

谈谈 RocketMQ 5.0 分级存储背后一些有挑战的技术优化

RocketMQ 5.0 提出了分级存储的新方案，经过数个版本的深度打磨，RocketMQ 的分级存储日渐成熟，并成为降低存储成本的重要特性之一。事实上，几乎所有涉及到存储的产品都会尝试转冷降本，如何针对消息队列的业务场景去做一些有挑战的技术优化，是非常有意思的事。

这篇文章就跟大家探讨下，在消息系统这样一个数据密集型应用的模型下，技术架构选型的分析与权衡，以及分级存储实现与未来演进，让云计算的资源红利真正传达给用户。

1. 背景与需求

RocketMQ 诞生于 2012 年，存储节点采用 shared-nothing 的架构读写自己的本地磁盘，单节点上不同 topic 的消息数据会顺序追加写 CommitLog 再异步构建多种索引，这种架构的高水平扩展能力和易维护性带来了非常强的竞争力。

随着存储技术的发展和各种百G网络的普及，RocketMQ 存储层的瓶颈逐渐显现，一方面是数据量的膨胀远快于单体硬件，另一方面存储介质速度和单位容量价格始终存在矛盾。在云原生和 Serverless 的技术趋势下，只有通过技术架构的演进才能彻底解决单机磁盘存储空间上限的问题，同时带来更灵活的弹性与成本的下降，做到“鱼与熊掌兼得”。

在设计分级存储时，希望能在以下方面做出一些技术优势：

- **实时：** RocketMQ 在消息场景下往往是一写多读的，热数据会被缓存在内存中，如果能做到“准实时”而非选用基于时间或容量的淘汰算法将数据转储，可以减小数据复制的开销，利于缩短故障恢复的 RTO。读取时产生冷读请求被重定向，数据取回不需要“解冻时间”，且流量会被严格限制以防止对热数据写入的影响。
- **弹性：** shared-nothing 架构虽然简单，扩容或替换节点的场景下待下线节点的数据无法被其他节点读取，节点需要保持相当长时间只读时间，待消费者消费完全部数据，或者执行复杂的迁移流程才能扩容，这种“扩容很快，扩容很慢”的形态一点都不云原生，更长久的消息保存能力也会放大这个问题。分级存储设计如果能通过 shared-disk (共享存储) 的方式让在线节点实现代理读取下线节点的数据，既能节约成本也能简化运维。
- **差异化：** 廉价介质随机读写能力较差，类 LSM 的结构都需要大量的 compaction 来压缩回收空间。在满足针对不同 topic 设置不同的生命周期（消息保留时间，TTL）等业务需求的前提下，结合消息系统数据不可变和有序的特点，RocketMQ 自身需要尽量少的做格式“规整”来避免反复合并的写放大，节约计算资源。

- 竞争力： 分级存储还应考虑归档压缩，数据导出，列式存储和交互式查询分析能力等高阶技术演进。

2. 技术架构选型

2.1. 不同视角

不妨让我们站在一个新的视角看问题，消息系统对用户暴露的是收发消息，位点管理等一系列的 API，为用户提供了一种能够优雅处理动态数据流的方式，从这个角度说：消息系统拓宽了存储系统的边界。 其实服务端应用大多数是更底层 SQL，POSIX API 的封装，而封装的目的在于简化复杂度的同时，又实现了信息隐藏。

消息系统本身关注的是高可用，高吞吐和低成本，想尽量少的关心存储介质的选择和存储自身的系统升级，分片策略，迁移备份，进一步冷热分层压缩等问题，减少存储层的长期维护成本。

一个高效的、实现良好的存储层应该对不同存储后端有广泛的支持能力，消息系统的存储后端可以是本地磁盘，可以是各类数据库，也可以是分布式文件系统，对象存储，他们是可以轻松扩展的。

2.2. 存储后端调研

幸运的是，几乎所有的“分布式文件系统”或者“对象存储”都提供了“对象一旦上传或复制成功，即可立即读取”的强一致语义，就像 CAP 理论中的描述 **“Every read receives the most recent write or an error”** 保证了“分布式存储系统之内多副本之间的一致性”。对于应用来说，没有“拜占庭错误”是非常幸福的（本来有的数据变没了，破坏了存储节点的数据持久性），更容易做到“应用和分布式存储系统之间是一致的”，并显著减少应用的开发和维护成本。

常见的分布式文件系统有 Ali Pangu，HDFS，GlusterFS，Ceph，Lustre 等。对象存储有 Amazon S3，Aliyun OSS，Azure Blob Storage，Google Cloud Storage，OpenStack Swift 等。他们的简单对比如下：

- **API 支持**： 选用对象存储作为后端，通常无法像 HDFS 一样提供充分的 POSIX 能力支持，对于非 KV 型的操作往往存在一定性能问题，例如列出大量对象时需要数十秒，而在分布式文件系统中这类操作只需要毫秒甚至微秒。如果选用对象存储作为后端，弱化的 API 语义要求消息系统本身能够有序管理好这些对象的元数据。
- **容量与水平扩展**： 对于云产品或者大规模企业的存储底座来说，以 HDFS 为例，当集群节点超过数百台，文件达到数亿量级以上时，NameNode 会产生性能瓶颈。一旦底层存储由于容量可用区等因素出现多套存储集群，这种“本质复杂度”在一定程度上削弱了 shared-disk 的架构简单性，并将这种复杂度向上传递给应用，影响消息产品本身的多租，迁移，容灾设计。典型的情况就是大型企业为了减少爆炸半径，往往会部署多套 K8s 并定

制上层的 Cluster Federation（联邦）。

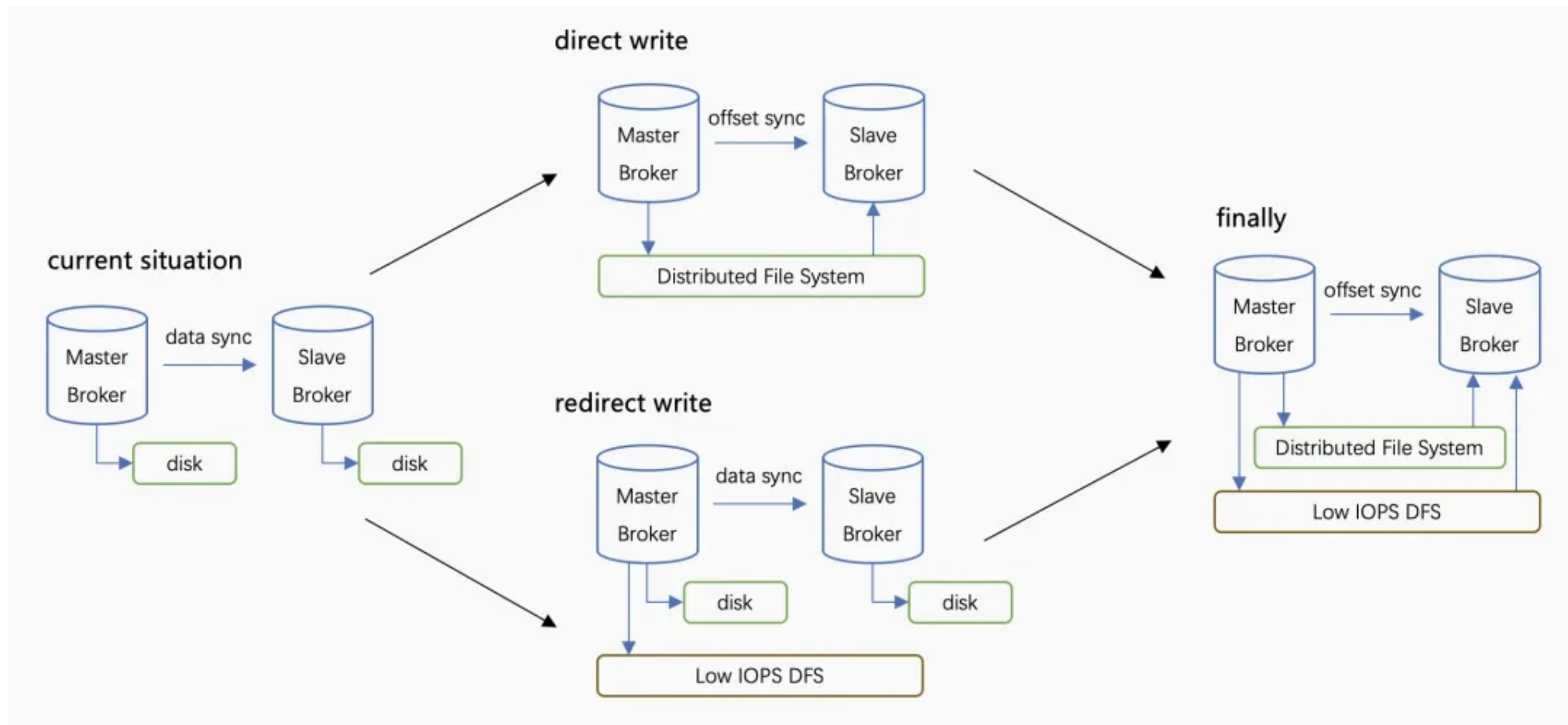
- 成本：以国内云厂商官网公开的典型目录价为例：
 - 本地磁盘，无副本 0.06-0.08 元/GB/月
 - 云盘，SSD 1元/GB/月，高效云盘 0.35 元/GB/月
 - 对象存储单 AZ 版 0.12 元/GB/月，多 AZ 版本 0.15 元/GB/月，低频 0.08 元/GB/月
 - 分布式文件系统，如盘古 HDFS 接口，支持进一步转冷和 EC。
- 生态链：对象存储和类 HDFS 都有足够多的经过生产验证的工具，监控报警层面对象存储的支持更产品化。

2.3. 直写还是转写

方案里，备受瞩目的点在于选择直写还是转写，我认为他们不冲突，两个方案“可以分开有，都可以做强”。

多年来 RocketMQ 运行在基于本地存储的系统中，本地磁盘通常 IOPS 较高，成本较低但可靠性较差，大规模的生产实践中遇到的问题包括但不限于垂直扩容较难，坏盘，宿主机故障等。

- 直写：指使用高可用的存储替换本地块存储，例如使用云盘多点挂载（分布式块存储形态，透明 rdma）或者直写分布式文件系统（下文简称 DFS）作为存储后端，此时主备节点可以共享存储，broker 的高可用中的数据流同步简化为只同步位点，在很大程度上减化了 RocketMQ 高可用的实现。
- 转写：对于大部分数据密集型应用，出于故障恢复的考虑必须实时写日志，意味着无法对数据很好的进行攒批压缩，如果仅使用廉价介质，会带来更高的延迟以及更多的内存使用，无法满足生产需要。一个典型思路就是热数据使用容量小的高速介质先顺序写，compaction 后转储到更廉价的存储系统中。



直写的目的是池化存储，转写的目的是降低数据的长期保存成本，所以我认为一个理想的终态可以是两者的结合。RocketMQ 自己来做数据转冷，那有同学就会提出反问了，如果让 DFS 自身支持透明转冷，岂不是更好？

我的理解是 RocketMQ 希望在转冷这个动作时，能够做一些消息系统内部的格式变化来加速冷数据的读取，减少 IO 次数，配置不同 TTL 等。

相对于通用算法，消息系统自身对如何更好的压缩数据和加速读取的细节更加了解。而且主动转冷的方案在审计和入湖的一些场景下，也可以被用于服务端批量转储数据到不同的平台，到 NoSQL 系统，到 ES，Click House，到对象存储，这一切是如此的自然～

2.4. 技术架构演进

那么分级存储是一个尽善尽美的最终解决方案吗？理想很美好，让我们来看一组典型生产场景的数据。

RocketMQ 在使用块存储时，存储节点存储成本大约会占到 30%-50%。开启分级存储时，由于数据转储会产生一定的计算开销，主要包括数据复制，数据编解码，crc 校验等，不同场景下计算成本会上升 10%-40%，通过换算，我们发现存储节点的总体拥有成本节约了 30% 左右。

考虑到商业和开源技术架构的一致性，选择了先实现转写模式，热数据的存储成本中随着存储空间显著减小，这能够更直接的降低存储成本，在我们充分建设好当前的转写逻辑时再将热数据的 WAL 机制和索引构建移植过来，实现基于分布式系统的直写技术，这种分阶段迭代会更加简明高效，这个阶段我们更加关注通用性和可用性。

- **可移植性：** 直写分布式系统通常需要依赖特定 sdk，配合 rdma 等技术来降低延迟，对应用不完全透明，运维，人力，技术复杂度都有一定上升。保留成熟的本地存储，只需要实现存储插件就可以轻松的切换多种存储后端，不针对 IaaS 做深度绑定在可移植性上会有一定优势。
- **延迟与性能：** 直写模式下存储紧密结合，应用层 ha 的简化也能降低延迟（写多数派成功才被消费者可见），但无论写云盘或者本地磁盘（同区域）延迟都会小于跨可用区的延迟，存储延迟在热数据收发链路不是瓶颈。
- **可用性：** 存储后端往往都有复杂的容错和故障转移策略，直写与转写模式在公有云下可用性都满足诉求。考虑到转写模式下系统是弱依赖二级存储的，更适合开源与非公共云场景。

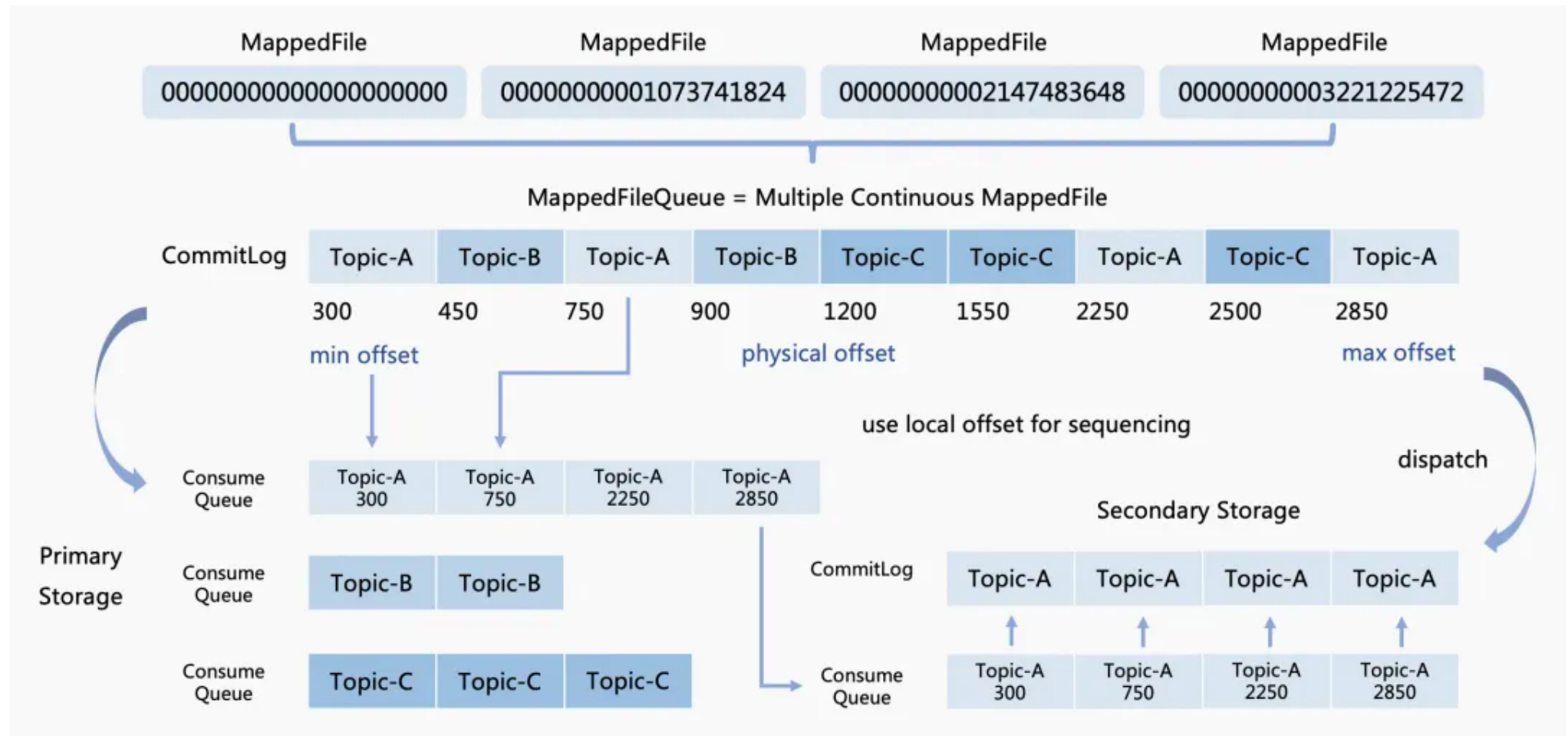
我们为什么不进一步压缩块存储的磁盘容量，做到几乎极致的成本呢？

事实上，在分级存储的场景下，一味的追求过小的本地磁盘容量价值不大。主要有以下原因：

- 故障冗余，消息队列作为基础设施中重要的一环，稳定性高于一切。对象存储本身可用性较高，如果遇到网络波动等问题时，使用对象存储作为主存储，非常容易产生反压导致热数据无法写入，而热数据属于在线生产业务，这对于可用性的影响是致命的。
- 过小的本地磁盘，在价格上没有明显的优势。众所周知，云计算是注重普惠和公平的，如果选用 50G 左右的块存储，又需要等价 150G 的 ESSD 级别的块存储能提供的 IOPS，则其单位成本几乎是普通块存储的数倍。
- 本地磁盘容量充足的情况下，上传时能够更好的通过“攒批”减少对象存储的请求费用。读取时能够对“温热”数据提供更低的延迟和节约读取成本。
- 仅使用对象存储，难以对齐 RocketMQ 当前已经存在的丰富特性，例如用于问题排查的随机消息索引，定时消息特性等，如果为了节约少量成本，极大的削弱基础设施的能力，反向要求业务方自建复杂的中间件体系是得不偿失的。

3. 分级存储的数据模型与实现

3.1. 模型与抽象



RocketMQ 本地存储数据模型如下：

- **MappedFile**: 单个真实文件的句柄，也可以理解为 handle 或者说 fd，通过 mmap 实现内存映射文件。是一个 AppendOnly 的定长字节流语义的 Stream，支持字节粒度的追加写、随机读。每个 MappedFile 拥有自己的类型，写位点，创建更新时间等元数据。
- **MappedFileQueue**: 可以看做是零个或多个定长 MappedFile 组成的链表，提供了流的无边界语义。Queue 中最多只有最后一个文件可以是 Unseal 的状态（可写）。前面的文件都必须都是 Sealed 状态（只读），Seal 操作完成后 MappedFile 是 immutable（不可变）的。
- **CommitLog**: MappedFileQueue 的封装，每个“格子”存储一条序列化的消息到无界的流中。

- ConsumeQueue: 顺序索引, 指向 CommitLog 中消息在 FileQueue 中的偏移量 (offset)。

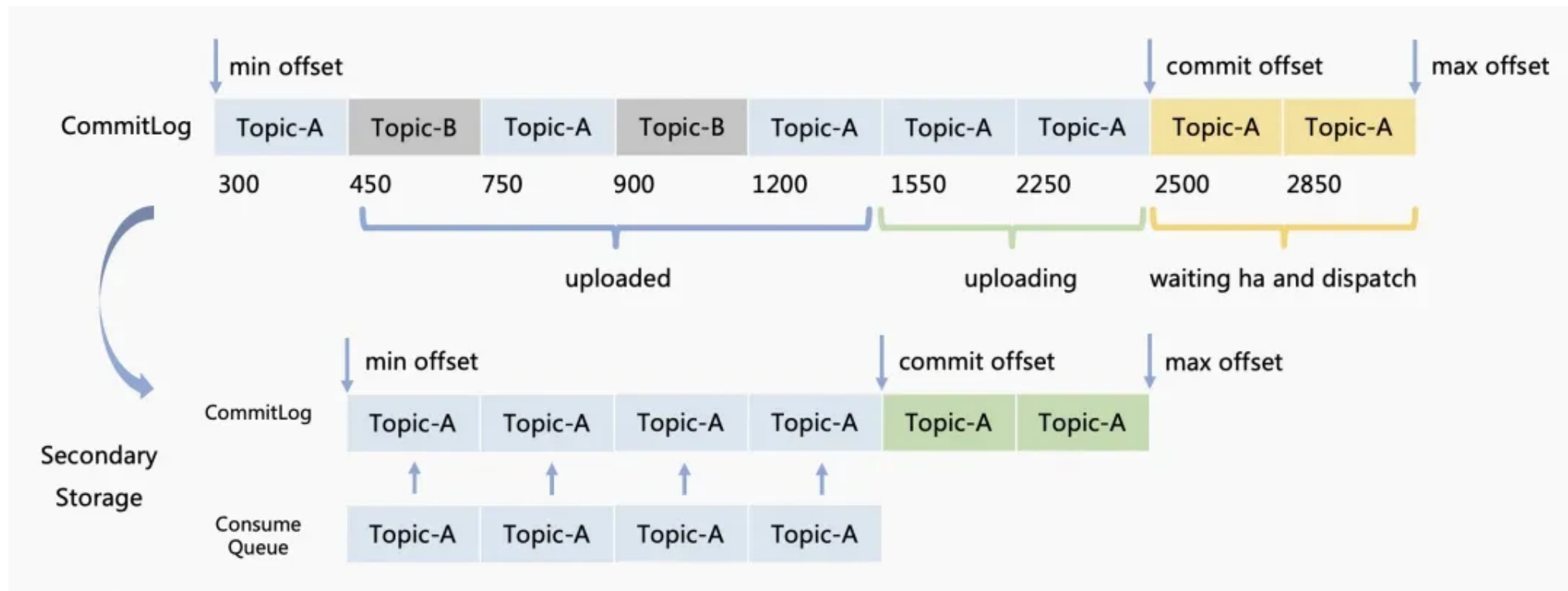
RocketMQ 分级存储提供的数据模型和本地模型类似, 改变了 CommitLog 和 ConsumeQueue 的概念:

- TieredFileSegment: 和 MappedFile 类似, 描述一个分级存储系统中文件的句柄。
- TieredFlatFile: 和 MappedFileQueue 类似。
- TieredCommitLog 和本地 CommitLog 混合写不同, 按照单个 Topic 单个队列的粒度拆分多条 CommitLog。
- TieredConsumeQueue 指向 TieredCommitLog 偏移量的一个索引, 是严格连续递增的。实际索引的位置会从指向的 CommitLog 的位置改为 TieredCommitLog 的偏移量。
- CompositeFlatFile: 组合 TieredCommitLog 和 TieredConsumeQueue 对象, 并提供概念的封装。

3.2. 消息上传流程

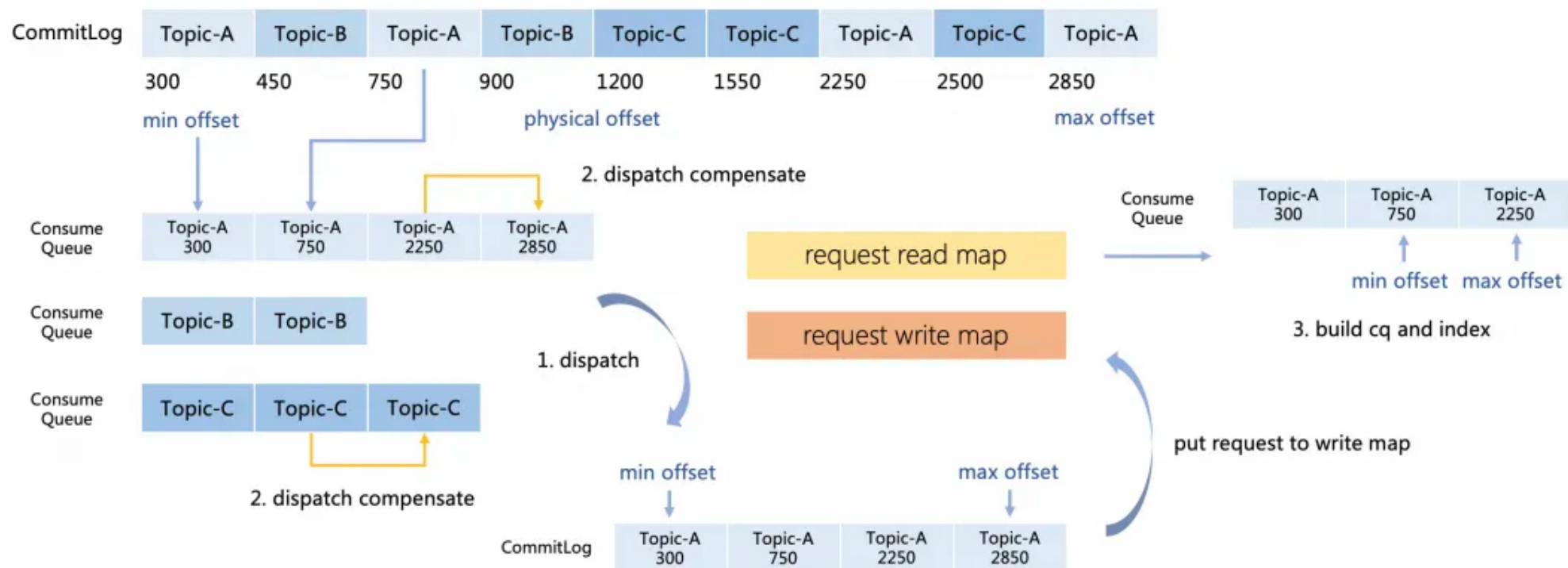
RocketMQ 的存储实现了一个 Pipeline, 类似于拦截器链, Netty 的 handler 链, 读写请求会经过这个 Pipeline 的多个处理器。

Dispatcher 的概念是指为写入的数据构建索引, 在分级存储模块初始化时, 会创建 TieredDispatcher 注册为 CommitLog 的 dispatcher 链的一个处理器。每当有消息发送到 Broker 会调用 TieredDispatcher 进行消息分发。下面我们来追踪单条消息进入存储层的流程:



1. 消息被顺序追加到本地 commitlog 并更新本地 max offset（图中黄色部分），为了防止宕机时多副本产生“读摆动”，多副本中多数派的最小位点会作为“低水位”被确认，这个位点被称为 commit offset（图中 2500）。换句话说，commit offset 与 max offset 之间的数据是正在等待多副本同步的。
2. 当 commit offset \geq message offset 之后，消息会被上传到二级存储的 commitlog 的缓存中（绿色部分）并更新这个队列的 max offset。
3. 消息的索引会被追加到这个队列的 consume queue 中并更新 consume queue 的 max offset。
4. 一旦 commitlog 中缓存大小超过阈值或者等待达到一定时间，消息的缓存将被上传至 commitlog，之后才会将索引信息提交，这里有一个隐含的数据依赖，使索引被晚于原始数据更新。这个机制保证了所有 cq 索引中的数据都能在 commitlog 中找到。宕机场景下，分级存储中的 commitlog 可能会重复构建，此时没有 cq 指向这段数据。由于文件本身还是被使用 Queue 的模型管理的，使得整段数据在达到 TTL 时能被回收，此时并不会产生数据流的“泄漏”。
5. 当索引也上传完成的时候，更新分级存储中的 commit offset（绿色部分被提交）。
6. 系统重启或者宕机时，会选择多个 dispatcher 的最小位点向 max offset 重新分发，确保数据不丢失。

在实际执行中，上传部分由三组线程协同工作。



1. store dispatch 线程，由于该线程负责本地 cq 的分发，我们不能长时间阻塞该线程，否则会影响消息进入本地存储的“可见性延迟”。因此 store dispatch 每次只会尝试对拆分后的文件短暂加锁，如果加锁成功，将消息数据放入拆分后的 commitlog 文件的缓冲区则立即退出，该操作不会阻塞。若获取锁失败则立即返回。
2. store compensate 线程组，负责对本地 cq 进行定时扫描，当写入压力较高时，步骤 1 可能获取锁失败，这个环节会批量的将落后的数据放入 commitlog 中。原始数据被放入后会将 dispatch request 放入 write map。
3. build cq index 线程。write map 和 read map 是一个双缓冲队列的设计，该线程负责将 read map 中的数据构建 cq 并上传。如果 read map 为空，则交换缓冲区，这个双缓冲队列在多个线程共享访问时减少了互斥和竞争操作。

各类存储系统的缓冲攒批策略大同小异，而线上的 topic 写入流量往往是存在热点的，根据经典的二八原则，RocketMQ 分级存储模块目前采用了“达到一定数据量”，“达到一定时间”两者取其小的合并方式。

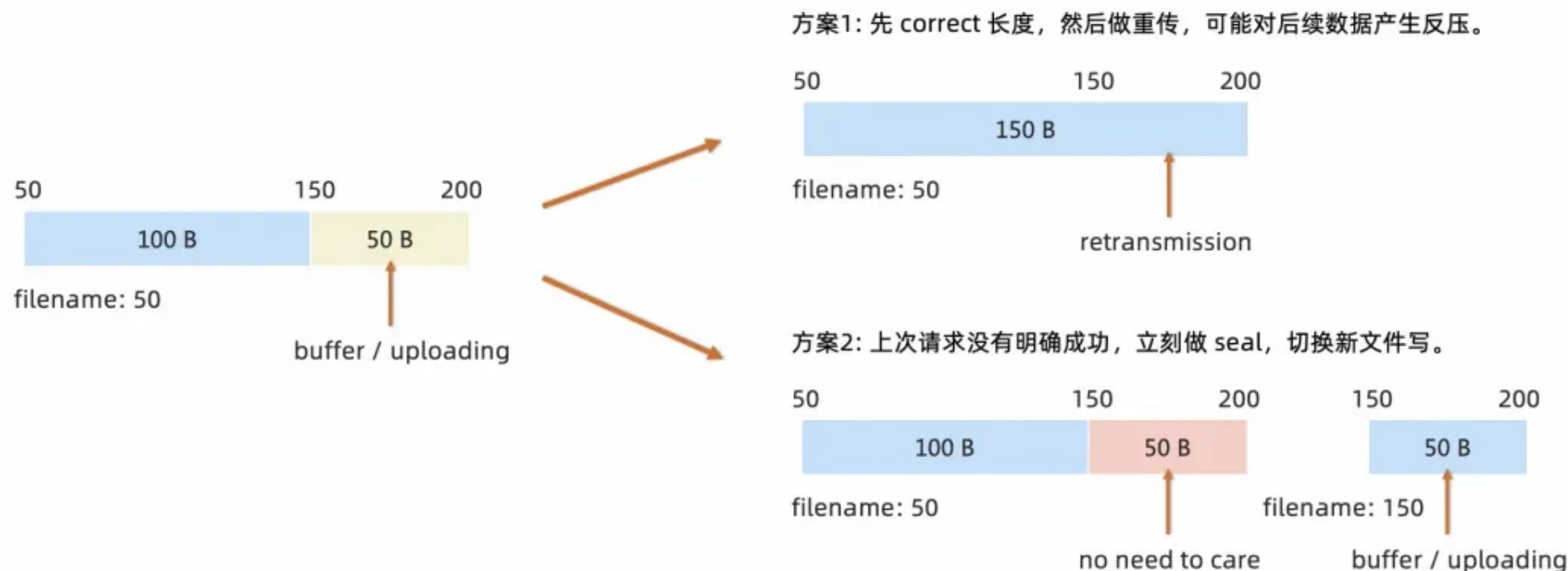
这种方式简单可靠，对于大流量的 topic 很容易就可以达到批的最小数据量，对于流量较低的 topic 也不会占用过多的内存。从而减少了对对象存储的请求数，其开销主要包括 restful 协议请求头，签名和传输等。诚然，攒批的逻辑仍然存在较大的优化空间，例如 IOT，数据分片同步等各个 topic 流量较为平均的场景使用类似“滑动窗口”的加权平均算法，或者基于信任值的流量控制策略可以更好的权衡延迟和吞吐。

3.3. Non-StopWrite 特性

Non-StopWrite 模型实际上是一致性模型的一部分。实际生产中，后端分布式存储系统的断连和网络问题偶尔会不可避免，而 Append 模型实际上一致性顺序的模型，参考 HDFS 的 2-3 异步写，我们提出了一种基于 **Append** 和 **Put** 的混合模型。

例如：对于如下图片中的 stream，commit / confirm offset = 150，max offset = 200。此时写出缓冲区中的数据包括 150-200 的 uncommitted 部分，还有 200 以后源源不断的写入的新数据。

RocketMQ non-stop write feature



假设后端存储系统支持原子性写入, 单个上传请求的数据内容是 150-200 这个区间, 当单次上传失败时, 我们需要向服务端查询上一次写入的位点并进行错误处理。

- 如果返回的长度是 150, 说明上传失败, 应用需要重传 buffer。
- 如果返回的长度是 200, 说明前一次上传成功但没有收到成功的 response, 提升 commit offset 至 200。

而另一种解决方案是, 使用 Non-StopWrite 机制立刻新切换一个文件, 以 150 作为文件名, 立刻重传 150 至 200 的数据, 如果有新的数据也可以立刻与这些数据一起上传, 我们发现混合模型存在显著优势:

- 对于绝大部分没有收到成功的响应时, 上传是失败的而不是超时, 立刻切换文件可以不去 check in 文件长度, 减少 rpc 数量。

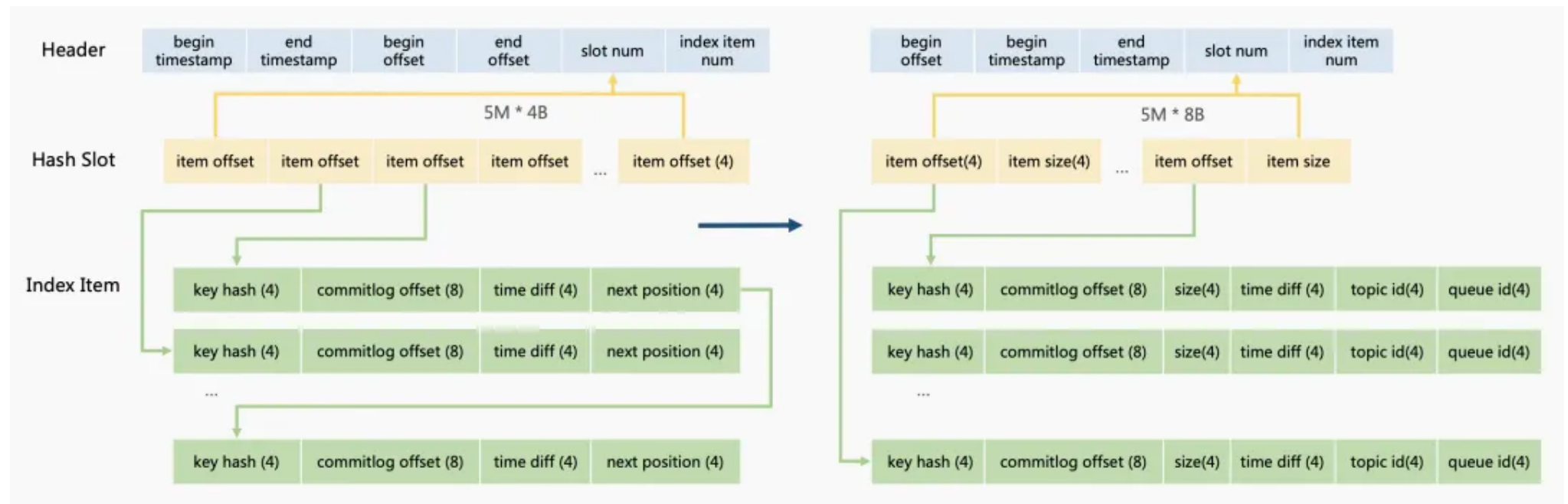
- 立刻重传不会阻塞后续新的数据上传，不容易由于后端数据无法写出造成反压，导致前端写失败。
- 无论 150-200 这段数据在第一个文件是到底是写成功还是失败都无关紧要，因为不会去读取这段数据。尤其是对于不支持请求粒度原子写入的模型来说，如果上一次请求的结果是 180，那么错误处理将会非常复杂。

3.4. 随机索引重排

21 年的时候，我第一次听到用“读扩散”或者“写扩散”来描述一个设计方案，这两个词简洁的概括了应用性能设计的本质。各种业务场景下，我们总是选择通过读写扩散，选择通过格式的变化，将数据额外转储到一份性能更好或者更廉价的存储，或者通过读扩散减少数据冗余（减少索引提高了平均查询代价）。

RocketMQ 会在先内存构建基于 hash 的持久化索引文件 IndexFile（非 AppendOnly），再通过 mmap 异步的将数据持久化到磁盘。这个文件是为了支持用户通过 key，消息 ID 等信息来追踪一条消息。

对于单条消息会先计算 $\text{hash}(\text{topic\#key}) \% \text{slot_num}$ 选择 hash slot（黄色部分）作为随机索引的指针，对象索引本身会附加到 index item 中，hash slot 使用“哈希拉链”的方式解决冲突，这样便形成了一条当前 slot 按照时间存入的倒序的链表。不难发现，查询时需要多次随机读取链表节点。



由于冷存储的 IOPS 代价是非常昂贵的，在设计上我们希望可以面向查询进行优化。新的文件结构类似于维护没有 GC 和只有一次 compaction 的 LSM

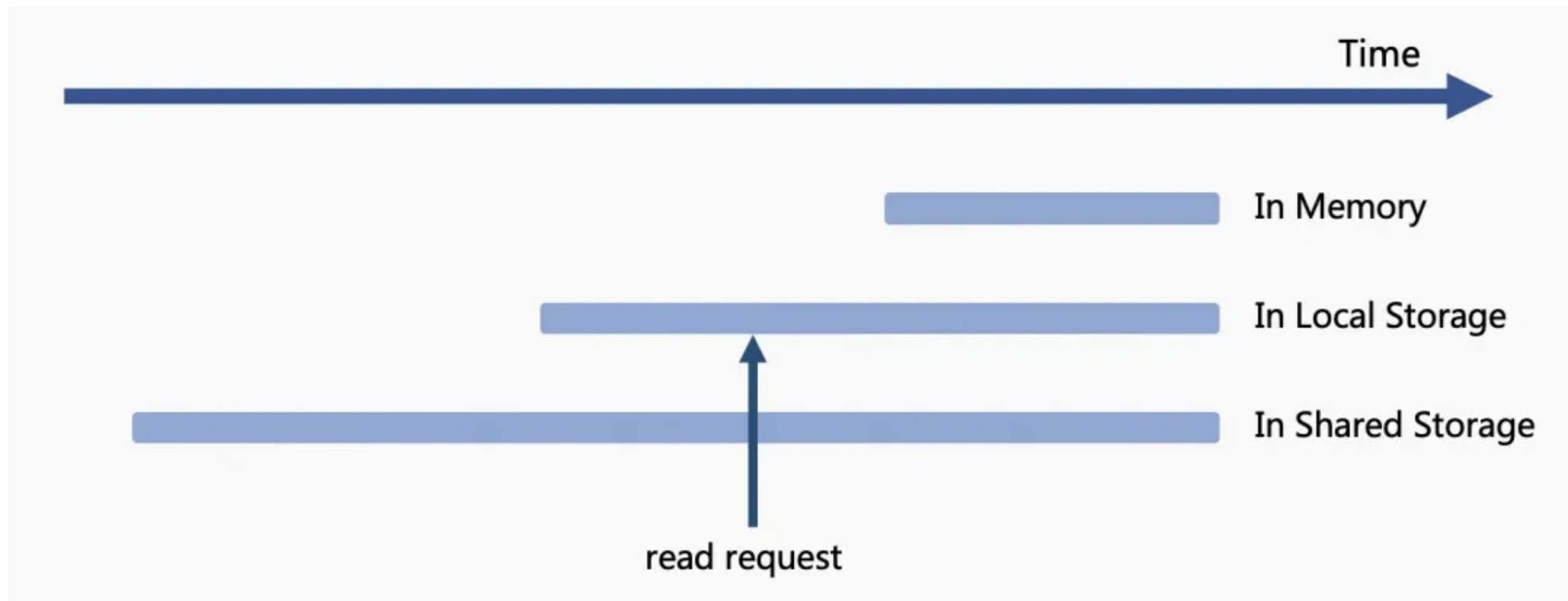
树，数据结构的调整如下：

1. 等待本地一个 IndexFile 完全写满，规避修改操作，在高 IOPS 的存储介质上异步 compaction，完成后删除原来的文件。
2. 从冷存储查询延迟高，而单次返回的数据量大小（不太大的场景）并不会明显改变延迟。compaction 时优化数据结构，做到用一次查询连续的一段数据替换多次随机点查。
3. hash slot 的指向的 List 是连续的，查询时可以根据 hash slot 中的 item offset 和 item size 一次取出所有 hashcode 相同的记录并在内存中过滤。

3.5. 消息读取流程

3.5.1 读取策略

读取是写入的逆过程，优先从哪里取回想要的数据必然存在很多的工程考虑与权衡。如图所示，近期的数据被缓存在内存中，稍久远的数据存在与内存和二级存储上，更久远的数据仅存在于二级存储。当被访问的数据存在于内存中，由于内存的速度快速存储介质，直接将这部分数据通过网络写会给客户端即可。如果被访问的数据如图中 request 的指向，存在于本地磁盘又存在于二级存储，此时应该根据一二级存储的特性综合权衡请求落到哪一层。



有两种典型的想法：

1. 数据存储被视为多级缓存，越上层的介质随机读写速度快，请求优先向上层存储进行查询，当内存中不存在了就查询本地磁盘，如果还不存在才向二级存储查询。
2. 由于在转冷时主动对数据做了 compaction，从二级存储读取的数据是连续的，此时可以把更宝贵一级存储的 IOPS 留给在线业务。

RocketMQ 的分级存储将这个选择抽象为了读取策略，通过请求中的逻辑位点（queue offset）判断数据处于哪个区间，再根据具体的策略进行选择：

- **DISABLE**：禁止从多级存储中读取消息，可能是数据源不支持。
- **NOT_IN_DISK**：不在一级存储的消息都会从二级存储中读取。
- **NOT_IN_MEM**：不在内存中的消息即冷数据从多级存储读取。
- **FORCE**：强制所有消息从多级存储中读取，目前仅供测试使用。

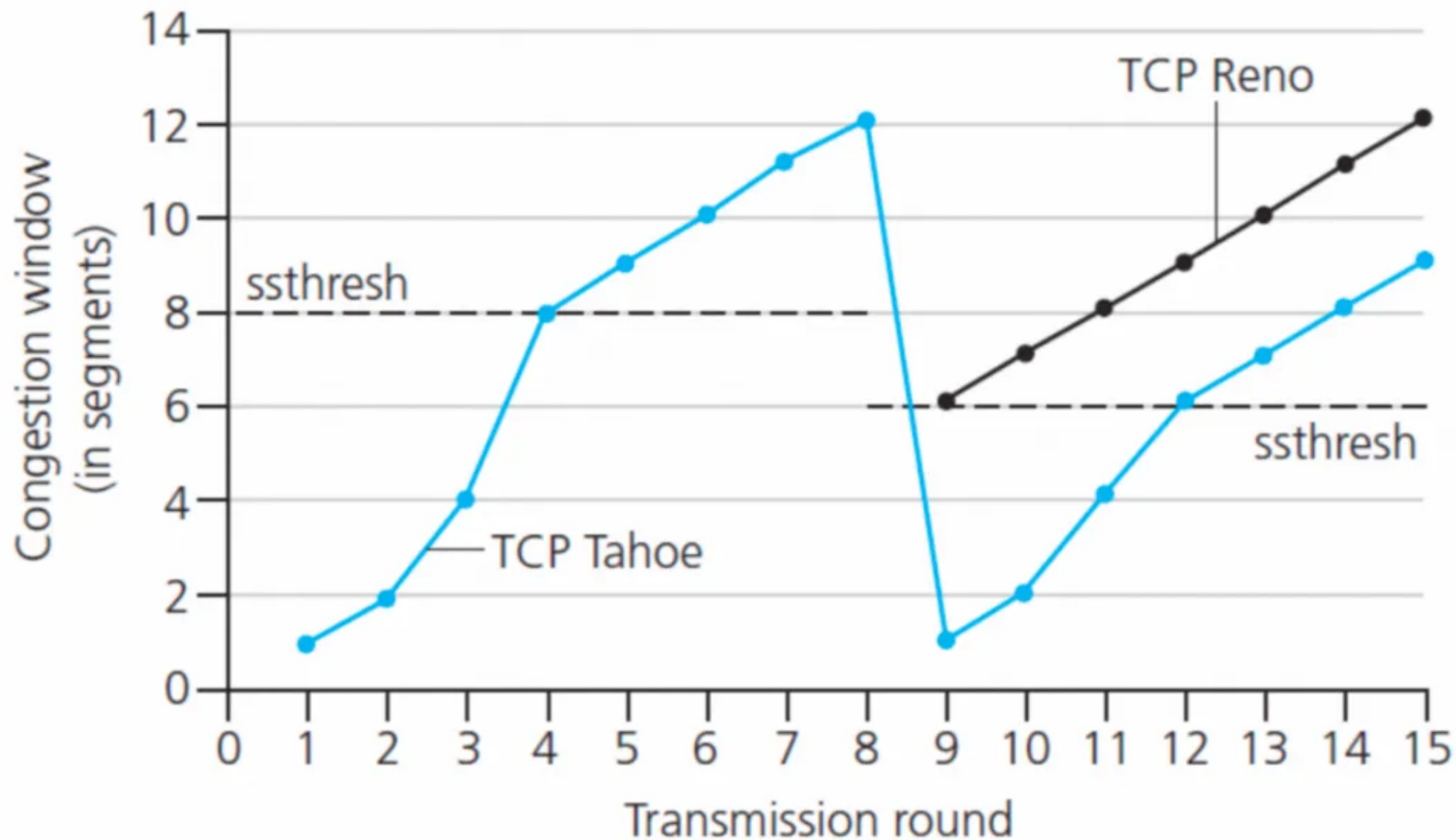
3.5.2 预读设计

TieredMessageFetcher 是 RocketMQ 分级存储取回数据的具体实现。

为了加速从二级存储读取的速度和减少整体上对二级存储请求数，采用了预读缓存的设计：

即 TieredMessageFetcher 读取消息时会预读更多的消息数据，预读缓存的设计参考了 TCP Tahoe 拥塞控制算法，每次预读的消息量类似拥塞窗口采用加法增、乘法减的流量控制机制。

- 加法增：从最小窗口开始，每次增加等同于客户端 batchSize 的消息量。
- 乘法减：当缓存的消息超过了缓存过期时间仍未被全部拉取，此时一般是客户端缓存满，消息数据反压到服务端，在清理缓存的同时会将下次预读消息量减半。
- 此外，在客户端消费速度较快时，向二级存储读取的消息量较大，此时会使用分段策略并发取回数据。



3.6. 定时消息的分级存储

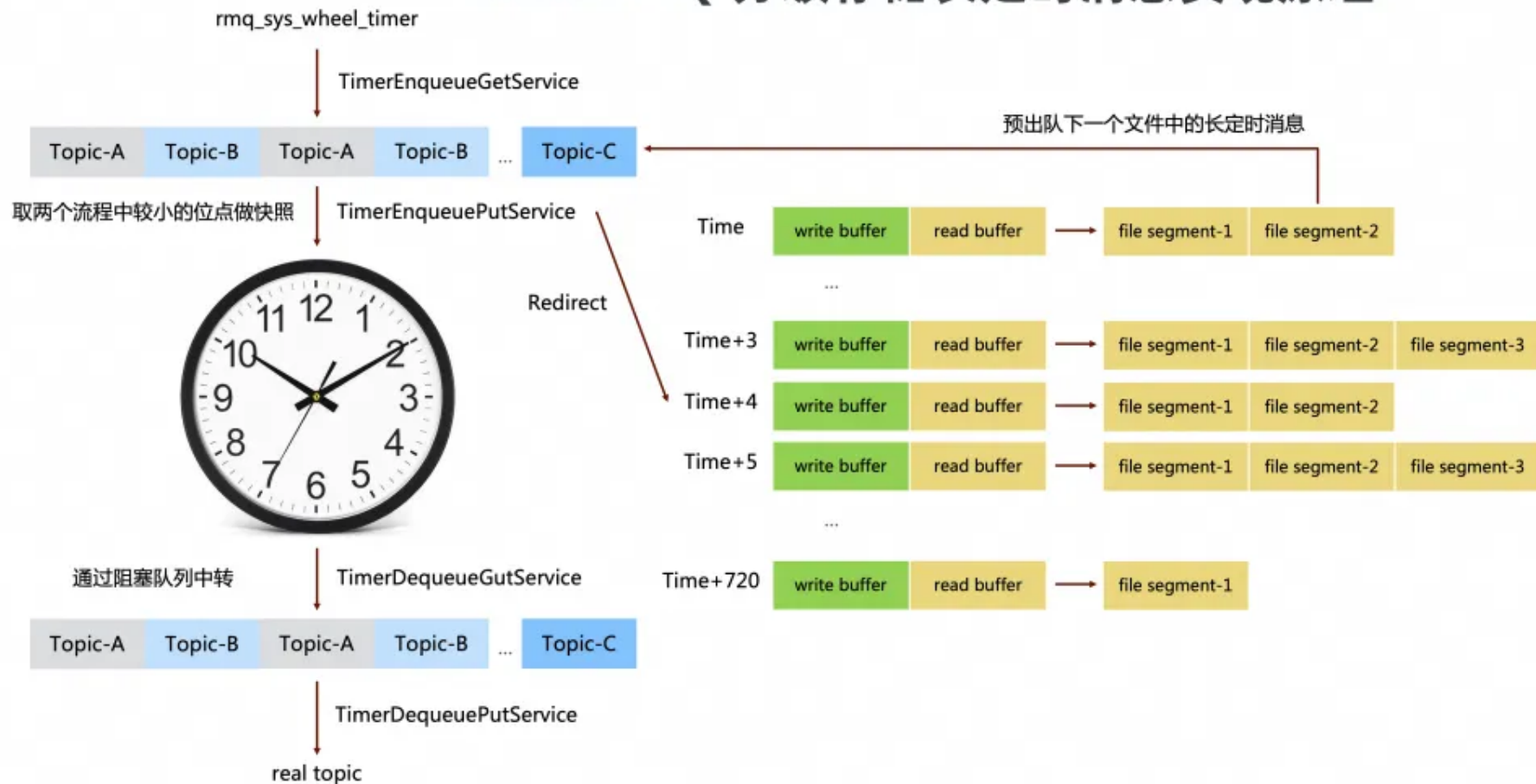
除了普通消息，RocketMQ 支持设置未来几十天的长定时消息，而这部分数据严重挤占了热数据的存储空间。

RocketMQ 实现了基于本地文件系统的时间轮，整体设计如左侧所示。单节点上所有的定时消息会先写入 `rmq_sys_wheel_timer` 的系统 topic，进入时间轮，出队后这些消息的 topic 会被还原为真实的业务 topic。

“从磁盘读取数据”和“将消息索引放入时间轮”这两个动作涉及到 IO 与计算，为了减少这两个阶段的锁竞争引入了 `Enqueue` 作为中转的等待队列，`EnqueueGet` 和 `EnqueuePut` 分别负责写入和读取数据，这个设计简单可靠。



RocketMQ 分级存储长定时消息实现原理

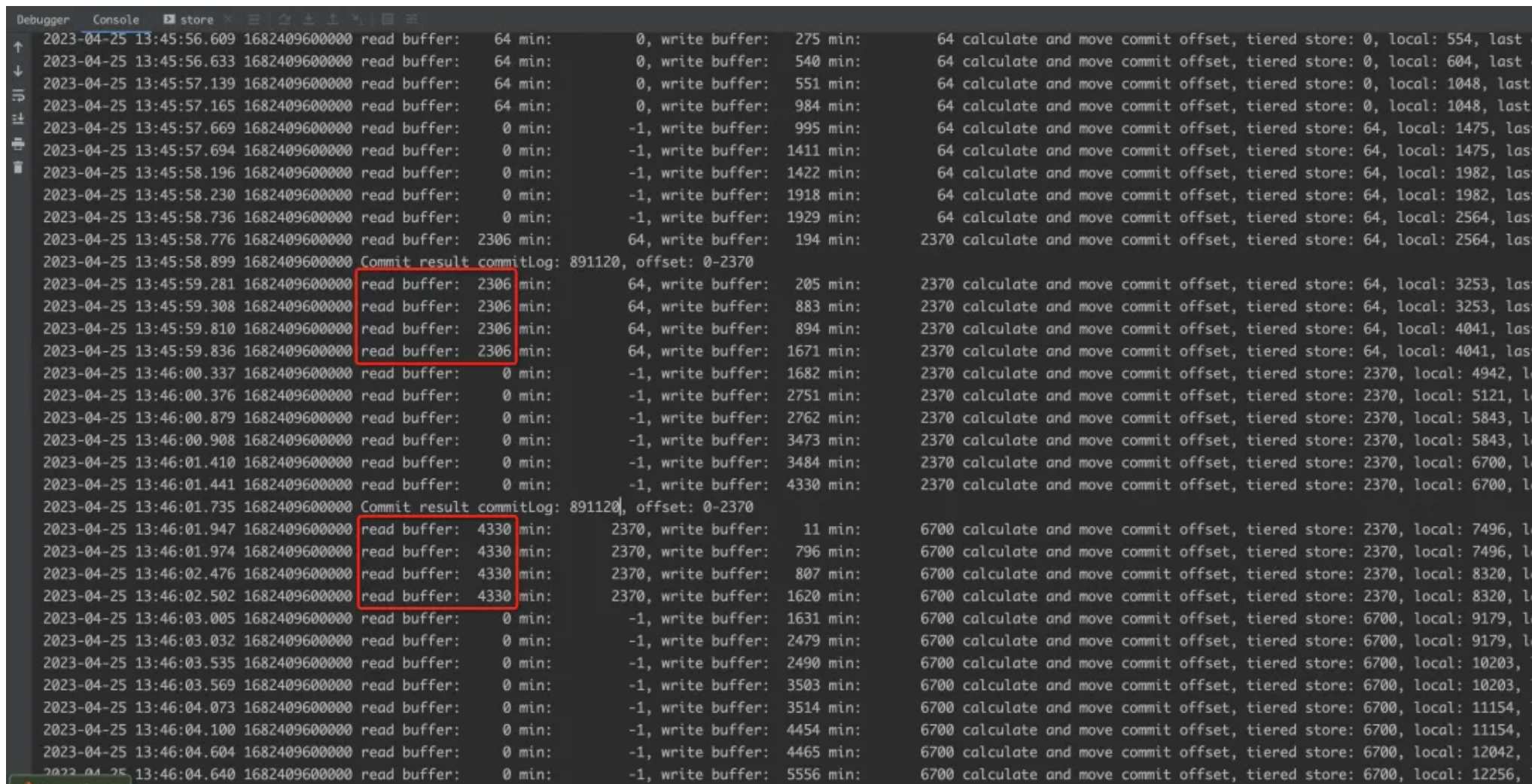


不难发现，所有的消息都会进入时间轮，这也是挤占存储空间的根本原因。

- 写入时，RocketMQ 的分级存储定时消息针对 EnqueuePut 做了一个分流，对于大于当前时间数小时的消息会被写入到基于分级存储的 TimerFlatFile 文件中，我们维护了一个 `ConcurrentSkipListMap<Long /* timestamp */, TimerFlatFile> timerFlatFileTable`；每间隔 1 小时，设置一个 TimerFlatFile，对于 $T+n$ 至 $T+n+1$ 的定时消息，会先被混合追加到 $T+n$ 所对应的文件中。
- 读取时，当前时间 + 1 小时的消息将被提前出队，这些消息又会重新进入本地 TimerStore 的系统 topic 中/此时，由于定时时间都是将来一小段时间的，他们不再会进入时间轮的结构中。

在这个设计上有一些工程性的考虑：

- timerFlatFileTable 中的 Key 很多，会不会让分级存储上的数据碎片化？分布式文件系统底层一般使用类 LSM 结构，RocketMQ 只关心 LBA 结构，可以通过优化 Enqueue 的 buffer 让写分级存储时数据达到攒批的效果。
- 可靠的位点，Enqueue 到“时间轮”和 timerFlatFileTable 可以共用一个 commit offset。对于单条消息来说，只要它进入时间轮或者被上传成功，我们就认为一条消息已经持久化了。由于更新到二级存储本身需要一些攒批缓冲的过程，会延迟 commit offset 的更新，但是这个缓冲时间是可控的。
- 我们发现偶尔本地存储转储到二级存储会较慢，使用双缓冲队列实现读写分离（如图片中绿色部分）此时消息被放入写缓存，随后转入读缓存队列，最后进入上传流程。



2023-04-25 13:45:56.609	1682409600000	read buffer: 64 min: 0, write buffer: 275 min: 64	calculate and move commit offset, tiered store: 0, local: 554, last
2023-04-25 13:45:56.633	1682409600000	read buffer: 64 min: 0, write buffer: 540 min: 64	calculate and move commit offset, tiered store: 0, local: 604, last
2023-04-25 13:45:57.139	1682409600000	read buffer: 64 min: 0, write buffer: 551 min: 64	calculate and move commit offset, tiered store: 0, local: 1048, last
2023-04-25 13:45:57.165	1682409600000	read buffer: 64 min: 0, write buffer: 984 min: 64	calculate and move commit offset, tiered store: 0, local: 1048, last
2023-04-25 13:45:57.669	1682409600000	read buffer: 0 min: -1, write buffer: 995 min: 64	calculate and move commit offset, tiered store: 64, local: 1475, last
2023-04-25 13:45:57.694	1682409600000	read buffer: 0 min: -1, write buffer: 1411 min: 64	calculate and move commit offset, tiered store: 64, local: 1475, last
2023-04-25 13:45:58.196	1682409600000	read buffer: 0 min: -1, write buffer: 1422 min: 64	calculate and move commit offset, tiered store: 64, local: 1982, last
2023-04-25 13:45:58.230	1682409600000	read buffer: 0 min: -1, write buffer: 1918 min: 64	calculate and move commit offset, tiered store: 64, local: 1982, last
2023-04-25 13:45:58.736	1682409600000	read buffer: 0 min: -1, write buffer: 1929 min: 64	calculate and move commit offset, tiered store: 64, local: 2564, last
2023-04-25 13:45:58.776	1682409600000	read buffer: 2306 min: 64, write buffer: 194 min: 2370	calculate and move commit offset, tiered store: 64, local: 2564, last
2023-04-25 13:45:58.899	1682409600000	Commit result commitLog: 891120, offset: 0-2370	
2023-04-25 13:45:59.281	1682409600000	read buffer: 2306 min: 64, write buffer: 205 min: 2370	calculate and move commit offset, tiered store: 64, local: 3253, last
2023-04-25 13:45:59.308	1682409600000	read buffer: 2306 min: 64, write buffer: 883 min: 2370	calculate and move commit offset, tiered store: 64, local: 3253, last
2023-04-25 13:45:59.810	1682409600000	read buffer: 2306 min: 64, write buffer: 894 min: 2370	calculate and move commit offset, tiered store: 64, local: 4041, last
2023-04-25 13:45:59.836	1682409600000	read buffer: 2306 min: 64, write buffer: 1671 min: 2370	calculate and move commit offset, tiered store: 64, local: 4041, last
2023-04-25 13:46:00.337	1682409600000	read buffer: 0 min: -1, write buffer: 1682 min: 2370	calculate and move commit offset, tiered store: 2370, local: 4942, last
2023-04-25 13:46:00.376	1682409600000	read buffer: 0 min: -1, write buffer: 2751 min: 2370	calculate and move commit offset, tiered store: 2370, local: 5121, last
2023-04-25 13:46:00.879	1682409600000	read buffer: 0 min: -1, write buffer: 2762 min: 2370	calculate and move commit offset, tiered store: 2370, local: 5843, last
2023-04-25 13:46:00.908	1682409600000	read buffer: 0 min: -1, write buffer: 3473 min: 2370	calculate and move commit offset, tiered store: 2370, local: 5843, last
2023-04-25 13:46:01.410	1682409600000	read buffer: 0 min: -1, write buffer: 3484 min: 2370	calculate and move commit offset, tiered store: 2370, local: 6700, last
2023-04-25 13:46:01.441	1682409600000	read buffer: 0 min: -1, write buffer: 4330 min: 2370	calculate and move commit offset, tiered store: 2370, local: 6700, last
2023-04-25 13:46:01.735	1682409600000	Commit result commitLog: 891120, offset: 0-2370	
2023-04-25 13:46:01.947	1682409600000	read buffer: 4330 min: 2370, write buffer: 11 min: 6700	calculate and move commit offset, tiered store: 2370, local: 7496, last
2023-04-25 13:46:01.974	1682409600000	read buffer: 4330 min: 2370, write buffer: 796 min: 6700	calculate and move commit offset, tiered store: 2370, local: 7496, last
2023-04-25 13:46:02.476	1682409600000	read buffer: 4330 min: 2370, write buffer: 807 min: 6700	calculate and move commit offset, tiered store: 2370, local: 8320, last
2023-04-25 13:46:02.502	1682409600000	read buffer: 4330 min: 2370, write buffer: 1620 min: 6700	calculate and move commit offset, tiered store: 2370, local: 8320, last
2023-04-25 13:46:03.005	1682409600000	read buffer: 0 min: -1, write buffer: 1631 min: 6700	calculate and move commit offset, tiered store: 6700, local: 9179, last
2023-04-25 13:46:03.032	1682409600000	read buffer: 0 min: -1, write buffer: 2479 min: 6700	calculate and move commit offset, tiered store: 6700, local: 9179, last
2023-04-25 13:46:03.535	1682409600000	read buffer: 0 min: -1, write buffer: 2490 min: 6700	calculate and move commit offset, tiered store: 6700, local: 10203, last
2023-04-25 13:46:03.569	1682409600000	read buffer: 0 min: -1, write buffer: 3503 min: 6700	calculate and move commit offset, tiered store: 6700, local: 10203, last
2023-04-25 13:46:04.073	1682409600000	read buffer: 0 min: -1, write buffer: 3514 min: 6700	calculate and move commit offset, tiered store: 6700, local: 11154, last
2023-04-25 13:46:04.100	1682409600000	read buffer: 0 min: -1, write buffer: 4454 min: 6700	calculate and move commit offset, tiered store: 6700, local: 11154, last
2023-04-25 13:46:04.604	1682409600000	read buffer: 0 min: -1, write buffer: 4465 min: 6700	calculate and move commit offset, tiered store: 6700, local: 12042, last
2023-04-25 13:46:04.640	1682409600000	read buffer: 0 min: -1, write buffer: 5556 min: 6700	calculate and move commit offset, tiered store: 6700, local: 12256, last

4. 分级存储企业级竞争力

4.1. 冷数据的压缩与归档

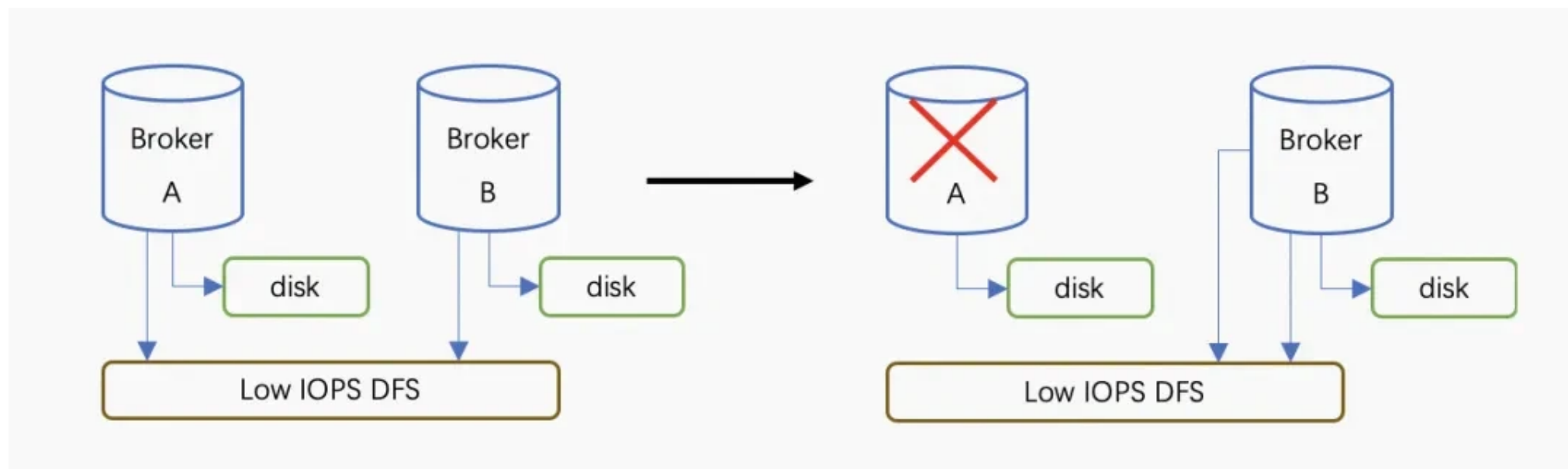
压缩是一种经典的时间与空间交换的权衡，其目的在于通过较小的 CPU 开销，实现更少的磁盘占用或网络 I/O 传输。目前，RocketMQ 的热存储在考虑延迟的情况下，仅对单条大于 4K 的消息进行单条压缩存储。对于冷存储服务其实可以做两个层面的压缩与归档处理。

- 消息队列业务层面，对于大多数业务 Topic，其 Body 通常存在相似性，可将其压缩至原大小的几分之一至几十分之一。
- 底层存储层面，使用 EC 纠删码，数据被分成若干个数据块，然后再根据一定的算法，生成一些冗余块。当数据丢失时，可以使用其余的数据块和冗余块来恢复丢失的数据块，从而保证数据的完整性和可靠性。典型的 EC 算法后存储空间的使用可以降低到 1.375 副本。

业界也有一些基于 FPGA 实现存储压缩加速的案例，我们将持续探索这方面的尝试。

4.2. 原生的只读挂载能力实现 Serverless

业界对 Serverless 有不同的理解，过去 RocketMQ 多节点之间不共享存储，导致“扩容快，缩容慢”，例如 A 机器需要下线，则必须等普通消息消费完，定时消息全部出队才能进行运维操作。分级存储设计通过 shared-disk 的方式实现跨节点代理读取下线节点的数据，如右图所示：A 的数据此时可以被 B 节点读取，彻底释放了 A 的计算资源和一级存储资源。



这种缩容的主要流程如下：

1. RocketMQ 实现了一个简单的选举算法，正常情况下集群内每一个节点都持有对自己数据独占的写锁。
2. 待下线的节点做优雅下线，确保近期定时消息，事务消息，pop retry 消息都已被完整处理。上传自己的元数据信息到共享的二级存储，并释放自己的写锁。

3. 集群使用一定的负载均衡算法，新的节点获取写锁，将该 Broker 的数据以只读的形式挂载。
4. 将原来节点的元数据注册到 NameServer 对客户端暴露。
5. 对于原节点的写请求，例如位点更新，将在内存中处理并周期性快照到共享存储中。

5. 总结

RocketMQ 的存储在云原生时代的演进中遇到了更多有趣的场景和挑战，这是一个需要全链路调优的复杂工程。出于可移植性和通用性的考虑，我们还没有非常有效的使用 DPDK + SPDK + RDMA 这些新颖的技术，但我们解决了许多工程实践中会遇到的问题并构建了整个分级存储的框架。在后续的发展中，我们会推出更多的存储后端实现，针对延迟和吞吐量等细节做深度优化。

参考文档： [1] Chang, F., Dean, J., Ghemawat, S., et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems, 2008, 26(2): 4.

[2] Liu, Y., Zhang, K., & Spear, M. Dynamic-Sized Nonblocking Hash Tables. In Proceedings of the ACM Symposium on Principles of Distributed Computing, 2014.

[3] Ongaro, D., & Ousterhout, J. In Search of an Understandable Consensus Algorithm. Proceedings of the USENIX Conference on Operating Systems Design and Implementation, 2014, 305-320.

[4] Apache RocketMQ. GitHub, <https://github.com/apache/rocketmq>

[5] Verbitski, A., Gupta, A., Saha, D., et al. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In Proceedings of the 2018 International Conference on Management of Data, 2018, 789-796.

[6] Antonopoulos, P., Budovski, A., Diaconu, C., et al. Socrates: The new sql server in the cloud. In Proceedings of the 2019 International Conference on Management of Data, 2019, 1743-1756.

[7] Li, Q. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. Fast 2023
<https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed>

[8] Lu, S. Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems. Fast 2023