 3 写入成功，那么节点 4 加入集群后，也会存在只有 1 个副本数据最新的情况，影响查询的一致性。
3. 节点正式加入集群的信息是最终一致性的，也就是存在某些节点知道新节点的加入，还有些节点不知道新节点加入的中间状态。此时，如果处理写入请求时协调者节点不知道新节点的加入并且节点 1 和节点 3 写入成功，处理查询请求时协调者节点知道新节点的加入并向节点 2 和节点 4 发起查询请求，还是会出现不能读到最新数据的情况。

综上所述，和本文设计的集群扩展机制相比，Cassandra 系统的集群扩展机制有如下特点：

1. Cassandra 系统采用先进行数据迁移再加入集群并更新数据分区信息的方式来实现集群扩展，保证了节点正式加入集群时已经获取到了要管理的那部分数据，但是当需要迁移的数据量比较大时节点加入集群的时间会很长。本文设计的集群扩展机制是采用节点先加入集群并更新数据分区信息再进行数据迁移的方式，相较于 Cassandra 的集群扩展策略，新增的节点可以很快地加入集群并分摊其他节点的写入负载，但是由于节点加入后才开始数据迁移，因此数据迁移完成前的查询要合并新老分区持有者的查询结果。
2. Cassandra 在进行集群扩展时不会严格保证数据的一致性，可能会带来数据副本数的降低以及存在数据查询不一致的情况，此外数据只有单副本时也容易出现数据丢失的现象。针对这些问题虽然 Cassandra 也提供了 repair 机制进行副本修复来解决，但是代价昂贵，极大程度地增加了运维成本。相比之下，本文设计的集群扩展机制通过 raft 协议和两阶段方法保证了数据的强一致性，不需要引入额外的机制。

7.2　TiKV 集群扩展机制

TiKV 负责 TiDB 的底层数据存储，是一个分布式 Key-Value 数据库。如图 7.1 所示，TiKV 采用范围分区的分区策略并以 Region 为单位存储数据，每个 Region 负责一个分区范围的数据。Raft 协议保证了 Region 与副本之间的一致性，因此 Region 构成的 raft 组可以类比为 Apache IoTDB 集群中的数据组概念。

PD 是 TiKV 集群的控制中心，负责整个集群的调度和负载均衡，维护集群所有节点的状态和负载情况。TiKV 的节点通过心跳主动上报节点信息和负载状况，

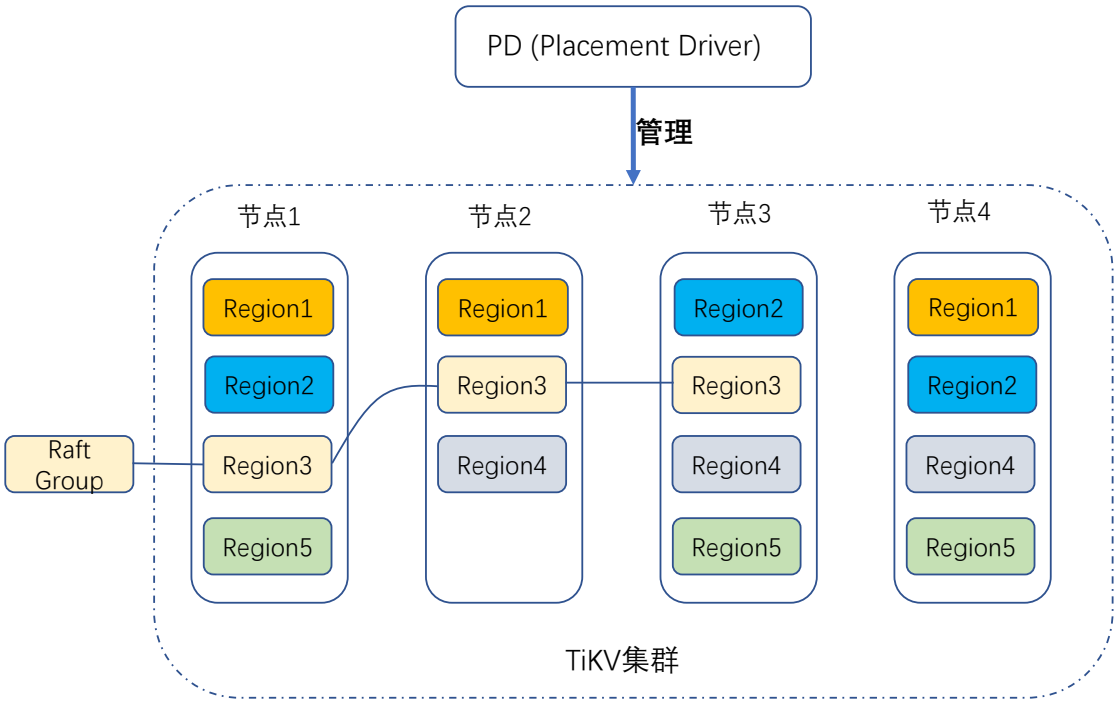


图 7.1　TiKV 架构图

PD 根据 Region 的大小决定是否执行 Split/Merge Region 操作。当某个 Region 的数据量达到一定大小时，PD 会将这个 Region 分裂为两个新 Region，即将范围分区的一个大分区划分为两个小分区，此时一个 raft 组会变成两个 raft 组；当某两个 Region 的数据量很少时，PD 会对这两个 Region 实现合并，即将范围分区的两个小分区合并为一个大分区，此时两个 raft 组会合并成一个新的 raft 组。此外，PD 还维护分区元信息提供全局路由功能。

TiKV 的集群扩展流程很简单，新加入的节点启动后通过心跳向 PD 发送自身信息即完成了节点加入，PD 根据集群各节点的负载状况将一些 Region 交由新节点管理，本质上是进行了多个 raft 组的成员变更。删除节点时将被删除节点上每个 Region 的数据迁移到由 PD 选择的新节点中，本质上也是进行了 raft 组的成员变更。TiKV 执行一个 raft 组的成员变更时，采用了和本文集群扩展相同的方式，即先增后删，成员变更过程的一致性由 PD 进行管理和保证。

综上所述，和本文设计的集群扩展机制相比，TiKV 系统的集群扩展机制有如下特点：

1. 由于采用 PD 进行统一管理，TiKV 集群可以根据负载均衡策略按需选择一个新节点来替换当前 raft 组的某个成员。而在 Apache IoTDB 分布式系统中由于采用一致性哈希环的方式来决定每个 raft 数据组的成员，因此 raft 组的成员不能自由分配。造成这种差异的原因是 PD 存储 raft 数据组成员信息采

用了查找表的方式，而 Apache IoTDB 集群中由于 raft 组成员信息和分区信息要在所有节点中保存，采用查找表的方式会带来大量的内存占用，因此采用的是一致性哈希环的方式。

2. 由于采用 PD 进行负载均衡，TiKV 在集群扩展时不需要立刻进行数据的重分区和 raft 组的成员变更，PD 可以根据负载状态在需要的时候进行 Region 的成员替换和 Region 的拆分/合并操作。而在 Apache IoTDB 系统中，采用一致性哈希的方式会在集群扩展时就带来了成员变更和数据分区信息的修改。
3. TiKV 集群中 raft 组的生成和删除是在 PD 控制 Region 的拆分/合并时完成的，和集群扩展阶段解耦。而在 Apache IoTDB 中新节点的加入必然会带来一个新数据组的生成，旧节点的删除必然会带来一个旧数据组的删除，本质上还是由于采用了一致性哈希环的方式来决定数据组和数据组的成员。

7.3　本章小结

本章通过和 Cassandra 系统、TiKV 系统的集群扩展机制进行对比，分析了本文设计的集群扩展机制的优势和不足，为今后的改进工作提供了帮助。

第 8 章　总结与展望

8.1　本文总结

本文的主要工作是在已有的 Apache IoTDB 的分布式框架上设计并实现集群扩展机制，主要贡献如下：

1. 针对需要保证多数据组成员变更安全性和数据安全性的需求，提出了一种基于一致性哈希和 raft 协议的两阶段集群扩展方法（Two-stage method of cluster scalability，简称 TSM）。第一阶段将集群成员变更信息同步于所有数据组中并执行数据组的预成员变更；第二阶段执行数据组的正式成员变更和数据迁移。该方法避免了数据组出现脑裂的情况，保证了成员变更的安全性。此外，该方法保证了在数据迁移前集群中所有写入请求都会由新的数据分区持有组处理，从而保证了数据的安全性。
2. 针对需要提供高效数据迁移策略的需求，提出了基于哈希槽和时间片的文件级数据迁移方法（File-level data migration method，简称 FDM）。该方法采用哈希槽分区策略，每个节点以槽为单位管理数据。此外，该方法利用了存储引擎按照时间片进行数据管理使得每个数据文件的所有数据都不会跨越两个槽的特性，在进行数据迁移时直接对整个文件进行传输和加载，从而避免了数据查找和重建元信息等过程，减少了对系统资源的使用。
3. 在 Apache IoTDB 的分布式框架上实现了集群扩展机制，并设计实验对集群扩展机制进行功能和性能测试。首先对集群扩展机制的效率进行了测试，测试结果表明集群扩展的成员变更可以在秒级完成，并验证了文件级别的数据迁移方案的高效性。其次，测试并分析了集群扩展机制对系统性能的影响。

8.2　不足与展望

本文设计的集群扩展机制虽然可以支持集群高效地进行单成员变更操作，但还存在一些不足和可改进的内容，主要分为以下几个方面：

1. 集群不支持同时增删多个节点操作。在当前实现方案下更复杂的集群扩展操作会拆分为多次单成员变更操作进行，而每次集群扩展操作只有在前一个集群扩展操作的所有任务完成后才可以执行，包括数据迁移和日志的 catch up 等，否则请求会被拒绝。
2. 集群缺少负载均衡器。当前的分布式系统只有在集群增删节点时会采用均分

哈希槽的方式来均衡负载。这种实现在集群负载倾斜程度比较高时就不能很好地将负载进行合理分配，系统设计并实现负载均衡器后，可以针对不同目标采用相应的负载均衡策略来更好地分配负载。

3. 集群不支持灵活地分配 raft 数据组成员。当前的实现下，raft 数据组的成员由一致性哈希算法决定，不够灵活。后续工作可以考虑采用查找表搭配负载均衡器来更合理自由地分配 raft 数据组成员，但是需要解决查找表会占用大量内存的问题。

参考文献

[1] Stoyanova M, Nikoloudakis Y, Panagiotakis S, et al. A survey on the Internet of things (IoT) forensics: challenges, approaches, and open issues[J]. IEEE Communications Surveys & Tutorials, 2020, 22(2): 1191-1221.

[2] Fu T. A review on time series data mining[J]. Engineering Applications of Artificial Intelligence, 2011, 24(1): 164-181.

[3] Ye F, Liu Z, Zhu S, et al. Research of Benchmarking and Selection for TSDB[C]//International Conference on Algorithms and Architectures for Parallel Processing. Springer, 2019: 642-655.

[4] Namiot D. Time series databases.[J]. DAMDID/RCDL, 2015, 1536: 132-137.

[5] Naqvi S N Z, Yfantidou S, Zimányi E. Time series databases and influxdb[J]. Studienarbeit, Université Libre de Bruxelles, 2017: 12.

[6] Prasad S, Avinash S. Smart meter data analytics using opentsdb and hadoop[C]//2013 IEEE Innovative Smart Grid Technologies-Asia (ISGT Asia). IEEE, 2013: 1-6.

[7] Hawkins B. Kairos db: Fast time series database on cassandra[M]. jan, 2017.

[8] Wang C, Huang X, Qiao J, et al. Apache IoTDB: time-series database for internet of things[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2901-2904.

[9] 李天安, 黄向东, 王建民, 等. Apache IoTDB 的分布式框架设计[J]. 中国科学: 信息科学, 2020, v.50(05): 5-20.

[10] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). 2014: 305-319.

[11] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web[C]//Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. 1997: 654-663.

[12] 申德荣, 于戈, 王习特, 等. 支持大数据管理的 NoSQL 系统研究综述[J]. 软件学报, 2013, 24(08): 1786-1803.

[13] 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). Ieee, 2010: 1-10.

[14] Dubreuil M, Gagné C, Parizeau M. Analysis of a master-slave architecture for distributed evolutionary computations[J]. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 2006, 36(1): 229-235.

[15] Lakshman A, Malik P. Cassandra: a decentralized structured storage system[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

[16] Stoica I, Morris R, Liben-Nowell D, et al. Chord: a scalable peer-to-peer lookup protocol for internet applications[J]. IEEE/ACM Transactions on networking, 2003, 11(1): 17-32.

[17] Vora M N. Hadoop-HBase for large-scale data[C]//Proceedings of 2011 International Conference on Computer Science and Network Technology: volume 1. IEEE, 2011: 601-605.

[18] Hunt P, Konar M, Junqueira F P, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems.[C]//USENIX annual technical conference: volume 8. 2010.

[19] Van Renesse R, Dumitriu D, Gough V, et al. Efficient reconciliation and flow control for anti-entropy protocols[C]//proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. 2008: 1-7.

[20] Aguilera M K. Stumbling over consensus research: Misunderstandings and issues[M]// Replication. Springer, 2010: 59-72.

[21] Gifford D K. Weighted voting for replicated data[C]//Proceedings of the seventh ACM symposium on Operating systems principles. 1979: 150-162.

[22] Haerder T, Reuter A. Principles of transaction-oriented database recovery[J]. ACM computing surveys (CSUR), 1983, 15(4): 287-317.

[23] 江天, 乔嘉林, 黄向东, 等. 开源软件中的大数据管理技术[J]. 科技导报, 2020, 38(3): 103-114.

[24] Brewer E. Cap twelve years later: How the" rules" have changed[J]. Computer, 2012, 45(2): 23-29.

[25] Lamport L, et al. Paxos made simple[J]. ACM Sigact News, 2001, 32(4): 18-25.

[26] Burrows M. The Chubby lock service for loosely-coupled distributed systems[C]//Proceedings of the 7th symposium on Operating systems design and implementation. 2006: 335-350.

[27] Huang D, Liu Q, Cui Q, et al. TiDB: a Raft-based HTAP database[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 3072-3084.

[28] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data [J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 1-26.

[29] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store[J]. ACM SIGOPS operating systems review, 2007, 41(6): 205-220.

[30] Chen Z, Yang S, Tan S, et al. Hybrid range consistent hash partitioning strategy–a new data partition strategy for NoSQL database[C]//2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, 2013: 1161-1169.

[31] Felber P, Kropf P, Schiller E, et al. Survey on load balancing in peer-to-peer distributed hash tables[J]. IEEE Communications Surveys & Tutorials, 2013, 16(1): 473-492.

[32] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[33] Vogels W. Eventually consistent[J]. Communications of the ACM, 2009, 52(1): 40-44.

[34] Liu R, Yuan J. Benchmarking time series databases with IoTDB-benchmark for IoT scenarios [J]. arXiv preprint arXiv:1901.08304, 2019.

致　谢

本人在读研的三年期间，受到了很多人的帮助和支持，在此向他们表示真诚的感谢。

感谢导师王建民老师，带领我走进了实验室，接触了实验室项目 IoTDB。在整个研究生期间，王老师经常来到实验室和我们共同探讨实验室项目的进展和落地情况，并在项目遇到问题的时候给予我们很大的帮助。在学习上，王老师更是带领我养成了严谨的科学素养，使我受益匪浅。

感谢黄向东老师的悉心教导，他是我在读研期间接触时间最长的一位老师。不仅在做项目的过程中教会了我很多知识，极大地提高了我的工程和科研能力，在生活上他也是非常有趣的人，对我们十分关心，让我们感到亲切和温暖。

感谢同一年入学的勾王敏浩、苏月、江天和芮蕾同学；感谢乔嘉林、徐毅、刘昆、曹高飞、毛东方等学长和学姐；感谢田原、谭新宇等学弟。有了你们，让我的实验室生活更加精彩和难忘。

感谢我的两位室友刘志成和卢光宏，经常在宿舍制造欢笑，让我的业余生活更加放松和有趣。

感谢我的女朋友赵旭蕾，在我学习和生活中遇到困难时给我予以鼓励和支持，陪伴我渡过各种难关。

感谢默默支持和关心我的父母，感谢你们的养育之恩，我会用余生来回报你们；感谢我的姐姐李淑娴，感谢你给予我不断向前的动力。

感谢 ThuThesis 模板，帮我节省了大量的论文排版时间。

本论文承蒙国家重点研发计划 (2019YFB1707001)、自然科学基金 (61802224) 资助，特此致谢。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间完成的相关学术成果

个人简历

1996 年 1 月 17 日出生于天津市。

2014 年 9 月考入北京邮电大学电子工程学院电子科学与技术专业，2015 年 3 月转专业至北京邮电大学信息与通信工程学院通信工程专业，2018 年 7 月本科毕业并获得工学学士学位。

2018 年 9 月免试进入清华大学软件学院攻读软件工程硕士至今。

在学期间完成的相关学术成果

学术论文：

[1] 李天安, 黄向东, 王建民, 等. Apache IoTDB 的分布式框架设计 [J]. 中国科学: 信息科学, 2020, v.50(05):5-20.

指导教师学术评语

时间序列数据的分布式管理是研究热点之一。李天安同学针对一致性哈希机制和多 Raft 组协议引入的成员变更的原子性问题展开研究，选题具有理论意义与应用价值。

主要工作包括：

1. 对多 Raft 组带来的成员变更问题，提出两阶段集群扩展方法，避免集群出现分裂。

2. 针对数据迁移问题，利用哈希槽与时间片粒度进行分区，确保数据文件可以被整体迁移，获得较小的迁移代价。

3. 在 Apache IoTDB 上进行了实现与验证。

论文结构清晰，工作成果已提交至 Apache 社区。

论文工作表明，李天安同学掌握了软件工程学科的基础理论和专业知识，具备了独立展开工程技术工作的能力，达到了工程硕士的学术水平，同意组织论文答辩。

答辩委员会决议书

李天安同学研究了 Apache IoTDB 集群扩展机制，论文选题具有重要的理论意义与应用价值。

论文主要工作包括：

1. 提出了基于一致性哈希（DHT）和 Raft 协议的两阶段集群扩展方法；

2. 提出了基于哈希槽与时间片的文件级数据迁移方法；

3. 对上述模型和方法进行了实验验证。

论文叙述清楚，结构合理。论文反映李天安同学掌握了软件工程领域的基础理论和专门知识，掌握了解决工程问题的先进技术方法和现代技术手段，具备了独立担负工程技术工作的能力。

论文答辩叙述清楚，回答问题正确，系统演示正常。答辩委员会一致同意通过毕业论文答辩，建议授予李天安同学工程硕士学位。