÷ 3Á & _____ @& M201172374

@ÛL 10487 5ÿ— _____

華中科技大學

;×@H>5<>%

LSMQ:0Aû LSM-Tree ¢

)2$,14ë:8B>

A Thesis Submitted to Academic Evaluation Committee of Huazhong University of Science and Technology for the Degree of Master of Engineering

LSMQ：A Persistence Message Queue Based on LSM-Tree

Candidate : Fang Bo

Major : Computer Architecture

Supervisor : A.P. Shi Zhan

Huazhong University of Science and Technology

Wuhan, Hubei 430074, P. R. China

Jan.2014

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于　保密□，在_____年解密后适用本授权书。

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

分布式应用程序为了降低模块相互之间的耦合程度以及更好的应对可扩展、可操作性的要求，通常引入消息队列来降低模块之间的耦合和平衡各模块处理能力，消息队列允许分布式系统模块之间在保证遵守同样接口约束的前提下，独立的扩展和修改各自的处理过程。各模块对数据的处理能力不同，持久化消息队列能保证消息持久化到他们已经完全处理好，从而避免了数据丢失。消息队列解耦了处理过程，所以系统扩容时，各模块只需独立增加各自的处理能力即可，大大增强了分布式系统的可扩展性。

在充分调研现有消息队列的基础上，发现现有的消息队列系统存在消息无法持久化、消息持久化和高吞吐量之间无法兼顾、单机容量有限等问题。LSM-Tree（Log-Structured Merge tree）是一个为了提高 B-树在更新插入数据时的性能而提出来的优秀算法，思想来源于日志文件系统，主要优点是在不牺牲查找效率的前提下，大幅度提升随机写、随机更新、删除效率。在研究了现有的几种消息队列的基础上，通过引入 LSM-Tree 结构，设计和实现了一个高吞吐量、消息堆积能力强的持久化消息队列 LSMQ。结果显示，在单机容量、读写吞吐量方面，LSMQ 表现均达到预期。

测试结果表明，LSMQ 读性能达到了 12000 条消息/秒，写性能达到了 15000 条消息/秒，略低于内存队列 Redis，远远高于基于 B+树的持久化消息队列 ActiveMQ。在消息堆积能力方面，由于 LSMQ 轻量级的内存结构设计和基于磁盘的存储结构，LSMQ 可以在低内存占用的情况下，达到充分利用磁盘存储消息的目的。

关键词：消息队列，持久化，LSM-Tree，存储模型

Abstract

Distributed applications in order to reduce the degree of coupling between each module and better cope with scalability, operability requirements, message queues are usually introduced to reduce the coupling between modules and balance processing capabilities of each module. On the premise of implementing the same interface, message queue allow distributed system components to extend its processing capacity independently. Persistence message queue can ensure that the message is not lost until it is been consumed.Decoupling the message queue process, so the system expansion, independent of each module simply to increase their processing capacity, greatly enhancing the scalability of distributed systems.

This paper research a log of message queue system and find out that some message queue system provides good ready and write performance, but lack of persistence characteristics and priority control. The Log-Structured Merge-Tree (or LSM-tree) is a tree structure in order to improve performance when updating data insertion algorithm proposed to excellent, LSM-tree is designed from the log data system. This paper analyzed different message queue system, design and implements a message queue system with high read and write performance and persistence characteristics called LSMQ. Test results shows that, compared with other message queue, such as ActiveMQ and Redis, LSMQ perform well in read and write volume and Single system capacity.

The Test result shows that LSMQ reached 9000 read MPS and 6000 write MPS.And perform better than famous persistence message queue ActiveMQ. In the message persistance capacity test, due to the LSMQ lightweight design and memory disk-based storage structure, LSMQ can in the case of low memory usage.

Keywords： Message Queue，persistence，LSM-Tree，Storage model

目 录

1 绪论

1.1 引言

随着网络基础设施的初步成熟，越来越多的系统运行在分布式环境中，分布式的目标是希望系统之间的通信不再因为不同语言与平台而产生困难，分布式系统一般拥有多个子系统，对消息的处理是它的基础设施，消息的本质是一中数据载体，它包含了消费者与生产者都能识别的字段[4]，这些数据是在不同的模块之间进行传递和处理，消费者可能自身的处理能力完全不同，在许多分布式系统中，消息队列与文件船体和远程调用而言，有着更轻量级的设计和更广阔的市场。因为消息具有更好的平台无关性，并能够很好的支持并发和异步调用。

消息队列是一种由消息传送机制或队列模式组成的中间件，是基于事务模型的松耦合和可靠的网络通信服务，利用消息队列可以实现分布式系统中平台无关的数据交流，并基于数据通信来进行分布式系统的集成[5]。消息队列可以看作是网络中暂时存放路由消息的地方，是在消息传递过程中保存消息的容器。队列的主要目的是提供路由并保证消息的传递，如果发送消息时接受者不可用或不在线，消息队列会将消息储存起来，直到可以成功的传递消息。现代的消息队列技术还可以提供可靠的交付保证，同时通过构建分布式的集群，允许消息在系统内堆积，消息队列是分布式系统的基础组件，主要起子系统间解耦合、使系统结构灵活、网络屏蔽及平衡各子系统处理性能不均衡等问题，在移动互联网、电子商务等领域的应用尤其广泛，阿里巴巴公布 2013 年双十一中，其分布式消息队列系统共接收消息 136 亿条，投递 244 亿次。

LSM-Tree 是这样一种树结构，它通过使用两组排序算法和日志结构，对数据在持久化存储介质上的变更进行延迟及批量处理，并在磁盘上通过一种类似于归并排序的方式高效地将更新迁移到磁盘[7]。将数据存储到磁盘上的这一过程进行批量累积和延迟处理，是 LSM-Tree 的根本思想，LSM-Tree 结构通常就包含了一系列延迟更新机制。LSM-Tree 结构特殊优点在于不仅显著提高数据写入性能，并没有减少对其他的操作，比如删除，更新甚至是那些耗时很长的查询操作比如范围查询的支持。LSM-Tree 的主要应用场景是写入频率低于读取频率或写入频率与读取频率相当的情况。

现有的消息队列系统中，消息主要存放在消息队列服务端的内存或硬盘中，存放在内存中的队列能够对消息进行实时排序并保证存取速度，但缺点是队列容量有限、消息可靠性不能得到保证，一些队列通过将消息存储在数据库中来支持消息持久化，但是这样无法保证消息的存取性能，本文在研究了多种消息队列技术的基础上，利用 LSM-Tree 高插入读取效率并且保证有序的特性，实现了一套高吞吐量、消息堆积能力强、支持多维度优先级的持久化消息队列 LSMQ。

1.2 消息队列的工作模式

符合消息队列接口协议的消息，可以通过消息队列被任意多个调用者发送和接收，从而达到分布式系统模块之间协同工作的目的。消息队列作为不同系统间沟通的中间组件，不可避免的要解决消息的持久化问题。消息队列整体框架包括网络传输层、消息处理层和消息存储结构三部分构成。其中网络管理层主要负责和消息队列生产者消费者之间网络连接，消息处理层主要负责消息的过滤、校验等操作，消息的存储结构主要负责消息在内存和磁盘上的策略。消息队列系统整体框架如下图 1.1 所示。

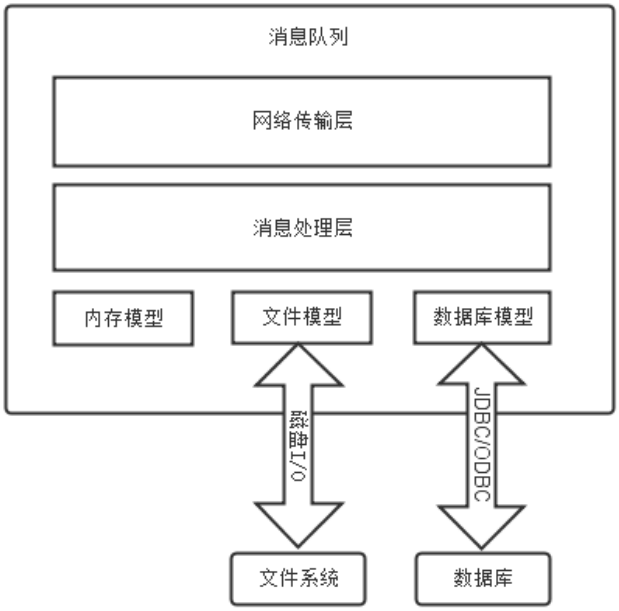

图 1.1 消息队列系统整体框架

消息队列系统工作模式分为推模式和拉模式。

推模式系统结构图如下图 1.2 所示。推模式为每一个消费者都建立一个存储容

器，符合该消费者消费需求的消息，都会被推送到该容器中。对于推模式来说，消费者无需了解生产者，生产者传递的是消息而不是生产者自身。同时，生产者可以根据应用场景，指定不同的消费者，或者选择发送消息给不同的消费者。例如在新浪微博架构中，粉丝数量小于一定数量时，微博的发布采用推模式，当用户发表了微博时，用户每个粉丝的消息队列容器中就会增加一条消息。

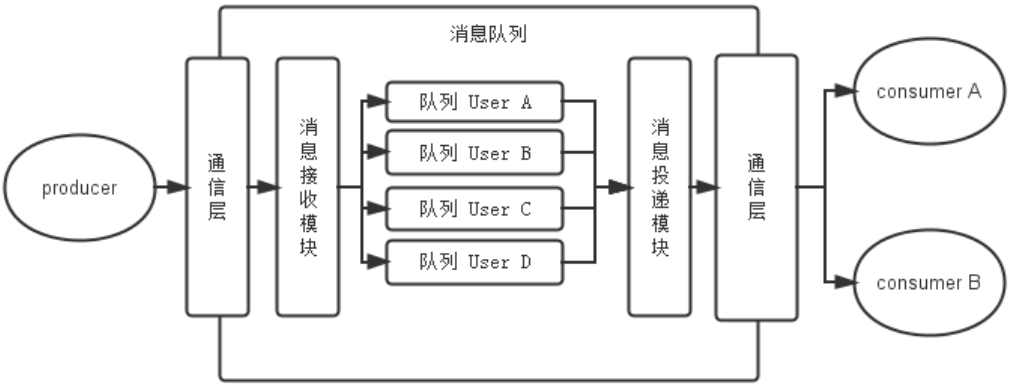

图 1.2 基于推模式消息队列的结构图

拉模式系统结构图如下图 1.3 所示。拉模式则只需要维护一个消息存储容器，但同时需要保存每一个用户的消费记录。拉模式的消费者根据设定的时间间隔，定期主动查询队列的情况，一旦发现有队列状态变化，可以实时发起请求读取未消费的消息，并记录下上次消费消息的偏移。例如在新浪微博的架构中，粉丝数量特别多时，微博的发布采用拉模式，当用户发表了微博，用户的每个粉丝在拉取消息时会发小有一条新消息，从而去根据上次消费的消息便宜，读取未消费的消息。

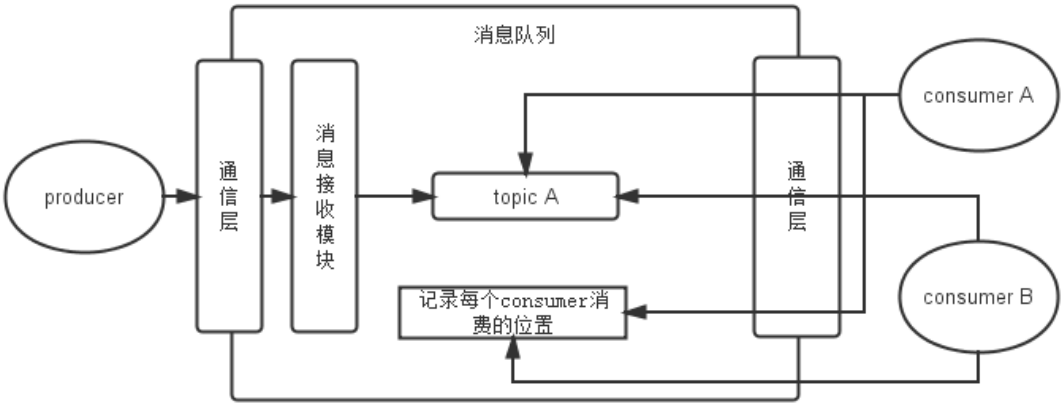

图 1.3 基于拉模式消息队列的结构图

1.3 消息队列的存储模型

消息存储模型是消息队列系统设计的关键点，目前主流的消息队列存储模型有以下几类：内存模型，文件系统模型和数据库模型。

1.3.1 内存存储模型

消息通过键值对形式进行存储，根据消息的类型不同，采取两种不同的策略：第一种针对消息队列保证消息的顺序性，内存模型需要对每个队列存储一份消息记录[8]。另一种针对消息主题模式，内存模型对每个消息主题的订阅者（subscriber）单独开辟消息记录存储区。内存模型一般顺序结构在内存中实现，由于顺序结构的结构简单，排序之后访问效率高，又在内存中进行操作，其消息读写性能是三种模型中最好的。但是内存模型有一定的局限性：使用本地内存，内存容量有限，直接影响了消息队列的消息堆积能力；内存造价高，单位数据存储成本高；内存不能持久化保存消息，有丢失数据的风险，无法满足关键行业消息可靠交付的需求。

1.3.2 磁盘文件系统存储模型

一个对外提供稳定服务的消息队列系统，必须能够提供很好的队列读写性能和挡住前端的数据洪峰。有了持久化消息队列，即使消息的接受者不同时在线，或者消息队列系统意外宕机，在消息队列系统重新启动后，接受者任然可以接收到之前没有收到的消息，消息的持久化保障了消息的可靠性传输。

消息持久化理论是很简单的，即在发送方发送消息，消息队列系统消息保存到磁盘，然后尝试发送一个消息给接收者，发送成功是从系统中删除，发送失败则继续尝试。

磁盘文件系统模型在内存模型的基础上进行改造。消息的存储策略并没有改变，同样针对基于 queue 的队列和基于 topic 的队列有不同的存储策略，但消息的存储介质由内存替换为磁盘文件系统。在保持了内存模型功能的前提下，该模型提供了对数据持久化存储的功能，为关键领域的应用提供了消息安全性的保证，但同时带来了性能上的影响，即对磁盘的访问延迟很高，降低了中间件的性能；大规模兵法操作会造成对单一磁盘资源的竞争，导致整体性能降低。

目前市面上的持久化消息队列系统代表主要有基于 kahaDB 存储引擎的 ActiveMQ 和内存消息队列 Redis。ActiveMQ 是 Apache 基金会研发的开源消息队列系统，也是当前使用最多的、对消息队列规范支持最完善的消息队列系统。ActiveMQ

支持多种语言和多种传输，也很好的支持了消息的持久性，kahaDB 是当前 ActiveMQ 的默认存储引擎，kahaDB 是基于 B+树算法的磁盘存储引擎[12]。Redis 是一个基于内存的键值对数据库，同时支持队列、哈希表等数据结构。

1.3.3 数据库模型

数据库模型是对文件系统模型的另一种改进。在系统接收到消息时，消息记录通过 JDBC/ODBC[15]方式存储到关系型数据库，这样可以达到消息持久化的目的，数据库模型虽然实现简单，但是数据库数据库的读写性能一般难以满足消息队列系统对读写性能的要求 [13]。

由于数据库系统本身设计机制的限制，大规模数据插入会触发无数行锁和表锁，严重影响消息队列系统的读写吞吐量，而且消息队列系统中消息一旦插入，基本不会修改，数据库系统设计的排序、范围查询等特性在消息队列系统中基本用不到，因此越来越多的消息队列系统设计都抛弃了数据库模型。本文通过利用 LSM-Tree 这种数据结构的特性，将磁盘模型和内存模型协调起来，以达到读和写的性能最佳平衡点。

1.4 国内外研究现状

在业务并发请求量越来越大的环境下，由于系统模块之间处理能力不平衡，请求往往会发生堵塞，比如说大量的强一致性更新请求同时到达后端关系型数据库，会导致大量的请求堆积甚至等待超时，从而影响业务的稳定性。通过使用消息队列，分布式系统可以平衡各模块之间处理能力的不平衡，模块尽可能的处理消息，处理不完的消息将堆积在队列中，这样模块不至于因为压力过大而无法对外提供服务，从而缓解系统压力。在消息队列系统中，数据的可靠性是通过持久化到磁盘的方式来保证的。每份磁盘数据在系统中保存了三个以上的备份，当有节点失效时，通过数据恢复可将副本数目控制在三个以上，并且可容忍两个节点同时故障[14]。采用这种机制能够保障数据高安全性，这种方案技术实现并不复杂，文件创建和读取不需要太多计算性能，读写效率比较高。但需要浪费两倍的额外磁盘空间，需要更多的额外的存储成本。

消息队列系统在短时间的快速发展中，发展出了多种系统原型，其中最为著名并为大家广泛接受的系统为 ActiveMQ 系统，在此基础上，知名互联网人才招聘服务公司 Linkedin 开发了自己的消息队列系统 kafaka，阿里巴巴开发了自己的分布式消息

队列系统 MetaQ[6]，奇虎 360 开发了自己的分布式消息队列系统 QBus 等等，这些系统在各自企业内部发挥着日益重要的作用，其中阿里巴巴的分布式消息队列系统 MetaQ 在 2012 年双 11 购物狂欢节中消息发送量达到 230 亿次。

ActiveMQ 是开源组织 Apache 的产品，是应用比较广泛的给予 Java 的开源消息队列系统，ActiveMQ 由网络模块、消息处理模块和存储模块组成，网络模块处理网络连接和数据传输，消息处理模块对接收消息进行过滤、校验，包括对消息的属性分析、分类存储以及是否需要事务的支持。存储模块存储消息实际存储介质。ActiveMQ 的存储模块支持各种不同的存储插件。插件由一组相关的 Java 类进行实现。通过实现 ActiveMQ 的基础接口，就可以达到构建不同类型存储模型的目的。只需要修改 ActiveMQ 的 XML 配置文件，就可以实现不同存储插件之间的切换。已有的存储插件包括内存存储插件、文件存储插件 kahaDB、数据库存储插件等。

一个 ActiveMQ 存储插件由以下几个部分组成：

(1) 队列存储组件

负责管理消息在消息队列中的存储策略，以及在持久化前对消息进行过滤和校验，以及消息的缓存策略和消息顺序性维护。

(2) Topic 存储组件

负责管理 topic 在消息队列中的存储，功能与队列存储组件相似，但是针对主题消息模式做了部分功能增强，分别存储对同一主题的消息订阅者记录。

(3) 恢复组件

恢复组件负责在 ActiveMQ 发生故障时重新启动消息队列系统，对存储在磁盘上的消息记录进行恢复，保证 ActiveMQ 恢复到故障前的状态。

1.5 主要工作和论文组织结构

本文充分研究了消息队列和 LSM-Tree 的相关技术，以国内外科研现状为背景，提出并设计实现了一种高效的基于 LSM-Tree 的持久化消息队列系统 LSMQ。该系统利用磁盘作为存储介质保证了单机存储容量，同时保证了高效的消息读写效率。本文共有五章，分别如下：

第一章介绍了消息队列系统的发展现状以及消息队列系统在分布式系统中的角色，消息队列的工作模式、存储模型，对消息队列有了整体的认识，同时指出了消息队列系统设计时需要关注的重点。

第二章通过比较 B+树和 LSM-Tree 在插入删除操作原理上的不同，分析了 LSMQ 采用 LSM-Tree 作为主要数据结构的优势。分析了 B+树在插入节点导致分裂时性能的缺陷。通过对比发现，LSM-Tree 批量合并延迟更新的的策略非常适合消息队列这种读请求少于写请求的应用场景。

第三章详细论述了 LSMQ 内存结构、磁盘结构、日志结构的设计及实现细节。LSQM 的高吞吐量主要依赖于日志结构的顺序写和内存结构的轻量快速，以及磁盘结构的有序性。这三个结构在 LSMQ 内部协调工作，从而使 LSMQ 达到高吞吐量和高消息堆积能力的目标。

第四章介绍了 LSMQ 整体架构设计，详细分析了消息读、写流程，分析了读和写操作在 LSMQ 内部数据流转过程，还介绍了 LSMQ 如何应对消息队列推模式和拉模式两种使用场景。LSMQ 同时支持推和拉模式，使得 LSMQ 有更广阔的使用场景。最后分析了 LSMQ 中采用的零拷贝技术，从而大大降低了磁盘文件到 socket 缓冲区的系统开销，提高了系统的读性能，使得系统在消息大量堆积到磁盘的情况下，读性能也不会显著降低。

第五章对比了 LSMQ 和目前主流的持久化消息队列 ActiveMQ、内存快照消息队列 Redis 和非持久化消息队列 ActiveMQ 的读写性能和消息堆积能力，讲述了系统中待完善的地方，以及对未来的工作的展望。最后是致谢及参考文献。

2 LSM-Tree

在分布式式系统设计中，主要数据结构需要根据系统的读写比例以及硬件特性来综合考虑。持久化是 LSMQ 的一个重要特性，综合考虑了单位容量成本和单机容量之后，LSMQ 选择磁盘作为持久化介质。磁盘是一个顺序读写速度远远快于随机写的设备，如果想高效的从磁盘中找到数据，势必要满足一个最重要的条件：减少寻道次数。基于磁盘的存储结构大都围绕磁盘的特性进行优化，传统消息队列在磁盘中通常使用 B+树来进行文件存储，LSMQ 采用 Log-Structure Merge Tree 来进行文件存储，下面通过对比这两种数据结构来说明选择 LSM-tree 的原因。

2.1 B+树的缺陷

机械磁盘每次读写数据由寻道阶段、旋转阶段以及存取阶段三部分时间构成。其中寻道时间是指将磁头移动到特定的磁道上的时间[18]。旋转延迟指旋转盘片直至磁头移动到对应扇区的时间．而存取时间是指磁头从当前位置读写数据的时间。因此．在传统的磁盘操作中．各种数据结构都致力于减少磁盘的寻道时间以达到提升读写性能的目的[14]。

下面举例说明，B+ 树在插入数据时，会导致节点分裂产生磁盘碎片的情况，下图 2.1 是一个完整的 B+树结构。

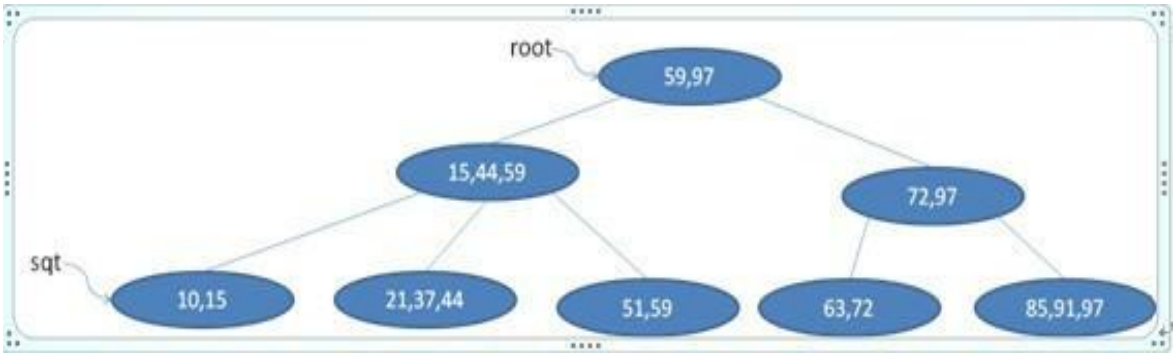

图 2.1 B+树插入前

首先查找相应的叶子节点，如下图 2.2 所示：

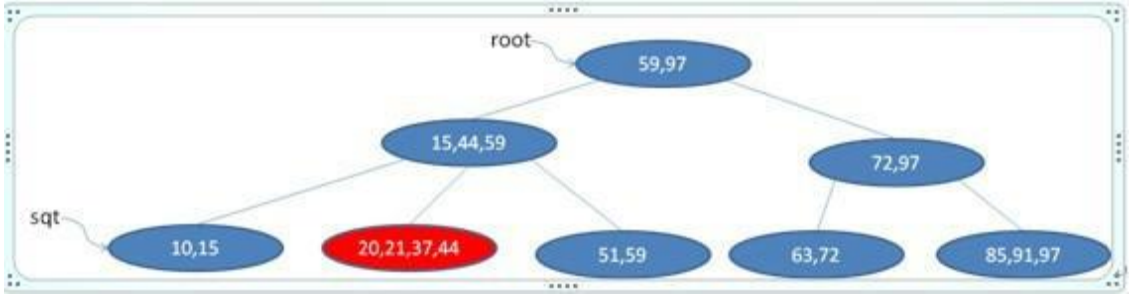

图 2.2 插入叶子节点

发现修改完叶子节点后， B+已经不满足要求，必须把叶子节点分解为[20 21]，[37 44]两个，并把 21 往父节点移，如下图 2.3：

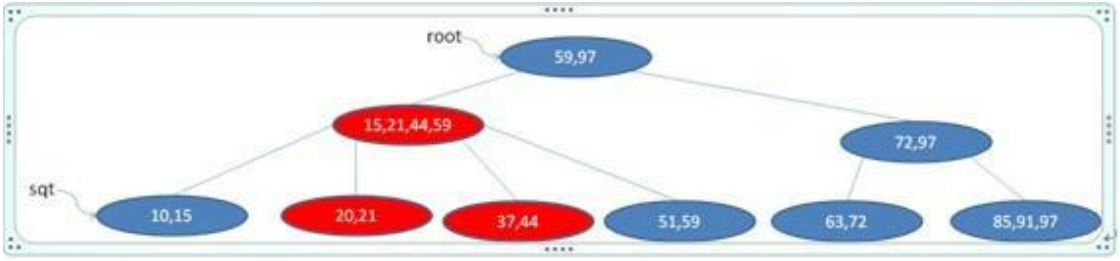

图 2.3 叶子节点分裂

发现叶子节点分裂后，父节点也不满足 B+树的要求,则需要把非叶子节点再分解为[15 21], [44 59]两个,并把 21 往其父节点移，如下图 2.4 所示：

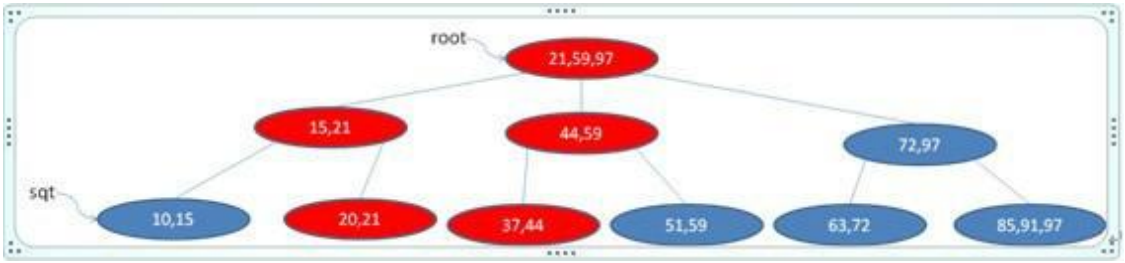

图 2.4 向父节点转移

这次整棵树都满足 B+树的要求，插入完毕。如上所示。B+树插入节点会导致新的关键字的节点产生。一旦遇到关键字个数较多的场景时，B+树的分裂会发生地越来越频繁，节点分裂会导致磁盘寻道次数急剧增加，节点更新所花费的时间将随着数据的增加增长到系统无法承担的地步[41]。

B+树节点分裂，除了影响写性能外，新产生的节点页和逻辑上相邻的节点不一定在物理存储上也是相邻节点，这时候需要额外的优化操作来保证 B+树的查找效率[38]。这就是为什么基于 B 树存储引擎数据库系统提供优化表的命令，该命令序列表

重写，清除杂物，减少文件大小，从而保证范围查询在磁盘上也是顺序进行的。

消息队列是插入频率和读取频率相当的应用，插入效率直接影响了系统的吞吐量，面对 B+树在更新频率高的环境下节点分裂而导致更新时间长、磁盘碎片化的问题，在系统设计中需要对持久化消息队列的存储模型进行大变革，主要思想是减少寻道次数以及增加数据批量读写比例，用 LSM-Tree 来代替 B+树可以很好的平衡更新造成的磁盘 IO 操作，从而提高队列写吞吐量。

2.2 LSM-Tree

LSM-Tree(Log-Structured Merge-Tree)的主要思想是利用存储器层次结构。通过延迟及批量处理更新数据.降低磁盘读写次数。其主要适用于 更新频率较大的系统。是一种牺牲零散读性能来增强延迟批量写的方案.最简单的 LSM—Tree 包含两个组件,称为 C0 树和 C1 树。其中 C0 树为内存中的索引结构,负责将更新操作在内存中快速收集。而 C1 为磁盘上的索引结构。负责对数据进行持久化存储。根据存储器层次结构原理.访问速率越高的设备.单位容量的成本越高。内存访问快,但容量小.适合对数据进行收集与封装;而磁盘访问慢,但容量大,适合对数据进行持久化存储。

LSM—Tree 的核心在于延迟性以及滚动合并。所有的更新操作都会现在内存的 C0 中进行累积,当累积量到达一定程度(如一定时间或者一定容量)后,才将其更新到磁盘上。其特点就是在进行更新时.一旦较低一层组件达到时间或者存储空间的阈值.就执行往较高一层组件的滚动合并操作。

在 LSMQ 设计，首先将接收的数据存储在完全的订单日志中的日志文件，保存后的变化，更新内存储器中的数据结构，内存中持有最近的更新以便于快速定位。当内存中积累了足够的更新，内存中的顺序存储结构填满时，就会将内存中的消息组成有序链表 flush 到磁盘，创建一个新的磁盘文件。此时所有的更新操作已经被持久化到磁盘，日志文件中对应的更新就没有意义了。

LSM-Tree 在磁盘中的组织形式，可以使用 B+树等结构，也可以使用更加简化的排序结构，例如顺序存储加索引的方式。LSMQ 在磁盘中的文件组织结构并非树形结构，而是简单的顺序存储加索引的方式，这样在满足队列系统需求的前提下，大大简化了系统设计，也提高了磁盘中排序结构合并的效率。

2.3 LSM-Tree 和 B+树的比较

在 B 树和 LSM 树本质的不同，在于他们使用现代的硬件，特别是磁盘。对于大型数据的情况下，在磁盘传输的计算瓶颈。CPU，内存和磁盘空间将增加一倍，每 18-24 个月[36]，但是每年大约有 5%来提高磁盘性能。而且消息队列是顺序型应用，

不存在范围查找，所以在这种场景下，LSM-Tree 可以更好发挥自己的优势。

B 树和 LSM 树比较发现，如果没有太多的节点更新，B 树可以很好地工作，因为 B+树的非叶子节点索引会进行比较繁重的优化来保证较低的读取时间[39][40]。随着越来越多的数据添加到 B 树，磁盘上的数据将变得支离破碎，最终，更新和删除会产生大量的随机 IO 操作。LSM-Tree 的数据结构决定了对机械存储器的操作都是顺序的，这样更好地利用廉价的机械存储器将系统的存储规模扩展到更大的水平。因为使用日志文件和内存存储结构把随机写操作转化为顺序写，也可以保证数据的插入速率不会随着数据量的堆积而下降。读操作与写操作是独立的，读操作在消息没有产生堆积的情况下，在内存中就可以完成，在消息堆积之后，读操作需要访问顺序的磁盘数据，通过二分查找和磁盘文件索引，可以很好的保证系统的读性能。

2.4 本章小结

本章首先介绍了 B+树在磁盘存储结构中的应用，然后通过实例分析了 B+树在插入操作产生分裂而造成的系统额外 IO 操作的案例。通过分析得出结论，B+树在组织大数据方面，随着数据的增长，B+树插入操作系统开销会增长到系统无法承担的地步。

介绍了 LSM-Tree 这种能显著提高系统写性能但不降低系统读性能的结构，同时结合消息队列的应用场景，发现消息队列中消息的写入操作时按照消息写入先后排序的顺序操作，特别适合 LSM-Tree 这种排序结构。消息的读取操作当没有堆积到磁盘时，只涉及到一次内存 IO，当消息堆积到磁盘时，由于磁盘上的数据是有序的，可以通过二分查找加快查找效率，再加上磁盘文件的索引，系统的读性能也不会显著降低。从而得出了 LSM-Tree 在持久化消息队列系统中应用的理论可行性。

3 LSM 存储结构

3.1 总体

上一章对 LSM-Tree 在消息队列系统中应用的理论可行性进行验证后，开始设计基于 LSM-Tree 的存储结构 ，LSM-Tree 的存储结构包含三个主要部分，内存结构、磁盘结构、日志结构，一条消息的插入操作，首先会更新日志结构，然后更新内存结构，等到内存结构堆积到设定的阈值之后，批量将内存结构顺序写到磁盘排序结构中，下面分别就着三部分讲解 LSMQ 的设计。

3.2 内存结构

LSMQ 在收到消息的写请求时，首先将消息以顺序的方式被追加到 log 文件中，只有 log 文件成功保存了变更，才去更新一个内存中的数据结构。内存中的排序结构的效率和空间利用关系到整个系统的单位时间消息读写量，由此可见内存中的数据结构时整个系统中性能的关键影响因素。LSMQ 的内存数据结构提供了将消息数据写入、删除以及按照消息 ID 读取消息的接口。需要注意的是，LSMQ 的内存数据结构中对消息是按照消息 ID 排序的，在系统插入新的消息时，首先会去请求队列元数据，获取新的自增 ID 为消息 ID，因此 LSMQ 在消息插入过程中，对内存数据结构来说是有序插入，根据这一特点，选择了一种在插入前基本有序的情况下，效率高、内存利用率高的数据结构—跳表[19]。

LSMQ 的核心内存数据结构是是一种轻量、高效的排序结构——跳表。跳表的结构如下图 3.1 所示。

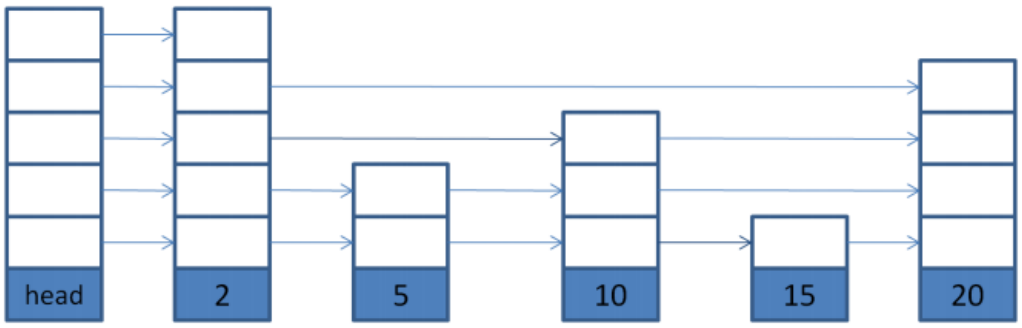

图 3.1 跳表

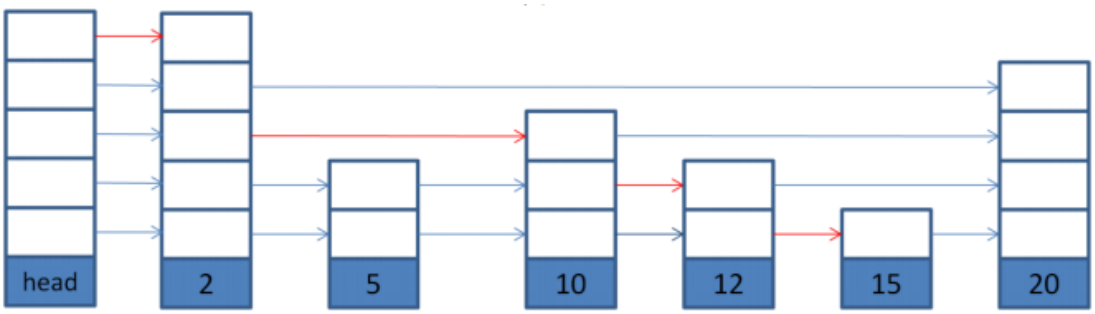

图 3.2 跳表插入和查找示意图

图 3.2 是跳表插入和查找示意图。

跳表的插入操作先找到不小于该 key 的节点，然后随机产生新节点的高度，再对各个高度的链表做插入操作即可；跳表的删除操作先找到节点，并对其所在各个高度的链表做相应的更新。跳表的读取操作先从根节点开始，找到不小于 key 的节点。高度从高向低，知道找到节点或者到达链表结尾。上图 3.2 中红色路径即是对 15 的查找路径。

跳表是从链表演变过来的一种数据结构，是快速随机性查找数据结构的一个很好的解决放哪。由于是高阶的链表，其效率查找效率非常高 [21]。要查找一个目标元素，从头元素和层级最高的列表开始，并沿着每个路径搜索，一直到达小于或着等于匹配元素[22]。

表 3.1 跳表数据格式

User-key	Sequence Number（with ValueType）

跳表中的数据结构如上表 3.1 所示，全局来看，LSMQ 中的跳表是所有节点的排序链表，考虑搭配操作效率，为这个链表再添加不同维度的辅助链表，跳表不仅是一个性能出众实现简单的排序数据结构，而且普通的排序树相比在插入数据的时候可以避免叶子节点分裂带来的开销，所以可以达到很高的写入效率，LSMQ 整体而言是个高写入系统，跳表在其中起到了很重要的作用。

首先跳表的节点保存的是消息 ID 与消息体组成的数据，每一条消息都会有全局的自增 ID，再加上消息操作类型，全局消息 ID 大的消息排序值大，这样保证不会有相同权值的消息出现，也就保证了不会有跳表节点更新的情况。其次消息的 delete 操作等同于 put 操作，LSM-Tree 在内存和磁盘中的结构都是排序结构，如果相同的键值被 put 两次，后一次操作逻辑上会覆盖前一次操作。基于以上两个特点，跳表的

操作不需要任何锁。

跳表的写入操作，先找到不小于该 Key 的 node，随机产生新 node 的高度，再对各个高度的链表做插入操作[23]。跳表的删除，先找到 node，并对其在各个高度的链表做更新。跳表的读取操作，先找到不小于 key 的 node，从根节点开始，高度从高到低，与 node 的 key 比较，知道找到或者到达链表尾。

3.3 磁盘结构

上一节分析了内存结构，LSMQ 在内存中的结构是一个轻量级的排序结构-跳表，当内存中的结构容量达到阈值时，LSMQ 需要将内存中的数据批量转存到磁盘中，这样可以将批量的随机写操作，堆积成顺序写操作，在保证写成功的前提下，极大的提高了消息队列的写入性能[9]。保证了写入速度，为了尽可能的提高，数据在磁盘中必须提供索引，磁盘中索引数据结构很容易想到 B+树，但 B+树会产生大量的随机 IO，最终写入速度会受制于磁盘寻道时间。LSM-tree 本质上就是通过延迟写入磁盘和顺序更新日志等策略，显著的提高了写效率，但没有明显降低读效率。所以我们设计的 LSM-Tree 在磁盘中并没有采用单一的 B+树结构[42]。

刚生成的磁盘文件可以看作是内存结构的快照，磁盘内的文件数据范围可能出现重叠，一旦出现重叠，就意味着读取操作时，需要读取多个磁盘文件来查找对应的数据，因此采用双层磁盘文件结构，通过定期将第 0 层的文件合并为不可能重复的第一层的磁盘文件，来减少查找操作时需要读取的文件个数。层次化磁盘文件结构是 LSM-Tree 中最重要的结构，通过将数据文件进行分层管理以达到读写均衡的作用[16]，其结构图如下图 3.2 所示：

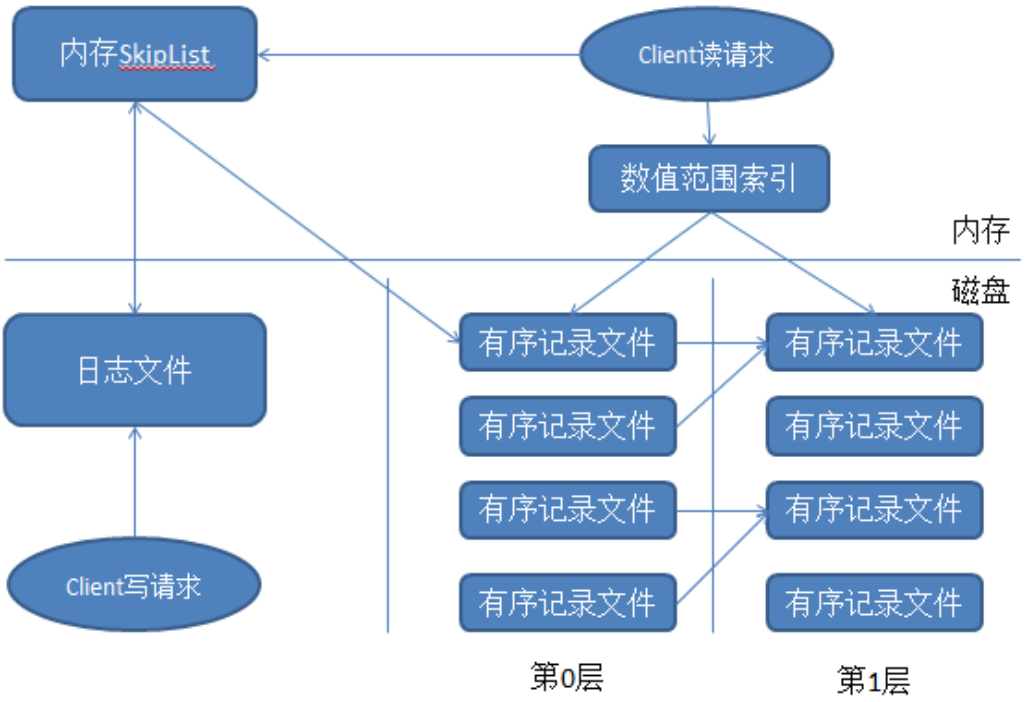

图 3.3 层次化磁盘文件结构

LSMQ 把磁盘上的很大的排序结构分割成很多个排序值可能重叠的小排序结构，在内存中构建一个有序小队列，随着小队列越来越大，内存的小队列会 flush 到磁盘上的小队列，磁盘上的小队列会定期合并成大队列。读操作时，磁盘上每个队列内部数据是有序的，并且每个队列都有内部索引来加快查找[10]。

单个磁盘文件的结构如下图 3.3 所示：

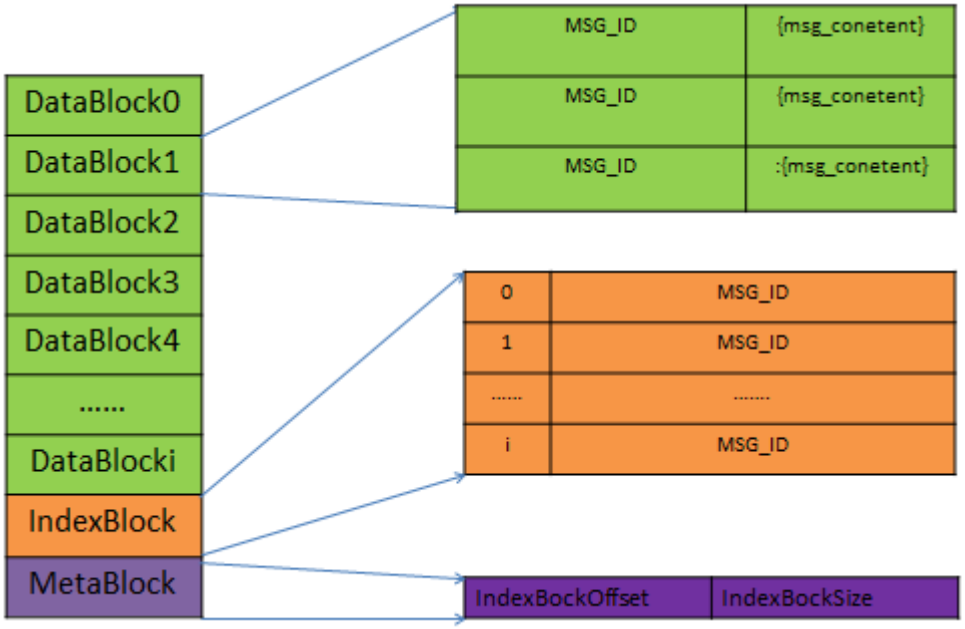

图 3.4 LSMQ 磁盘文件结构

磁盘索引文件结构如下图 3.4 所示:

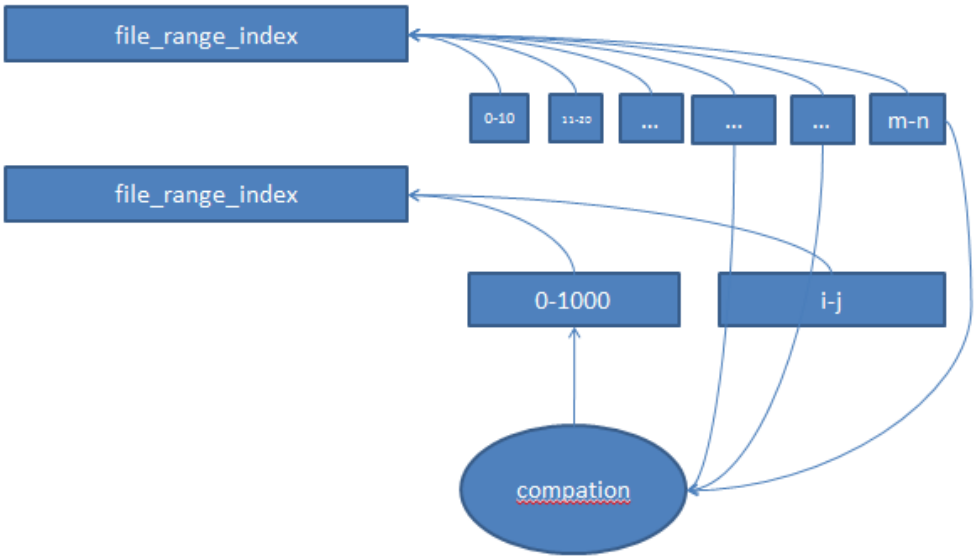

图 3.5 磁盘索引文件结构

LSMQ 将磁盘文件划分为固定大小的物理存储块，上图展示了 LSMQ 磁盘文件的静态存储结构。从大的方面来说，可分为数据存储和数据管理领域的文件，身份信息的数据存储区存储实际数据管理领域，提供一些指标管理数据，目的是更快速

记录[32][33]容易找到。两个区域的块中的文件的基础上，在实际的数据存储多个数据存储管理前，在数据管理领域。

身份信息的数据存储区是根据 ID 消息排序，每个记录的数据指标是基于区域的块的索引信息，每个索引信息包括三个方面的内容，第一场记录黄色部分大于消息 ID 等于在数据块的最大消息 ID，第二战场封锁了整个数据文件中的初始位置，第三场表示数据块的大小。在内部结构的文件数据块结尾如紫色，indexblockoffset 指出 indexblock 开始的位置和大小，这个字段可以理解为一个指数，它是建立在以正确读取的索引值。

表格 3-2 磁盘数据组织格式

shared_bytes (varint)	unshared_bytes (varint)	value_bytes (varint)	unshared_key_data (unshared_bytes)	value_data (value_bytes)

磁盘中数据的组织格式如上表 3-2 所示，磁盘结构的写入操作时，不会对 key 执行排序的逻辑，因为磁盘结构是由内存排序结构直接生成的，key 的顺序已经在上一层得到了保证。如果是一个新的 block 的开始，计算出上一个 block 的结束偏移，在 IndexBlockOffset 中追加索引数据。磁盘结构读取操作时，首先需要传入磁盘文件的大小，读取文件末尾的索引数据，解析索引数据，根据传入的 key 定位到对应的 block，再遍历 block 内的记录，获得 key 对应的 value。

磁盘文件都是由内存文件直接生成的，内存文件的有序性保证了磁盘文件的有序性，有序的磁盘文件可以应用二分查找快速查找磁盘记录[10]。不同磁盘文件中的记录可能有重叠，最优的情况是，一个可以只出现在一个文件中。因此 LSMQ 中有一个后台进程专门用于磁盘文件的整理操作，后台进程会找出有重叠记录的文件进行合并操作，再生成新的磁盘文件，这个后台进程可以认为是一个优化进程，不停的优化系统对磁盘文件的访问。

3.4 日志结构

日志结构是是由 John K. Ousterhout 和 Fred Douglis 在 1988 年提出的为文件系统而设计的结构[1]。传统的文件系统希望通过高效的数据结构和索引结构来减少磁盘 IO 以达到提高文件系统效率的目的，减少磁盘 IO 的方式往往都是能够让数据连续的分布，减少碎片，增加顺序磁盘操作。但日志结构恰好相反，它用数据流的方式将数据用追加写的方式写到存储介质中，这样不存在修改操作，所有的数据都以某一

版本的方式写入了存储介质，所有的随机写操作都变成了顺序写操作，而读需要经过选择以后读取数据的最新版本。

LSM-tree 的主要思想就是在内存数据未堆积满之前，可以直接将更新保存在日志和内存中，这主要是为了减少频繁的对磁盘中的小文件进行更新。但是在内存中的数据是不稳定的，当服务器突然宕机或停电时这部分数据可能丢失。这个问题的解决方案通常是先在日志文件中保存操作记录，即每次对内存数据结构进行更新之前都先将操作记录更新在日志文件中，只有当日志文件和磁盘文件都更新成功时才通知用户更新成功。服务端就可以根据内存数据的变化是否达到指标决定是否同步写到磁盘。最终，当内存存储结构堆满后，或者经过了一段时间，数据才异步的顺序写到磁盘文件中。

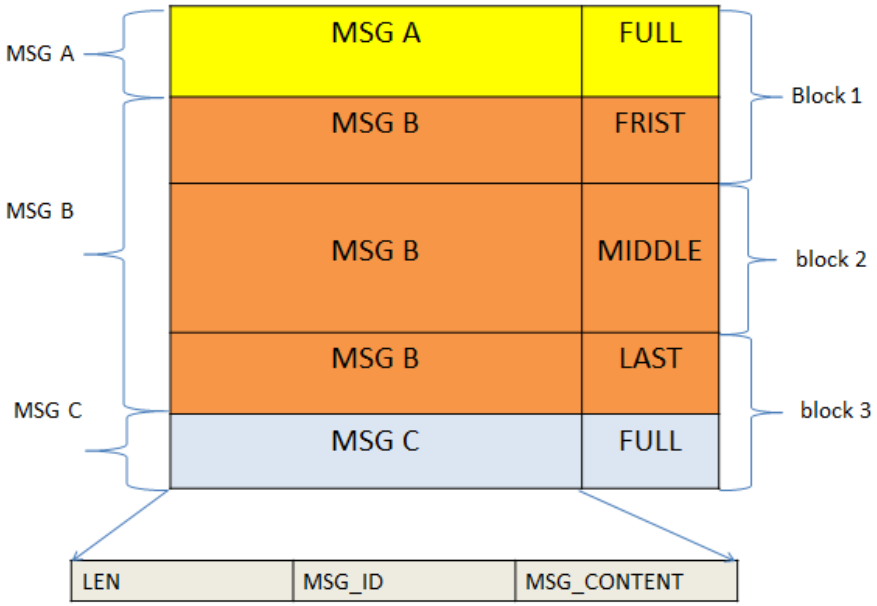

图 3.6 LSM-Tree 日志文件结构

表 3.1 记录与物理 block 对应关系表

FULL	当前记录完整的存在物理 block 中
FIRST	当前记录起始部分存在物理 block 中
MIDDLE	当前记录中间一段存在物理 block 中
LAST	当前记录剩下的部分存在物理 block 中

Log 文件的具体物理和逻辑布局如上图 3.5 所示。一个日志文件 LSMQ 会把它切成一块一块的 32K，阅读的基本单元，由上述显示日志文件由 3 块，MSG 部分的 LEN 部分记载了一次消息写入的长度，MSG_ID 为消息全局唯一 ID，MSG_CONTENT 为消息内容，消息内容最大不超过 90K。在信息存储每个块，一些存储消息的类型，这里的类型是指每个记录的日志文件的逻辑结构和物理结构之间的关系的块。记录与 block 对应关系如上表 3.1 所示，FULL 类型表示该 block 中的 MSG 数据是完整的，FIRST 类型表示该 block 中的 MSG 数据只是第一部分，MIDDLE 表示该 block 中的 MSG 数据只是中间一部分，LAST 表示该 block 中的 MSG 数据是最后一部分。

表 3.2 日志文件数据组织格式

length (uint16)	type (uint8)	type (uint8)

日志文件数据格式如上表 3.2 所示，length 是记录内保存的 data 长度，type 记录的是数据类型是插入还是删除。虽然在日志文件中的数据块的组织形式。但在写日志时，出于一致性的考虑，并没有按 block 为单位来批量写，而是每次更新都对 log 文件进行 IO 操作。Log 文件在读取时，以 block 为单位批量读取。Log 文件的写入是顺序追加写入，读取时系统启动的时候一次性发生，因此 log 文件的读写都不会是性能瓶颈。

3.5 本章小结

本章首先介绍了 LSMQ 最重要的三种数据组织结构，内存结构、磁盘结构和日志结构。

LSM-Tree 的优势在于其能推迟写回硬盘的时间，进而达到批量地插入数据的目的。这种优势是通过内存结构、磁盘结构、日志结构协同工作来保证的，日志结构时顺序写结构，每一条数据的操作都会产生一次 IO 操作，这样保证了写操作的原子性，同时也保证了系统在任何时候出现故障，日志中都记录了完整的已经写成功的数据，虽然这些数据可能还没有写到磁盘中，但是下次系统恢复的时候，可以通过日志发现这些丢失的数据，再通过一定的策略恢复这些丢失的数据到磁盘中。日志结构时顺序写操作，因此也不会成为性能瓶颈。磁盘结构是基于层级的顺序磁盘文件，每个磁盘文件都是由内存文件直接生成的，内存文件的有序性，保证了磁盘文件的有序性，同时磁盘文件结构的有序性保证了系统对磁盘数据的查找性能。内存结构时轻量级的跳表结构，跳表是可以快速排序结构，最好的情况是插入的数据基本有序，这样每次插入操作只需要进行一次查找就可以找到节点的位置。由于跳表

时一个有索引的链表结构，最坏的情况也只是在两个链表索引节点之间遍历所有节点，采用跳表保证了系统高效的内存操作和内存利用率。

4 整体结构设计

4.1 设计目标

LSMQ 持久化消息队列系统的设计重点在于：1.保障系统的高吞吐率，LSMQ 的应用场景是海量消息的读写访问，因此保证系统对分布式系统各个模块提供高吞吐率是基本需求；2.保障系统的单机容量，在保证读写性能的前提下，利用磁盘优势，保证单机容量；3.保障系统的高可用性，系统可以应对各种物理故障而不丢数据。

4.2 LSMQ 架构设计

LSMQ 整体架构主要由 topic 管理模块、LSM-tree 模块、网络连接模块、消息消费 offset 管理模。LSMQ 整体架构图如下图 4.1 所示。

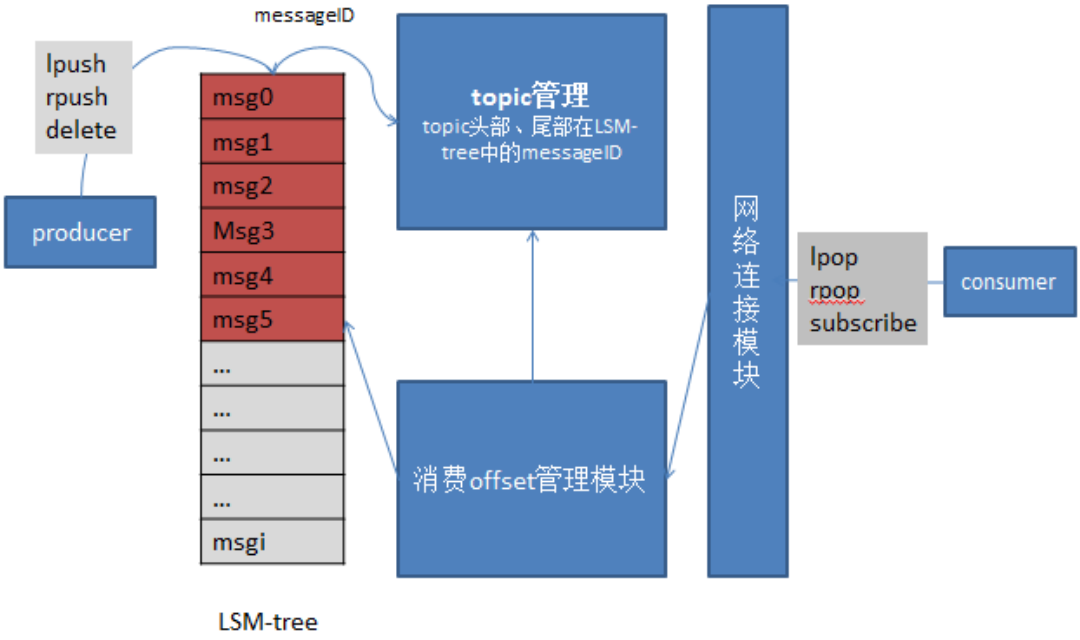

图 4.1 LSMQ 整体结构

LSM-tree 模块是系统最主要的模块，包括内存、磁盘、日志三个部分，当有新消息写入时，先根据消息指定的 topic，找到该 topic 对应的队列头部消息 ID，然后顺序生成一个新的消息 ID，作为新的消息 ID 插入到 LSM-tree 模块中的日志和内存中，随即返回给用户消息发送成功。

topic 管理模块管理了每个 topic 的队列长度、队列头部消息 ID，每次队列的写入和读取操作都会对 topic 模块进行访问。topic 管理模块是一个队列名称到队列头部节点的映射关系，topic 管理模块数据量非常少，可以完全放在内存中。这样保证了模块频繁的访问不会成为性能瓶颈。

消费 offset 管理模块管理了每一个以拉模式访问队列的用户的 offset。如果队列以拉模式访问，那么新增一个消费者时，会在消费 offset 管理模块中添加一个用户 ID 和队列名称到最近一次消费的消息 ID 的映射。每一个以拉模式访问队列的用户自己管理着一个定时器，定时器定期向队列发起读请求，队列首先查看消费 offset 管理模块，得到该用户最新消费的消息 ID，再和 topic 管理模块中队列头部节点比对，如果发现该用户有新消息，则读取未消费的消息返回给用户。

网络连接模块维护了的系统和消息生产者、消费者之间的 socket 句柄。LSMQ 采用 epoll 方式来管理连接句柄，epoll 是 linux 内核为处理大批 socket 连接句柄而改进的 poll，是 linux 下多路复用 IO select/poll 的增强版，epoll 能显著减少程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率，epoll 不是迫使开发者每次等待事件发生之前都必须重新准备要被监听的句柄集合，而且获取事件时无需遍历整个被侦听的文件描述符集合，只要遍历是内核事件异步唤醒并加入就绪队列描述符集在线[24]。epoll 在系统资源调用等方面显著优于 poll 和 select，LSMQ 的网络连接模块采用了非阻塞网络 IO 和 epoll 结合的方式来管理网络连接句柄，大大提高了系统能承载的用户连接数量，也降低了网络连接模块的系统资源占用。

消息队列的主要功能是平衡各模块之间处理能力的不均衡，当前端的数据洪峰到来时，能够留住足够多的消息来保证后端系统的稳定性，这就要求消息队列系统有一定的消息持久化能力，消息出现洪峰需要持久化时主要有两种情况：

(1) 消息堆积在内存中，一旦超过内存限制，普通的内存消息队列只能选择根据一定的策略丢弃消息，这样的系统的持久化能力取决于内存大小，当大量消息需要持久化时性能会急剧下降，因为内存已满，新的写操作将带来直接的随机磁盘写操作。

(2) 消息堆积在磁盘中，磁盘队列系统在设计的时候，需要考虑的一个重要问题就是，消息读取时，如果不能在内存中命中，要不可避免的访问磁盘，会产生大量的 IO 操作，读取 IO 吞吐量直接决定着系统消息的积累量，也有信息积累问题，如果写吞吐量将受到影响。LSMQ 在架构设计上保证了，消息在内存中达到一定阈值后顺序批量写入到磁盘中，因此 LSMQ 的堆积能力可以不受内存大小的限制。另外，由于

LSMQ 通过磁盘索引、内存缓存等策略保证了读操作不会受到磁盘消息堆积的影响[35]。

LSMQ 系统中有一个后台进程，负责将内存中的跳表转储为磁盘文件，以及均衡整个磁盘中两层文件结构

4.3 消息写入

当一个写消息请求到达时，首先写入日志结构中，并且日志结构是实时写入到磁盘的，这样保证了系统的一致性[25]。然后写入到内存的跳表中，内存的数据积累到一定阈值后通过转储的方式，写入磁盘。系统的插入消息流程图如下图 4.2 所示。

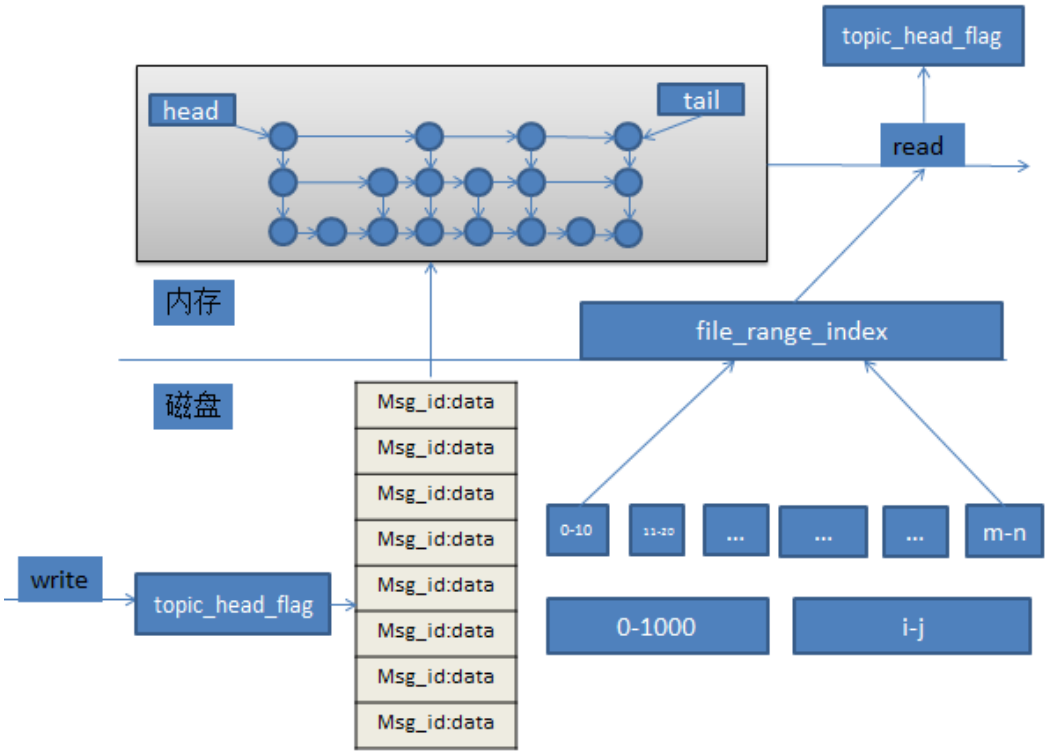

图 4.2 LSMQ 数据流动

对于消息的插入来说，完成插入操作包含三个具体步骤：首先从 topic 管理模块中查找该 topic 对应队列头的消息 ID，将该消息 ID 加 1 作为新的消息 id；然后记录的消息在写的方式添加到前面介绍的日志文件的末端的秩序，尽管磁盘的读写操作，但文件订购额外的效率很高，所以不会导致较低的写入速度;；第三个步骤，是如果写入 log 文件成功，那么将这条记录插入到内存的跳跃表中，由于该消息的 ID 对跳跃表来说是最大的，所以跳跃表插入一条新记录过程非常简单，只经过一次查找就

能找到合适的位置，然后修改相应的链接指针即可。完成这三个步骤，写入消息就算完成了，所以写入一条消息操作涉及到一次磁盘文件追加写和一次内存跳跃表的插入操作，而且跳跃表插入记录是最大的，查找过程非常简单，这是 LSMQ 消息插入速度快的根本原因。

根据负载的特点，刚写入的数据任然是热数据[26]，所以刚写入的数据放在内存中可以保证读性能的有事，随着内存数据堆积，会有越来越多的数据堆积到磁盘中刚，磁盘上有越来越多的由内存转储而来的文件，后台的线程会选择数据范围有重复的文件进行合并，如此下去，保证万一读操作不可避免的要访问磁盘，也只需要定位尽可能少的磁盘文件就可以找出记录所在。

由于 LSM-Tree 批量合并随机写为顺序写的策略，LSMQ 的写操作不会成为瓶颈，当消息少量堆积在内存中时，一个写操作只涉及到一系列的日志文件的写操作和存储器的写操作，内存跳表结构的顺序写操作，只需要一个搜索可以找到要插入的节点，效率很高。。当消息大量堆积到内存并且开始堆积草磁盘时，写操作依然是一次顺序日志文件追加和一次内存操作，只有当内存堆积达到阈值时，才会触发一次写磁盘操作，而且写磁盘操作时不需要对数据进行任何排序处理，内存中就保证了数据有序，这样写磁盘操作也就是一次顺序 IO 操作，因此当消息大量堆积到磁盘时，系统的写性能也不会出现瓶颈。

4.4 消息读取

读流程是一个从新到旧逐级查找的过程，最新的数据在内存中，其次是在磁盘第 0 层的文件中，最后是磁盘第 1 层的文件中，所以首先要查找内存中的跳表，然后通过内存中第 0 层文件数据范围索引来定位到第 0 层文件，最后是磁盘第 1 层文件。对于数据的定位，可以通过读取该文件的索引部分，然后在该索引中定位对应的数据块，如果查不到，旧到下一层去继续查找，消息读取操作的流程如下图 4.3 所示：

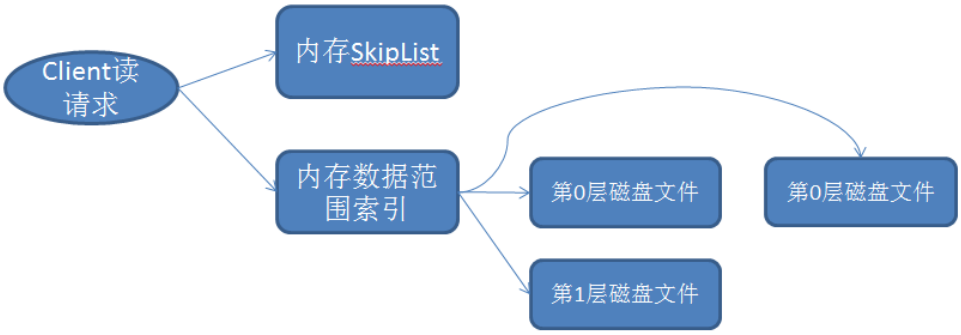

图 4.3 消息读操作流程图

LSM-Tree 是为读取和写入操作比例相当甚至写入操作比例大于读取操作比例而设计的，LSM-Tree 能在不显著牺牲读取效率的基础上，显著提升写入效率。假设应用提交了对某一个 topic 队列的 pop 请求。LSMQ 首先请求 topic 管理模块，得到该 topic 当前头部的消息 ID，然后根据该消息 ID 去内存中跳跃表中去查找效应记录，如果读到记录就直接返回，若没有读到，那么只能去磁盘文件中读取，先从较第一级磁盘文件中检索，如果找到直接返回；如果没有找到再从第二季磁盘文件中检索，如果找到直接返回，如果没找到，说明消息 ID 不存在。

这里要特别说明的是，队列应用读操作主要是 pop 操作，也就是查找 LSM-Tree 中最老的记录，如果出现消息堆积，最老的记录出现在第二级的文件中，如果没有出现大量消息堆积，最老的记录会出现在第一级文件甚至内存中，而这些操作通过磁盘文件索引，都能一次定位到所在的文件，并读取文件中的索引部分，快速定位到数据[27]。

对于 LSMQ 的读操作，分为两步。首先查找内存跳表，如果内存跳表中存在，直接返回给用户，若内存中没有找到，则对应的消息已经持久化到磁盘，这时需要借助加载到内存中的磁盘文件范围索引，找到对应的消息持久化到哪一个磁盘文件中，找到对应的磁盘文件后，由于磁盘文件内部记录是有序的，和文件位置偏移对应的索引的索引键值对应的范围，因此必须首先加载磁盘文件索引，然后在索引中搜索，找到相应的数据块，然后遍历块查找相应的记录。整个过程只需要两次文件 IO 操作，一次是加载文件索引，一次是加载记录所在的数据块。因此 LSMQ 在大量消息堆积到磁盘时，读性能不会出现显著下降。

4.5 磁盘文件合并策略

系统中有一个后台进程，负责将内存跳表结构转储为磁盘文件，以及均衡第 0 层和第 1 层的数据文件，合并进程会优先将已经写满的内存跳表持久化为第 0 层的磁盘文件，以及选取第 0 层的数据文件合并为第一层的数据文件，随着数据不断写入和合并操作的进行，第 0 层的数据文件不断向第 1 层数据文件迁移，第 0 层的数据文件是有跳表直接持久化得到，所以记录可能重叠，而第一层数据文件都是合并得到的，记录不会重叠。

数据合并可以减少系统维护的磁盘文件个数。随着磁盘文件堆积越来越多，文件个数也越来越多，内存中维护的数据范围索引占用空间越来越到，数据合并可以有效降低内存中数据范围索引所占空间，提高内存利用率[34]。

4.6 LSMQ 的推模式

LSMQ 支持消息的推模式，即某一个个 topic 的队列中由新消息到来时，队列系统可以主动将消息在最短的时间内推送给订阅者。使用推模式派发消息，消息的消费者无需了解消息的生产者，只需要发起订阅操作，然后等待消息到来即可。

Offset 管理模块是为了队列的推模式而设计的，本质上 LSMQ 是不支持推模式的，因为推模式需要在每一条消息写入的时候通知消费者，这样大大降低了系统的吞吐量。LSMQ 通过 offset 管理模块，管理了每一个消费者上一次消费的消息 ID，然后该模块定期发起读请求，查找 topic 模块中队列最新的 ID 和自己管理的 ID 是否一样，如果不一样，说明队列中有新消息，读取这一部分新消息，推送给队列消费者，通过这种方式可以满足实时性要求不是特别高的场景下推模式的需求。

LSMQ 中消息的推模式是通过 offset 管理模块与用户建立 socket 长连接，维护用户上次消费消息 offset 来实现的。用户消费 offset 管理模块定期向 topic 模块发起请求，查看是否有新消息到来，如果有，取出新消息，推送给用户。

推模式是有需要时及时的将用户感兴趣的消息推送到客户端。推模式的主要优点是能够保证消息到达的及时性，用户是被动的接收消息。但是，在随后实际应用中，但推模式当推送目标量很大时，推送成本会急剧增加，系统推送效率也会降低，因此，推模式适合于推送目标数量少、对及时性要求低的场景。

4.7 LSMQ 的拉模式

LSMQ 同时支持消息的拉模式，即用户可以主动的发起读消息的请求，根据用户

发送的 offset，LSMQ 可以从队列头部开始遍历直到 offset 之后的消息返回给用户，拉模式需要用户自己手动维护上次请求消息最新的 offset 以便于下次发起读请求时发送给 LSMQ。

拉模式一般是用户主动发起查询请求，查询是否有满足条件的消息到来，比如用户从浏览器发出请求，由浏览器向队列系统发起请求[11]。拉模式的有点在于系统只需要维护一个队列，以及应对各种 offset 的请求，必须要为每一个消费者维护一个消息容器，而且并非每一个用户都需要消费队列中的消息，需要消费的用户就发起消费请求，不需要消费的用户也不会产生空间上的浪费。这种模式适合于目标消费者数量巨大的应用场景。

4.8 零拷贝技术

LSMQ 在对磁盘文件读取过程中，使用了零拷贝技术[20]。一次典型的读文件 IO 操作过程如下表 4.1 所示：

表 4.1 一次典型 IO 操作

```
read(file, tmp_buf, len);
write(socket, tmp_buf, len);
```

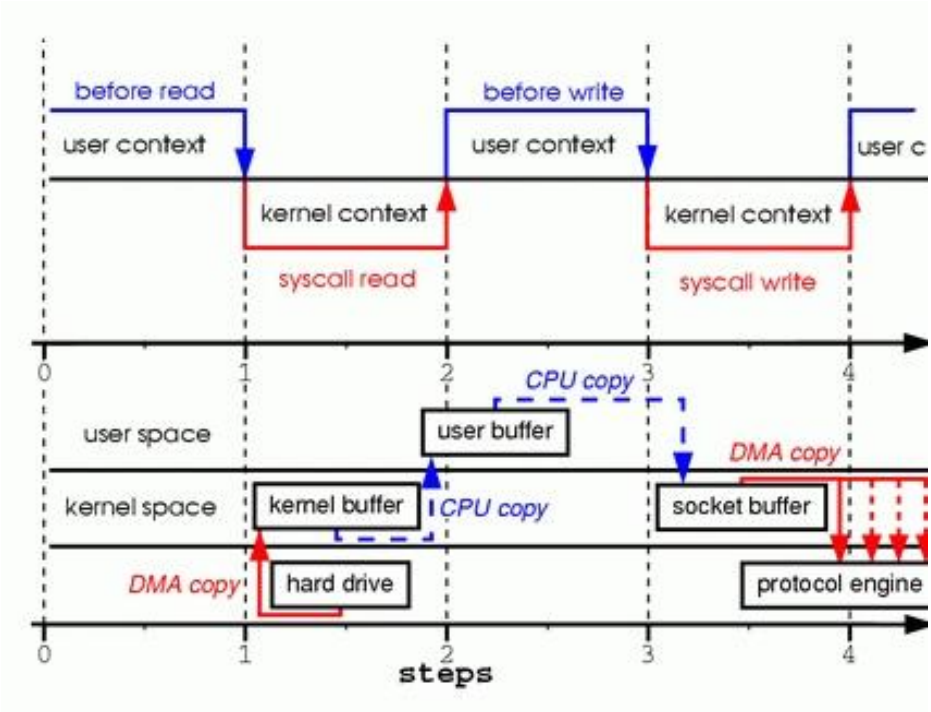

图 4.4 一次典型 IO 操作流程图

IO 操作流程图如图 4.4 所示，用户上下文切换从用户空间到内核空间通过 read 系统调用完成后，在 DMA 引擎的数据的第一个副本，DMA 引擎从磁盘读文件数据然后保存在内核缓冲区中[28]。第二次拷贝发生在数据从内核缓冲区复制到用户缓冲区，至此 read 系统调用结束。read 系统调用结束又会发生一次上下文从内核空间到用户空间的切换，通过两次上下文切换和两次数据拷贝，数据到达用户缓冲区供后续处理。write 系统调用又会引起一次上下文从用户空间到内核空间的切换，第三次拷贝发生在将数据从用户缓冲区拷贝到内核缓冲区，这一次数据被拷贝到一个特殊的和 socket 句柄相关的缓冲区中。当写系统调用返回时，会导致第四的上下文切换，并在同一时间，从套接字缓冲区的协议引擎[29]第四复制的数据中出现。

从上面的分析可以看出，一次简单的 read 和 write 系统调用会引起 4 次数据拷贝操作和 4 次上下文切换操作。如果数据到了用户缓冲区之后，没有进行任何操作直接发送到 socket 缓冲区，那么这一过程中一些数据拷贝和上下文切换操作时完全可以避免的，这就是所谓的零拷贝技术，零拷贝技术主要有 mmap 和 sendfile 两种方式，mmap 方式实现零拷贝如下表 4.2 所示。

表 4.2 mmap 方式零拷贝

```
tmp_buf = mmap(file, len);
write(socket, tmp_buf, len);
```

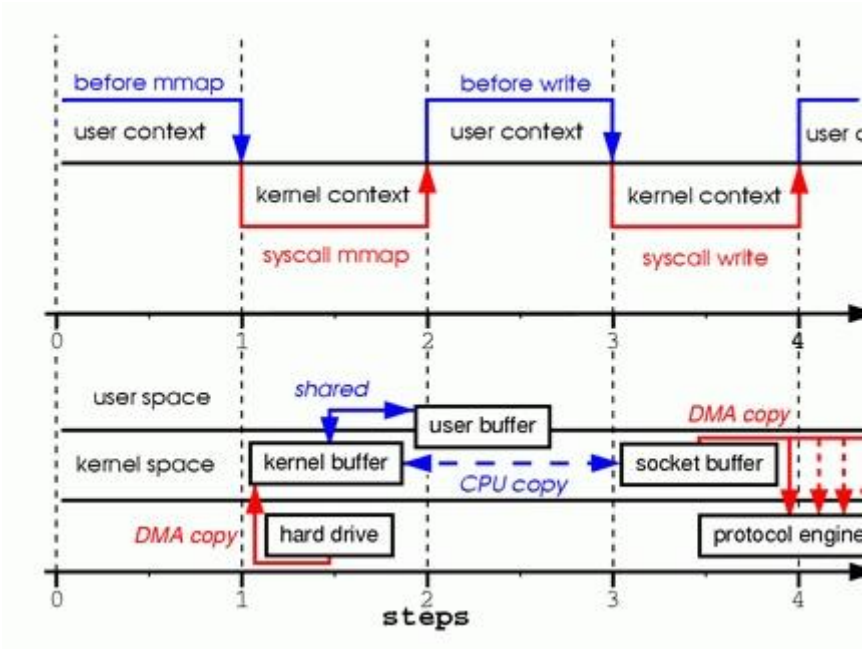

图 4.5 mmap 方式零拷贝流程图

mmap 方式零拷贝流程如上图 4.5 所示，首先 mmap 系统调用会引起上下文从用户空间到内核空间的切换以及一次数据从 DMA 引擎到内核缓冲区的拷贝操作，这时一个特殊的可以被用户进程共享的缓冲区，需要注意的是 mmap 系统调用不会引起任何从内核缓冲区到用户缓冲区的数据拷贝[30]。write 系统调用在引起上下文切换的同时将数据直接从共享缓冲区拷贝到 socket 缓冲区，当 write 调用返回时，上下文切换到用户空间，从套接字缓冲区的数据复制到网络协议引擎。

从上面的分析可以看出，mmap 的方式需要 4 次上下文切换和 3 次数据拷贝操作，其中将两次数据从用户空间到内核空间的复制减少为 1 次数据从内核共享缓冲区到套接字缓冲区的拷贝，这在大量磁盘 IO 操作时可以显著提高系统性能。但 mmap 方式的缺点在于，如果 write 系统调用之前有其他进程操作了该文件，会直接引起 write 系统调用中断，此时 mmap 所映射的地址已经发生了变化，会直接引起内存越界错误。因此 mmap 方式适合操作小数据块，不会出现多进程同时写的问题。

还可以通过 sendfile 系统调用达到零拷贝的效果，sendfile 系统调用实现零拷贝原理如下表 4.3 所示：

表 4.3 sendfile 实现零拷贝

sendfile(socket, file, len);

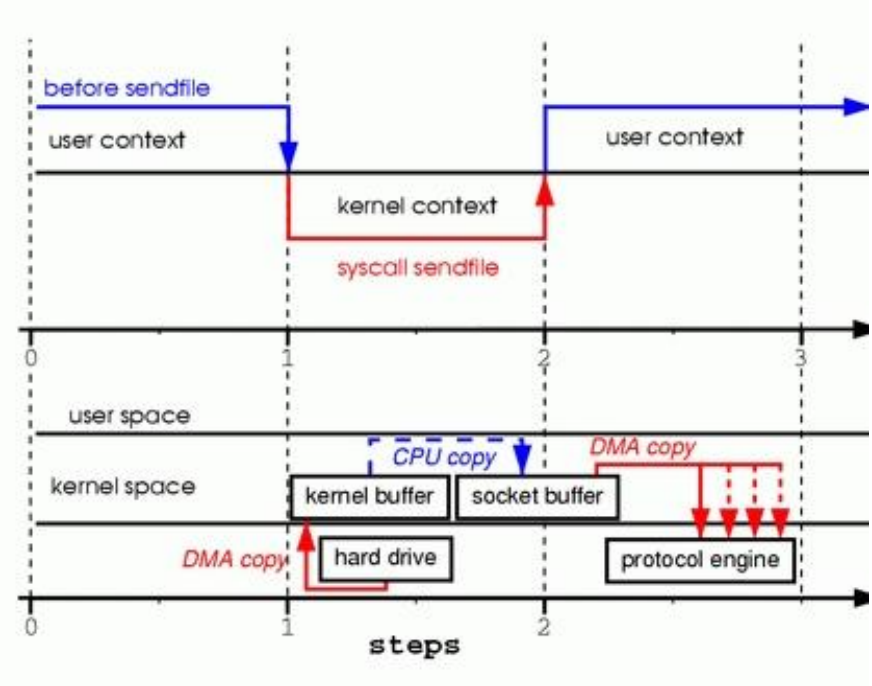

图 4.6 sendfile 实现零拷贝流程图

上图 4.6 是 sendfile 系统调用流程图。首先 sendfile 系统调用会引起数据从 DMA 引擎到内核共享缓冲区的拷贝和内核共享缓冲区到 socket 缓冲区的拷贝[43]。然后从套接字缓冲区中的数据复制到网络协议引擎。从分析可以看出 sendfile 方式相比于 mmap 方式数据拷贝次数和上下文切换次数更少[37]。

Sendfile 方式完全不需要将数据拷贝到用户空间中去，适合于应用程序不需要对数据进行处理的情况[31]。sendfile 方式完全没有内核空间到用户空间的跨越，可以极大的提高存储管理效率。

对于消息队列来说，队列系统只需要将消息持久化在磁盘并且在需要的时候将消息原封不动的传输给消费者，因此非常适合零拷贝技术中用户空间不需要对数据进行任何处理的需求，LSMQ 采用了 sendfile 的方式实现从磁盘到 socket 的数据传输，相比于传统方式 IO 调用，sendfile 方式只需要 1 次拷贝和 1 次系统调用，也可以减少一半的上下文从内核空间到用户空间的切换次数。大大提高了系统的存储管理效率。

4.9 本章小结

本章详细论述了 LSMQ 各模块的设计与实现。首先介绍了 LSMQ 的整体架构，然后介绍各模块的设计和工作机制，具体介绍了 LSMQ 的消息读写流程，推模式和拉模式实现，以及系统中采用的零拷贝技术。

系统主要包括 topic 管理模块、offset 管理模块、网络管理模块、LSM-Tree 模块，其中 LSM-Tree 包括第三章介绍的内存模块、磁盘模块、日志模块。Topic 管理模块使得 LSMQ 可以应对多队列的场景，该模块管理了每一个队列头部节点对应的 key 以及队列的长度，LSMQ 的读写操作都是从 topic 管理模块开始的。Offset 管理模块是为了队列的推模式而设计的，LSMQ 通过 offset 管理模块，管理了每一个消费者上一次消费的消息 ID，然后该模块定期发起读请求，查找 topic 模块中队列最新的 ID 和自己管理的 ID 是否一样，如果不一样，说明队列中有新消息，读取这一部分新消息，推送给队列消费者，通过这种方式可以满足实时性要求不是特别高的场景下推模式的需求。拉模式时用户自己保存了消费 offset 来实现的，用户的 offset 也就是整个队列的偏移。有了 offset 就可以判断用户是否有新消息以及有多少条新消息。

LSMQ 的写流程是先写日志，每一条记录都需要一次对日志文件的写操作，这样保证了系统的写操作一致性，当系统发生故障时，保证所有的数据即使没有实时写

到磁盘上，一定也到了日志中，这样在系统从故障中恢复的时候，数据不会丢。LSMQ 的读流程在系统没有产生消息堆积的时候，只涉及到一次内存 IO 操作。当数据堆积到磁盘时，首先通过内存中的磁盘文件数值范围索引，查找到消息所在的文件，然后通过文件内索引查找消息所在的数据块，最后通过二分查找数据块中的记录，读取消息记录。最后介绍了零拷贝技术的原理和在 LSMQ 中的应用。

5 测试及结果分析

本章节主要介绍给予 LSM-Tree 的持久化消息队列系统 LSMQ 的实际测试结果，并同时测试持久化 ActiveMQ、非持久化 ActiveMQ 以及基于内存的消息队列 Redis 的读写性能，并对测试结果进行对比分析。

5.1 测试目标

在分布式文件系统中，引入消息队列的主要目的是为了减少各模块之间的耦合，让各模块专注于提高自己的数据处理能力，以及挡住前端模块的数据洪峰，消息队列系统性能评价参数主要包括单位时间消息写入量（写 MPS）、单位时间内消息读取量（读 MPS）、系统消息堆积能力。因此，本文中的测试主要从这三个方面来评价整个系统的服务性能，MPS 的定义如下：

MPS = 发送 M 条消息 / 发送 M 条消息所用时间

5.2 测试环境

测试所采用的服务器配置如图 5.1 所示。

表 5.1 测试环境硬件配置

设备	具体配置
CPU	Intel(R) Xeon(R) CPU E5620 @ 2.40GHz * 16
内存	DDR3REG 2G
硬盘	SATA 300G @ 7200R/M
操作系统	Ubuntu server 12.04
内核版本	Linux 3.2.18-164.el5

5.3 MPS 测试

在进行读写 MPS 测试的过程中，用多进程模拟多个客户端并发生成消息，每个 Client 连续不断的发送多个消息写入请求，这里的 MPS 是指单位时间内各消息队列系统写入消息数量。MPS 测试程序实例如下表 5.2 所示：

表 5.2 MPS 测试程序实例

```
var logger = require('./lib/WPLogger.js');
var stomp = require('stomp');
var cluster = require('cluster');
var client_num = 1;
var max = 10000;
var persistent = true;
var start_timestamp,end_timestamp;
start_timestamp = new Date().getTime();
var recp_num = 0;
var receipt = true;
var stomp_args = {
	port: 61613,
	host: '10.161.78.251',
	debug: false
}

if (cluster.isMaster) {
	for (var i = 0; i < client_num; i++) {
		cluster.fork();
	}
	cluster.on('death', function(worker) {
		console.log('worker ' + worker.pid + ' died');
		cluster.fork();
	});
} else {
	var client = new stomp.Stomp(stomp_args);
	var queue = '/queue/test_stomp';
	client.connect();
	client.on('connected', function() {
		var i = 0;
		while(i <= max){
			client.send({
				'destination': queue,
```

```
                    'body': 'Testing\n\ntesting1\n\ntesting2 ' + i,
                    'persistent': persistent
                }, receipt);
                i++;
            }
        });

        client.on('receipt', function(receipt) {
            recp_num ++;
            if(recp_num > max){
                end_timestamp = new Date().getTime();
                logger.info("end at: " + end_timestamp);
                logger.info("MPS:",max * 1000 * client_num / (end_timestamp - start_timestamp));
            }
        });

        client.on('error', function(error_frame) {
            console.log(error_frame.body);
            client.disconnect();
        });
}
```

5.3.1 内存 ActiveMQ 读写 MPS 测试

ActiveMQ 在写消息时，如果消息配置中指定消息不持久化，消息将会堆积在内存中直到被消费者消费，这样虽然系统的读写性能可以达到最佳，但是系统容量却受到内存大小的限制。下面通过测试不同客户端数量下发送 1 万条消息所用时间来计算 MPS，下图 5.2 是单客户端下发送 1 万条消息的 MPS。

```
root@AY1310312224289811a4Z:/home/orandofang/paper# node activemq-write.js
[2014-01-17 00:43:21.221] [INFO] [default] - end at: 1389890601220
[2014-01-17 00:43:21.228] [INFO] [default] - MPS: 4411.1160123511245
```

图 5.1 内存 ActiveMQ 写 MPS 测试

内存 ActiveMQ 读消息测试。在写入消息时配置消息不持久化，通过测试不同客

户端数量下读取 1 万条消息所用时间来计算 MPS，下图 5.2 是 5 个客户端情况下，读取 1 万条消息的 MPS。

```
^Croot@AY1310312224289811a4Z:/home/orandofang/paper# node activemq-read.js
[2014-01-17 00:44:27.653] [INFO] [default] - end at: 1389890667652 2000
[2014-01-17 00:44:27.655] [INFO] [default] - end at: 1389890667652 2000
[2014-01-17 00:44:27.709] [INFO] [default] - MPS: 4870.920603994155
[2014-01-17 00:44:27.718] [INFO] [default] - MPS: 4844.961240310077
[2014-01-17 00:44:27.724] [INFO] console - worker 27453 died
[2014-01-17 00:44:27.736] [INFO] console - worker 27452 died
[2014-01-17 00:44:27.768] [INFO] [default] - end at: 1389890667768 2000
[2014-01-17 00:44:27.783] [INFO] [default] - MPS: 4683.84074941452
[2014-01-17 00:44:27.770] [INFO] [default] - end at: 1389890667766 2000
[2014-01-17 00:44:27.790] [INFO] console - worker 27456 died
[2014-01-17 00:44:27.788] [INFO] [default] - MPS: 4690.431519699812
[2014-01-17 00:44:27.793] [INFO] console - worker 27454 died
[2014-01-17 00:44:27.828] [INFO] [default] - end at: 1389890667828 2000
[2014-01-17 00:44:27.835] [INFO] [default] - MPS: 4595.588235294118
[2014-01-17 00:44:27.838] [INFO] console - worker 27455 died
```

图 5.2 内存 ActiveMQ 读 MPS 测试

5.3.2 磁盘 ActiveMQ 读写 MPS 测试

磁盘 ActiveMQ 写测试。在发送的消息时，指定消息需要持久化，并且持久化引擎为 kahaDB 中，kahaDB 时专门为 ActiveMQ 设计的持久化存储引擎，数据存储在磁盘文件中，磁盘文件使用 B+树来组织数据，使用 B+树是为了更快的将数据从磁盘中找出来，从而提高消息队列的读效率，kahaDB 设计中完整的 B+树结构存储在磁盘上，部分 B+树节点缓存在内存中，显而易见如果 B+树所有节点都存储在内存中效率会更高，但这样系统容量又会受到内存大小的限制。kahaDB 还可以检测到数据丢失，如果出现机器故障导致数据丢失，kahaDB 默认会抛出一个异常，然后系统关闭，需要有管理员手动去调查丢失的数据。

通过测试写入 1 万条消息所用时间来计算 MPS，下图 5.3 是 1 个客户端下磁盘 ActiveMQ 的写 MPS。

```
root@AY1310312224289811a4Z:/home/orandofang/paper# node activemq-write.js
[2014-01-17 00:49:16.721] [INFO] [default] - end at: 1389890956721
[2014-01-17 00:49:16.730] [INFO] [default] - MPS: 145.3319381467271
```

图 5.3 磁盘 ActiveMQ 写 MPS 测试

磁盘 ActiveMQ 读测试，通过测试读取 1 万条刚写入的磁盘消息所用时间来计算 MPS，下图 5.4 是 5 个客户端下，读取 1 万条磁盘消息 MPS。

```
root@AY1310312224289811a4Z:/home/orandofang/paper# node activemq-read.js
[2014-01-17 00:52:43.498] [INFO] [default] - end at: 1389891163496 2000
[2014-01-17 00:52:43.574] [INFO] [default] - MPS: 6578.9473684210525
[2014-01-17 00:52:43.601] [INFO] console - worker 27940 died
[2014-01-17 00:52:43.653] [INFO] [default] - end at: 1389891163652 2000
[2014-01-17 00:52:43.695] [INFO] [default] - MPS: 6016.847172081829
[2014-01-17 00:52:43.712] [INFO] console - worker 27941 died
[2014-01-17 00:52:43.741] [INFO] [default] - end at: 1389891163741 2000
[2014-01-17 00:52:43.760] [INFO] [default] - MPS: 5743.825387708213
[2014-01-17 00:52:43.781] [INFO] console - worker 27942 died
[2014-01-17 00:52:43.799] [INFO] [default] - end at: 1389891163799 2000
[2014-01-17 00:52:43.808] [INFO] [default] - MPS: 5617.9775280898875
[2014-01-17 00:52:43.810] [INFO] console - worker 27943 died
[2014-01-17 00:52:43.819] [INFO] [default] - end at: 1389891163819 2000
[2014-01-17 00:52:43.827] [INFO] [default] - MPS: 5515.7198014340875
[2014-01-17 00:52:43.832] [INFO] console - worker 27944 died
```

图 5.4 磁盘 ActiveMQ 读 MPS 测试

5.3.3 内存 Redis 队列的读写 MPS

Redis 队列的写测试，向 redis 中循环写入 1 万条长度固定的消息，测试在客户端数量变化的情况下，MPS 变化，下图 5.5 是 1 个客户端情况下，Redis 队列的写 MPS。

```
root@AY1310312224289811a4Z:/home/orandofang/paper# node redis-write.js
[2014-01-17 00:57:35.962] [INFO] [default] - write start at: 1389891455962
[2014-01-17 00:57:36.100] [INFO] [default] - write start at: 1389891456100
[2014-01-17 00:57:36.114] [INFO] [default] - redis redis on connected.
[2014-01-17 00:57:36.119] [INFO] [default] - redis redis on ready.
[2014-01-17 00:57:36.748] [INFO] [default] - write end at: 1389891456748
[2014-01-17 00:57:36.750] [INFO] [default] - time used: 648
[2014-01-17 00:57:36.750] [INFO] [default] - MPS: 15432.098765432098
[2014-01-17 00:57:36.756] [INFO] console - worker 28217 died
```

图 5.5 Redis 队列写 MPS 测试

Redis 队列的读测试，从 redis 中循环读取 1 万条消息，测试在客户端数量变化的情况下，MPS 变化，下图 5.6 是 5 个客户端情况下，Redis 队列的读 MPS。

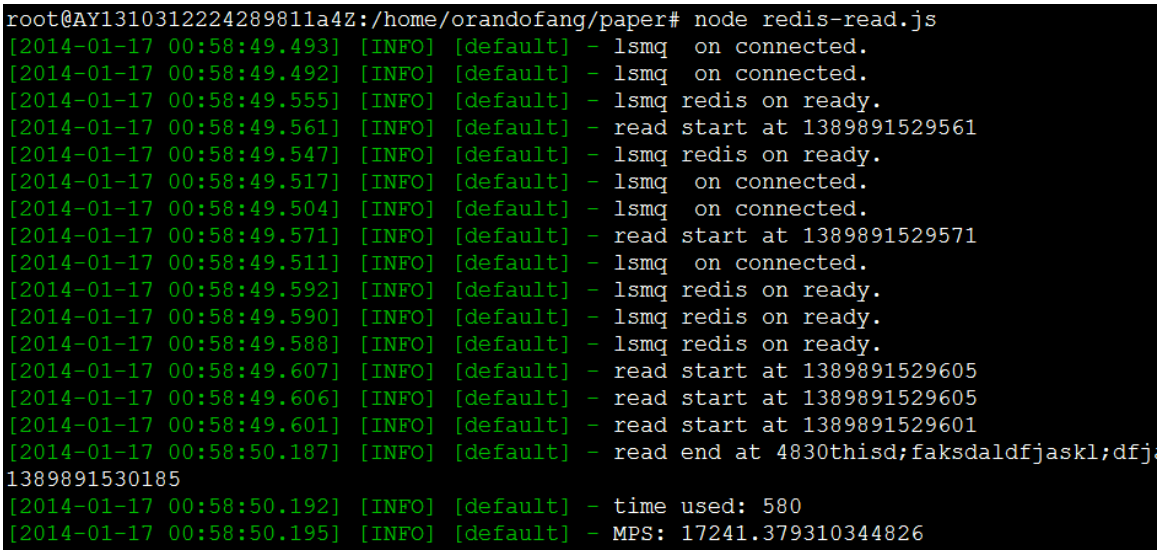

图 5.6 Redis 队列读 MPS 测试

5.3.4 LSMQ 队列的读写 MPS

LSMQ 的写 MPS 测试，循环向 LSMQ 中写入 1 万条消息，测试不同客户端数量下，LSMQ 的写 MPS 变化，下图 5.7 是 1 个客户端情况下 LSMQ 的写 MPS 测试。

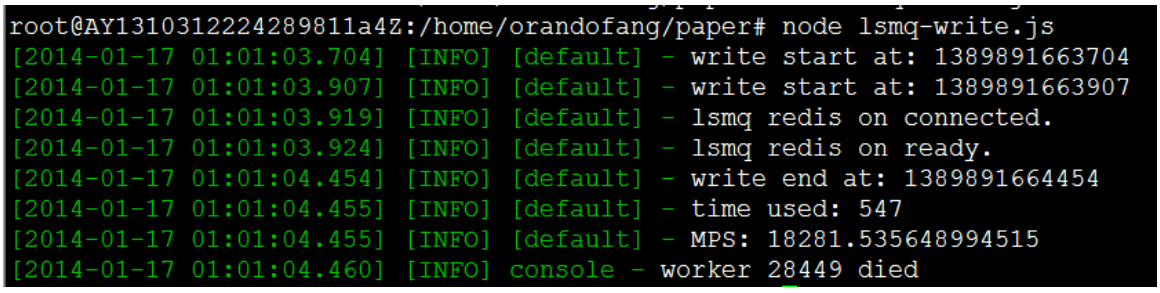

图 5.7 LSMQ 写 MPS 测试

LSMQ 的读 MPS 测试，从 LSMQ 中读取 1 万条消息，测试不同客户端数量下 LSMQ 的读 MPS 变化。下图 5.8 是 5 个客户端下 LSMQ 的读 MPS 数据。

```
root@AY1310312224289811a4Z:/home/orandofang/paper# node lsmq-read.js
[2014-01-17 01:02:06.918] [INFO] [default] - lsmq  on connected.
[2014-01-17 01:02:06.952] [INFO] [default] - lsmq redis on ready.
[2014-01-17 01:02:06.963] [INFO] [default] - read start at 1389891726963
[2014-01-17 01:02:06.941] [INFO] [default] - lsmq  on connected.
[2014-01-17 01:02:06.945] [INFO] [default] - lsmq  on connected.
[2014-01-17 01:02:06.973] [INFO] [default] - lsmq redis on ready.
[2014-01-17 01:02:06.979] [INFO] [default] - lsmq redis on ready.
[2014-01-17 01:02:06.979] [INFO] [default] - read start at 1389891726979
[2014-01-17 01:02:06.983] [INFO] [default] - read start at 1389891726983
[2014-01-17 01:02:06.968] [INFO] [default] - lsmq  on connected.
[2014-01-17 01:02:06.974] [INFO] [default] - lsmq  on connected.
[2014-01-17 01:02:07.003] [INFO] [default] - lsmq redis on ready.
[2014-01-17 01:02:07.008] [INFO] [default] - read start at 1389891727008
[2014-01-17 01:02:07.008] [INFO] [default] - lsmq redis on ready.
[2014-01-17 01:02:07.012] [INFO] [default] - read start at 1389891727012
[2014-01-17 01:02:07.602] [INFO] [default] - read end at 5525thisd;faksdaldfjaskl;dfjal;ks
1389891727602
[2014-01-17 01:02:07.606] [INFO] [default] - time used: 619
[2014-01-17 01:02:07.608] [INFO] [default] - MPS: 16155.08885298869
```

图 5.8 LSMQ 读 MPS 测试

5.4 消息堆积能力测试

在消息堆积能力测试中，通过测试内存 ActiveMQ、磁盘 ActiveMQ、Redis、LSMQ 写入 20 万条消息情况下内存和磁盘占用情况来测试系统的消息堆积能力，消息堆积能力测试程序实例如下表 5.4 所示。

表 5.3 消息堆积能力测试程序实例

```
var logger = require('./lib/WPLogger.js');
lsmq = require('lsmq-client');

var cluster = require('cluster');
var client_num = 1;
var max = 20000;
var start_timestamp,end_timestamp;
var recv_num = 0;
start_timestamp = new Date().getTime();
logger.info("write start at:",start_timestamp);

if (cluster.isMaster) {
    for (var i = 0; i < client_num; i++) {
        cluster.fork();
    }
```

```
    cluster.on('death', function(worker) {
        console.log('worker ' + worker.pid + ' died');
    });
} else {
    var lsmq_client = lsmq.createClient("8323", "10.161.78.251");

    lsmq_client.on("error", function (err) {
        logger.error("lsmq on error:" + err);
    });
    lsmq_client.on("ready", function () {
        logger.info("lsmq on ready.");
        var i = 0;
        while (i <= max) {
            (function(index){
                lsmq_client.lpush("test_list",
index+"thisd;faksdaldfjaskl;dfjal;ksdfjlkasdjfklasjdfklasdf", function (err, reply) {
                    if(!err){
                        recv_num ++;
                    }
                    if(recv_num >= max){
                        end_timestamp = new Date().getTime();
                        logger.info("write end at:",end_timestamp);
                        logger.info("time used:",end_timestamp-start_timestamp);
                        logger.info("MPS:",max * client_num *
1000/(end_timestamp-start_timestamp));
                        process.exit();
                    }
                });
            })(i);
            i++;
        }
    });

    lsmq_client.on("reconnecting", function (arg) {
```

```
        logger.info("lsmq on reconnecting: ", JSON.stringify(arg));
    });
    lsmq_client.on("connect", function () {
        logger.info("lsmq on connected.");
    });
}
```

消息堆积能力测试结果如下表 5.4 所示。

表 5.4 消息堆积能力测试结果

队列	物理内存	虚拟地址空间	磁盘
LSMQ	9588 KB	167084 KB	42 MB
Redis	55748 KB	91796 KB	23 MB
内存 ActiveMQ	461764 KB	1826804 KB	39 MB
磁盘 ActiveMQ	350328 KB	1903812 KB	67 MB

上表中 LSMQ 内存占用最少，原因是 LSMQ 并没有将所有消息数据都存储在内存中，内存中只保存了一个轻量级的排序结构以及磁盘数据文件范围索引。LSMQ 在消息堆积能力方面的测试基本达到了设计目标。

5.5 本章小结

本章以 LSMQ 持久化消息队列系统为测试背景，对 LSM 机制进行了测试与分析，分为读写 MPS 测试和消息堆积能力测试两个部分，分别与内存和磁盘消息队列系统 ActiveMQ、Redis 进行性能对比。

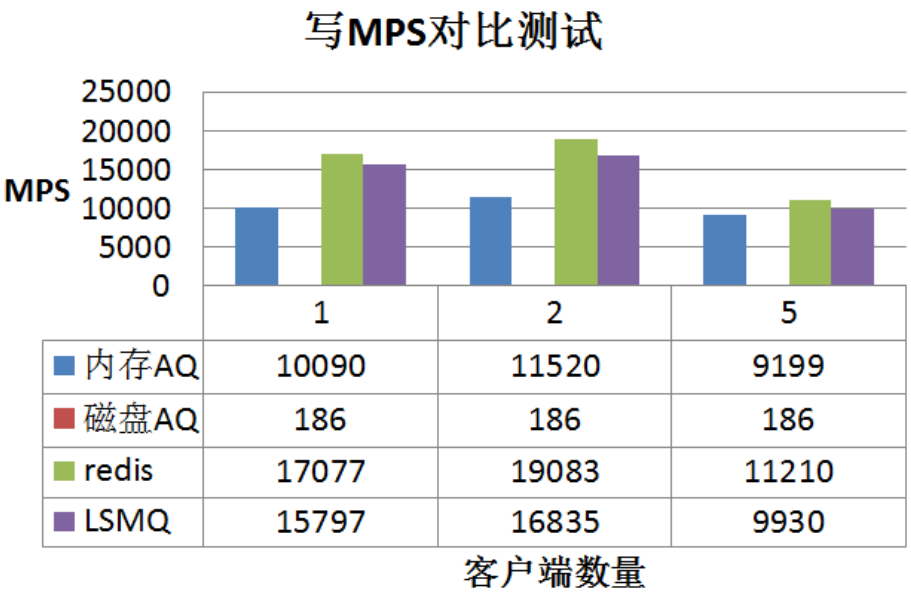

图 5.9 写 MPS 对比测试结果

写 MPS 对比测试结果如上图 5.9 所示，写 MPS 测试发现 LSMQ 由于 LSM-Tree 的堆积延时写入特性，写吞吐量几乎达到了内存队列 Redis 的标准，远远高于普通磁盘队列 ActiveMQ。

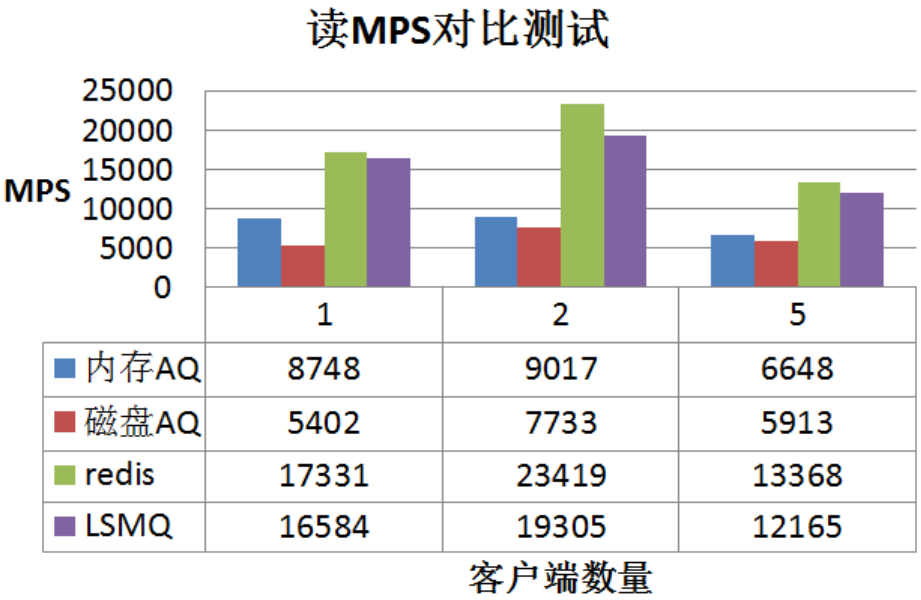

图 5.10 读 MPS 对比测试结果

读 MPS 对比测试结果如上图 5.10 所示，读 MPS 对比测试发现，LSMQ 由于队

列的 pop 操作数据都在内存跳表中，磁盘的预读取策略大大提高了消息的读效率，基本达到设计预期。

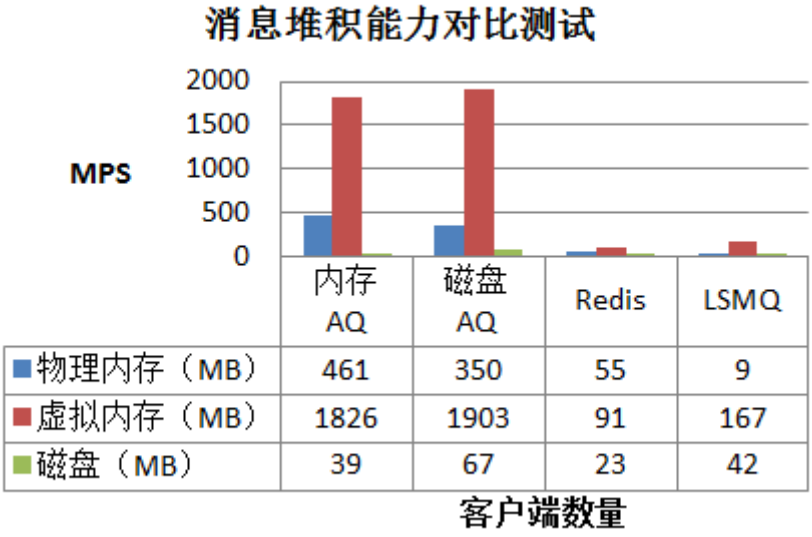

图 5.11 消息堆积能力对比测试

消息堆积能力对比测试结果如上图 5.11 所示。消息堆积能力测试中，LSMQ 由于内存中采用的是轻量的跳表结构，内存占用非常少，而消息堆积能力取决于磁盘大小，基本达到了设计预期。

6 全文总结

在分布式文件系统设计中，各个模块负责优化各自的处理能力，与其他模块之间产生耦合会极大的提高系统复杂度，同时各个模块之间存在着处理能力不均衡的问题，消息队列系统能避免各模块之前的耦合，同时均衡各模块之间处理能力的不均衡。本文所做的工作具体如下：

1. 介绍了在大数据时代，分布式以及集群化成为必然趋势。在这种分布式环境下，系统模块之间去耦合问题越来越突出。通过对各种消息队列解决方案的分析，提出了分布式文件系统持久化消息队列系统的设计思路。
2. 在分析了 B+树作为磁盘存储结构的缺陷后，设计与实现了 LSM-Tree 机制，包括系统的整体架构设计及 LSM 机制的具体设计及实现细节，主要包括高吞吐量模型，提高系统的消息堆积能力；基于定时请求的推模式，使得整个系统的设计大大简化。
3. 以 LSMQ 消息队列为测试背景，对内存消息队列、磁盘快照消息队列、磁盘消息队列进行了测试与分析，分为读请求测试和写请求测试、消息堆积能力三部分，测试结果表明 LSMQ 可以在保障消息读写性能的前提下明显提高单机消息堆积能力。

本文提出的方案虽然取得了一些成效，但仍有许多后续工作要做，系统仍然有一些不足之处需要解决，具体如下：

1. 针对读操作的缓存，可以设计以磁盘文件为缓存单位或以磁盘文件 block 块为缓存单位，来减少消息读取时的磁盘随机 IO 操作。
2. LSMQ 内部策略是基于机械硬盘 IO 特性设计的，使用其他硬件存储如 SSD 硬盘时，需要做相应的策略调整。

致 谢

光阴如梭，转眼间硕士生涯即将到达尾声。从保研加入信息存储及应用实验室这个大家庭到现在已三年有余，回首这三年的时光，在各位老师、师兄师姐以及实验室同学的关心和帮助下，我的硕士生涯即充实又快乐，在收获专业知识、技能的同时，我也收获了一群良师益友，相信这将成为我人生中宝贵的财富。在毕业之际，谨向信息存储及应用实验室大家庭的各位成员以及关心帮助我的朋友、亲人们表达我最诚挚的谢意!

首先要感谢我的导师施展老师，保研面试的时候，施展老师平易近人的面试风格给我留下了深刻的印象。后来很幸运的加入了施展老师所带领的广域网存储小组，从此开始了快乐充实的硕士生涯。施展老师在专业学术领域博学多识、造诣深厚，被大家誉为“百宝箱”。此外，施展老师乐观，积极向上的生活态度也深深影响着我。硕士阶段的成长离不开施展老师的悉心教导，施展老师，谢谢您！

感谢实验室的冯丹老师、王芳老师、李洁琼老师、童薇老师、谭志鹏老师、刘景宁老师、华宇老师、曾令仿老师、熊双莲老师等各位老师。正是源于各位老师的辛勤付出，实验室才会有如此浓厚的学习氛围，我们才有了如此优异的学习环境与科研平台。

感谢小分队赵千、付宁、王艳萍、桂笠同学，和大家一起工作学习娱乐的日子是那么愉快！不管是在科研技术方面，还有在为人处世方面，从你们身上我学到了很多，庆幸有你们。

感谢广域网存储小组的赵恒、陈云云学长，初入实验室的我懵懂无知，是你们耐心的为我解惑，耐心的帮助我；感谢贺艳、郑颖学姐在生活上给予我无微不至的关怀；感谢焦田丰、李宁、李勇、柳青博士的帮助；感谢张成文、郭鹏飞、黄力、韩江同学；正因为大家每个人的存在，F309 才如此欢乐，如此温暖，如此积极向上！

感谢我的室友白梁、唐路遥同学，谢谢你们一直以来对我的帮助和包容。感谢宣传部李彬彬老师在我研究生学习过程中对我的关照和帮助。

最后要感谢我的家人，感谢父母和兄长一直以来无私的关爱和默默的支持，感谢你们给了我一个和谐幸福的家庭，你们的爱是我最坚实的后盾。

参考文献

[1] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system. ACM SIGOPS Operating Systems Review, 1991. 25(5): 1-15.

[2] Paksula M. Persisting Objects in Redis Key-Value Database. Science, 2010: 24-30.

[3] The Apache Software Foundation. Active MQ Introduction[EB/OL]. (2008-12-08).http://activemq.apache.org/version-5-getting-started.html#GettingStarted-Introduction.

[4] 徐晶, 许炜. 消息中间件综述[J]. 计算机工程, 2005, 33(16):73-76.

[5] 黄华, 杨德志, 张建刚. 分布式文件系統发展历史. 王雪涛, 刘伟杰. 分布式文件系统[J]. 科技信息（学术版）, 2006, article 11.2~4

[6] 刘江, 淘宝开源平台正式上线. 程序员. 2010, 6: 6-7.

[7] O Neil, P., et al., The log-structured merge-tree (LSM-tree). Acta Informatica, 1996. 33(4): 351-385.

[8] Acquisti A, Gross R. Imagined communities: Awareness, information sharing, and privacy on the Facebook. in: Springer, 2006: 36-58.

[9] 那岩. leveldb实现解析. 淘宝核心系统研发, 2011: 1-32.

[10] Beaver D, Kumar S, Li H C, et al. Finding a needle in Haystack: Facebook’s photo storage. Proc. 9th USENIX OSDI. 2010: 146-159.

[11] Souders S. Even faster web sites. O'Reilly Media, Inc., 2009: 15-17.

[12] Comer D. Ubiquitous B-tree. ACM Computing Surveys (CSUR). 1979, 11(2): 121-137.

[13] 简朝阳. MySQL 性能调优与架构设计. 电子工业出版社, 2009: 136-140.

[14] 张江陵, 计算机研究, 冯丹. 海量信息存储. 科学出版社, 2003: 3-5.

[15] T.Oracle. Using the MySQL memcached UDFs. MySQL Engineering’s notes, April 2009: 30-36.

[16] A. Khetrapal, V. Ganesh. HBase and Hypertable for large scale distributed storage systems:A Performance evaluation for Open Source BigTable Implementations. Retrieved fromuavindia.com/ankur. 2008: 1-8.

[17] K. Seguin. The Little Redis Book. Creative Commons, 2011: 1-28.

[18] A. Lakshman, P. Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev, vol. 44. 2010: 35-40.

[19] H. Lim, B. Fan, D. G. Andersen et al. SILT: a memory-efficient, high-performance

key-value store. In: Proc. SOSP '11, New York, NY, USA, 2011: 1-13.

[20] V.F. Nicola, A. Dan, and D.M. Dias. Analysi of the generalized clock buffer replacement scheme for database transaction processing. In: Proc. ACM SIGMETRICS Conf, 1992: 35-46.

[21] T. Macedo, F. Oliveira. Redis Cookbook. O'Reilly Media, July 2011: 12-34.

[22] M. Paksula. Persisting Objects in Redis Key-Value Database. Science, 2010: 131-137.

[23] S. Sanfilippo. Redis Manifesto. Redis Engineering's notes, 2011: 1-5.

[24] Davies. BeansDB设计与实现. Douban's Engineering's notes. Dec. 2010: 1-23.

[25] Borthakur Dhruba. The hadoop distributed file system: Architecture and design. 2007.3~10

[26] 明亮. 基于InfiniBand互联的海量存储系统高性能策略研究: 博士学位论文, 华中科技大学, 2012.54~59

[27] 孙海峰.分布式文件系统数据读写流程分析与优化: 硕士学位论文. 华中科技大学, 2013. 22~36

[28] D.Bovet, M.Cesati. 深入理解Linux内核. 第三版. 陈莉君. 北京: 中国电力出版社, 2007. 312~313

[29] 周应超. Linux 内核的文件 cache 管理机制介绍. Technical report, 中科院计算所, 2006. URL http://www.ibm.com/developerworks/cn/linux/l-cache/.

[30] Sonny Rao, Dominique Heger, Steven Pratt. Examining linux 2.6 page-cache performance. In : OLS'05 Proceedings of the Linux Symposium. Ottawa, Ontario, Canada, 2005. 79~90

[31] Patterson R Hugo, Gibson Garth A, Ginting Eka, et al. Informed prefetching and caching[M]. In :Proc of the 15th ACM Symp on Operating System Principles, Copper Mountain Resort, CO, Dec. 3-6, 1995. 79~95

[32] 杨德志, 黄华, 张建刚等. 大容量, 高性能, 高扩展能力的蓝鲸分布式文件系统. 计算机研究与发展, 2005, 42(6): 1028-1033

[33] 黄华, 张建刚, 许鲁. 蓝鲸分布式文件系统的分布式分层资源管理模型. 计算机研究与发展, 2005, 42(6): 1034-1038

[34] Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system. in Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), Berkeley, CA, USA: USENIX Assoc, 2006: 307-320

[35] Singh H J, Singh V P. High Scalability of HDFS using Distributed Namespace. International Journal of Computer Applications, 2012, 52(17): 30-37

[36] Support for RW/RO snapshots in HDFS https: //issues. apache. org/jira/browse/ HDFS-2802. 2012

[37] Stevens W, Stephen A. UNIX环境高级编程, 人民邮电出版社 2006: 29-86.

[38] Toptsis A A. B**-tree: a data organization method for high storage utilization. in Proceedings of ICCI'93: 5th International Conference on Computing and Information, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1993: 277-281

[39] Rodeh O. B-trees, shadowing, and clones. Trans. Storage, 2008, 3(4): 1-27

[40] Braginsky A, Petrank E. A lock-free B+tree. in Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures, Pittsburgh, Pennsylvania, USA: ACM, 2012: 58-67

[41] Wu S, Jiang D, Ooi B C, et al. Efficient b-tree based indexing for cloud data processing. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1207-1218

[42] Graefe G. Modern B-Tree Techniques. Foundations and Trends in Databases, 2010, 3(4): 203-402

[43] Liu T, Zhu H, Chang G, et al. The design and implementation of zero-copy for Linux. in 2008 Eighth International Conference on Intelligent Systems Design and Applications, Piscataway, NJ, USA: IEEE, 2008: 121-126