

特别加餐 | 高性能检索系统中的设计漫谈

2020-05-04 陈东

检索技术核心20讲

[进入课程 >](#)




讲述：陈东

时长 18:09 大小 16.64M



你好，我是陈东。欢迎来到检索专栏的第三次加餐时间。

在进阶篇的讲解过程中，我们经常会提起一些设计思想，包括索引与数据分离、减少磁盘IO、读写分离和分层处理等方案。这些设计思想看似很简单，但是应用非常广泛，在许多复杂的高性能系统中，我们都能看到类似的设计和实现。不过，前面我们并没有深入来讲，你可能理解得还不是很透彻。

所以，今天我会把专栏中出现过的相关案例进行汇总和对比，再结合相应的案例扩展， 进一步的分析讨论，来帮助你更好地理解这些设计思想的本质。并且，我还会总结出一些可以参考的通用经验，让你能更好地设计和实现自己的高性能检索系统。

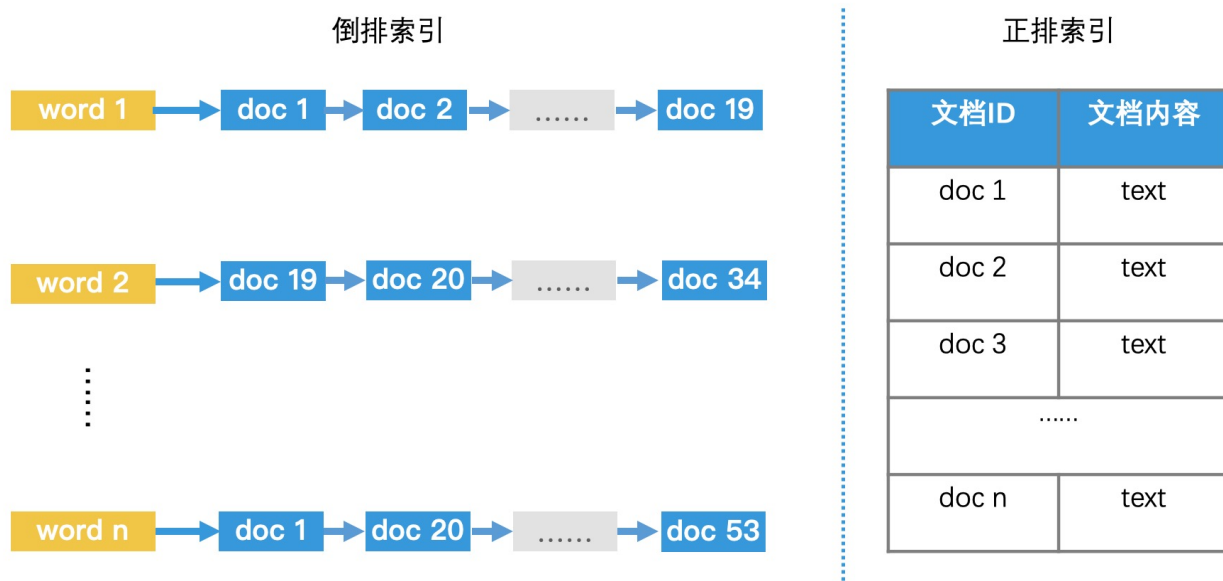
设计思想一：索引与数据分离

我要说的第一个设计思想就是索引与数据分离。索引与数据分离是一种解耦的设计思想，它能帮助我们更好地聚焦在索引的优化上。

比如说，对于无法完全加载到内存中的数据，对它进行索引和数据分离之后，我们就可以利用内存的高性能来加速索引的访问了。🔗第 6 讲中的线性索引的设计，以及 B+ 树中区分中间节点和叶子节点的设计，就都使用了索引和数据分离的设计思想。

那如果索引和数据都可以加载在内存中了，我们还需要使用索引和数据分离吗？在这种情况下，将索引和数据分离，我们依然可以提高检索效率。以🔗第 5 讲中查找唐诗的场景为例，我们将所有的唐诗存在一个正排索引中，然后以关键词为 Key 建立倒排索引。倒排索引中只会记录唐诗的 ID，不会记录每首唐诗的内容。这样做会有以下 3 个优点。

1. 节约存储空间，我们不需要在 posting list 中重复记录唐诗的内容。
2. 减少检索过程中的复制代价。在倒排索引的检索过程中，我们需要将 posting list 中的元素进行多次比较和复制等操作。如果每个元素都存了复杂的数据，而不是简单的 ID，那复制代价我们也不能忽略。
3. 保持索引的简单有效，让我们可以使用更多的优化手段加速检索过程。在🔗加餐 1 中我们讲过，如果 posting list 中都存着简单 ID 的话，我们可以将 posting list 转为位图来存储和检索，以及还可以使用 Roaring Bitmap 来提高检索效率。



总结来说就是，索引与数据分离的设计理念可以让索引保持简洁和高效，来帮助我们聚焦在索引的优化技术上。因此，保持索引的简洁高效是我们需要重点关注的。

当然，我相信你刚开始设计一个新系统的时候，可以很容易做到这一点。但也正因为它非常基础，恰恰就成了我们工作中最容易忽视的地方。

而一旦我们忽视了，那随着系统的变化，索引中掺杂的数据越来越多、越来越复杂，系统的检索性能就会慢慢下降。这时，我们需要牢记**奥卡姆剃刀原理，也就是“如无必要，勿增实体”这个基础原则**，来保证索引一直处于简洁高效的状态。

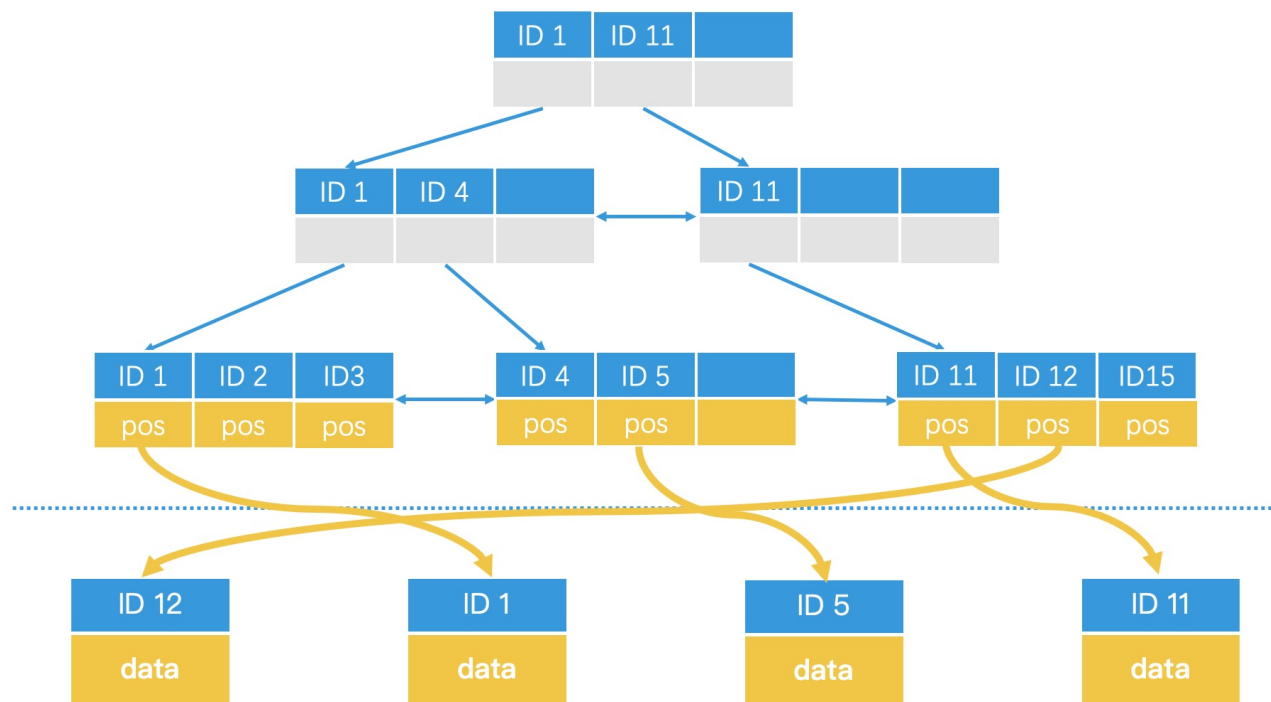
当然，索引和数据分离也会带来一些弊端，如不一致性。怎么理解呢？我们可以考虑这么一个场景：数据已经修改或是删除了，但索引还没来得及更新。如果这个时候我们访问索引，那得到的结果就很有可能是错误的。

那这个错误的结果会造成什么影响呢？这就要分情况讨论了。

对于不要求强一致性的应用场景，比如说在某些应用中更新用户头像时，我们可以接受这种临时性错误，只要能保证系统的最终一致性即可。但如果在要求强一致性的应用场景中，比如说在金融系统中进行和金钱有关的操作时，我们就需要做好一致性管理，我们可以对索引和数据进行统一加锁处理，或者直接将索引和数据合并。这么说比较抽象，我们以 MySQL 中的 B+ 树为例，来看看它是怎么管理一致性的。

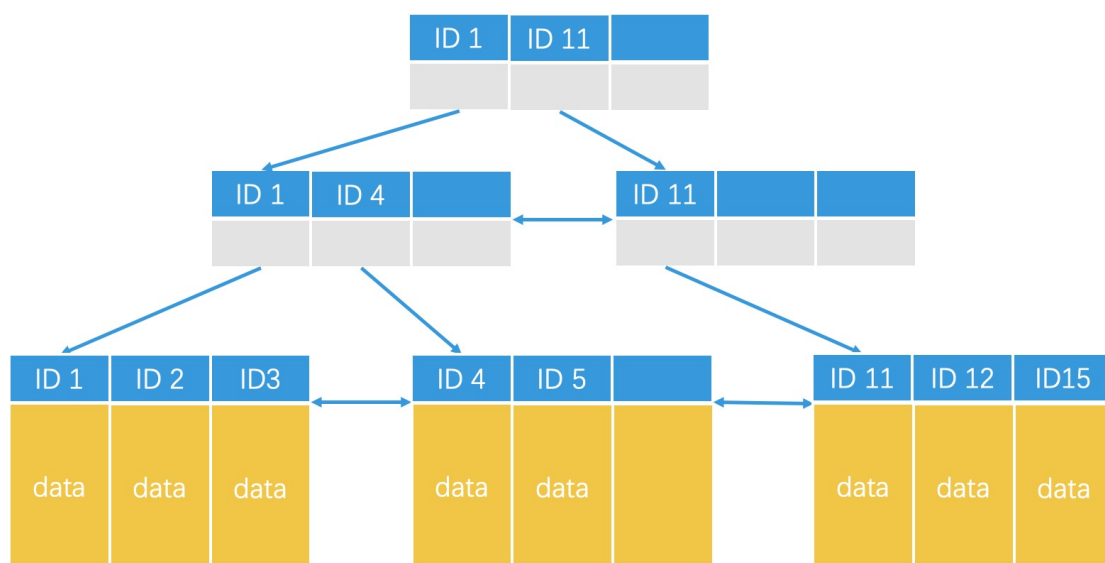
MySQL 中的 B+ 树实现其实有两种，一种是 MyISAM 引擎，另一种是 InnoDB 引擎。它们的核心区别就在于，数据和索引是否是分离的。

在 MyISAM 引擎中，B+ 树的叶子节点仅存储了数据的位置指针，这是一种索引和数据分离的设计方案，叫作非聚集索引。如果要保证 MyISAM 的数据一致性，那我们需要在表级别上进行加锁处理。



MyISAM的索引数据分离设计

在 InnoDB 中，B+ 树的叶子节点直接存储了具体数据，这是一种索引和数据一体的方案。叫作聚集索引。由于数据直接就存在索引的叶子节点中，因此 InnoDB 不需要给全表加锁来保证一致性，它只需要支持行级的锁就可以了。



InnoDB的索引数据一体设计

所以你看，索引和数据分离也不是万能的“银弹”，我们需要结合具体的使用场景，来进行合适的设计。

设计思想二：减少磁盘 IO

第 6 讲我们说过，在大规模系统中，数据往往无法全部存储在内存中。因此，系统必然会涉及磁盘的读写操作。在这种应用场景下，**尽可能减少磁盘 IO，往往是保证系统具有高性能的核心设计思路。**

减少磁盘 IO 的一种常见设计，**是将频繁读取的数据加载到内存中。**前面讲到的索引和数据分离的方案，就使得我们可以优先将索引加载在内存中，从而提高系统检索效率。

当频繁读取的数据也就是索引太大，而无法装入内存的时候，我们不会简单地使用跳表或者哈希表加载索引，而会采用更复杂的，具有压缩性质的前缀树这类数据结构和算法来压缩索引，让它能够放入内存中。

尽管，单纯从数据结构和检索效率来看，前缀树的检索性能是低于跳表或者哈希表的，但如果我们考虑到内存和磁盘在性能上的巨大差异的话，那这种压缩索引的优势就体现出来了。最典型的例子就是 lucene 中用 FST (Finite State Transducer, 中文：有限状态转换器) 来存索引字典。

除了使用索引和数据分离，在高维空间中使用 **乘积量化** 对数据进行压缩和检索，以及使用 **分布式技术** 将索引分片加载到不同的机器的内存中，也都可以减少磁盘的 IO 操作。

而如果不可避免要对磁盘进行读写，那我们要尽量避免随机读写。我们可以使用 **第 7 讲** 中学习到的，使用预写日志技术以及利用磁盘的局部性原理，来顺序写大批量数据，从而提高磁盘访问效率。这些都是一些优秀的开源软件使用的设计方案，比如，我们熟悉的基于 LSM 树的 Hbase 和 Kafka，就都采用了类似的设计。

你也许会问，如果改用 SSD 来存储数据（SSD 具有高性能的随机读写能力），那我们是不是就不用再关注减少磁盘 IO 的设计思想和技术了？当然不是，关于这个问题，我们可以从两方面来分析。

一方面，虽然 SSD 的确比磁盘快了许多，但 SSD 的性能和内存相比，依然会有 1 到 2 个数量级的巨大差距。甚至不同型号的 SSD 之间，性能也可能会有成倍的差距。再有，俗话

说“一分钱一分货”，内存和 SSD 的价格差异，其实也反映了它们随机读写性能的差距。因此内存是最快的，我们依然要尽可能把数据都存放在内存中。

另一方面，SSD 的批量顺序写依然比随机写有更高的效率。为什么呢？让我们先来了解一下 SSD 的工作原理。对于 SSD 而言，它以页（Page，一个 Page 为 4K-16K）为读写单位，以块（Block，256 个 Page 为一个 Block）为垃圾回收单位。由于 SSD 不支持原地更新的方式修改一个页，因此当我们写数据到页中时，SSD 需要将原有的页标记为失效，并将原来该页的数据和新写入的数据一起，写入到一个新的页中。那被标记为无效的页，会被垃圾回收机制处理，而垃圾回收又是一个很慢的操作过程。因此，随机写会造成大量的垃圾块，从而导致系统性能下降。所以，对于 SSD 而言，批量顺序写的性能依然会大幅高于随机写。

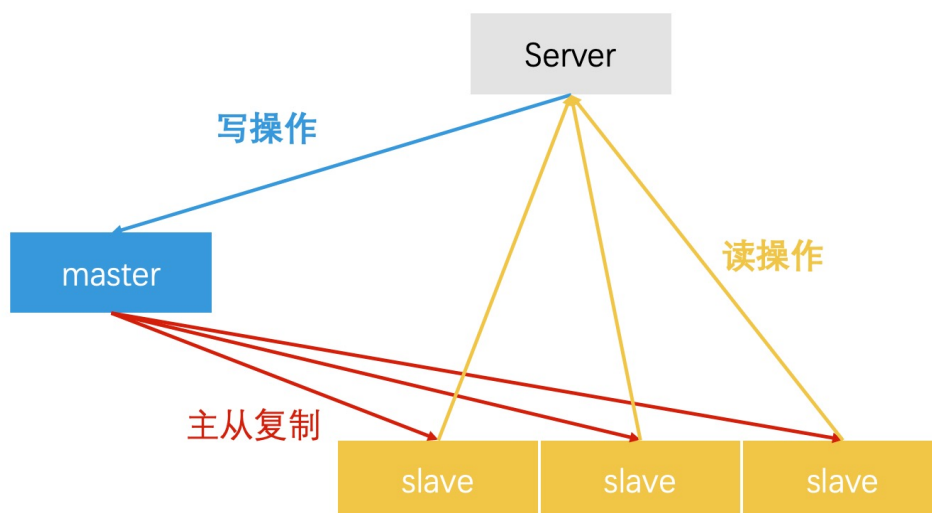
总结来说，无论存储介质技术如何变化，将索引数据尽可能地全部存储在最快的介质中，始终是保证高性能检索的一个重要设计思路。此外，对于每种介质的读写性能，我们都需要进行了解，这样才能做出合理的高性能设计。

设计思想三：读写分离

接下来，我们再说说读写分离，它也是很常见的一种设计方案。在高并发的场景下，如果系统需要对同一份数据同时进行读和写的操作，为了保证线程安全以及数据一致性，我们往往需要对数据操作加锁，但这会导致系统的性能降低。

而读写分离的设计方案，可以让所有的读操作只读一份数据，让所有的写操作作用在另一份数据上。这样就不存在读写竞争的场景，系统也就不需要加锁和切换了。在读操作频率高于写操作的应用场景中，使用读写分离的设计思想，可以大幅度提升系统的性能。很多我们熟悉的架构都采用这样的设计。

比如说，MySQL 的 Master - Slave 架构。在 MySQL 的 Master - Slave 的架构设计中，master 负责接收写请求。在数据写入 master 以后，master 会和多个 slave 进行同步，将数据更新到所有的 slave 中。所有的 slave 仅负责处理读请求。这样，MySQL 就可以完成读写分离了。



Master – slave架构实现读写分离示意图

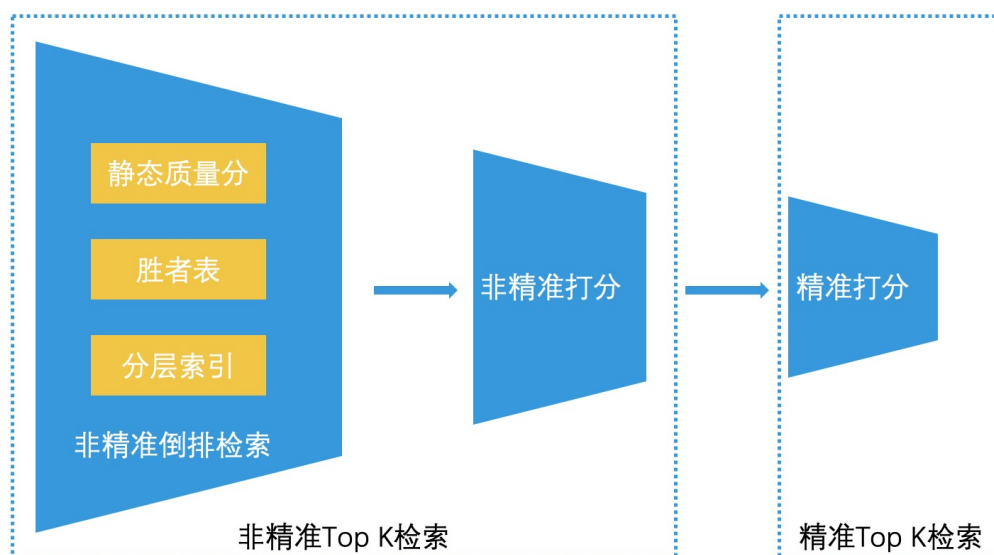
其实不仅仅是 MySQL，Redis 中也存在类似的 Master - Slave 的读写分离设计。包括在 LevelDB 中，MemTable - Immutable MemTable 的设计，其实也是读写分离的一个具体案例。

除了 MySQL 和 Redis 这类的数据存储系统，在倒排索引类的检索系统中，其实也有着类似的设计思路。比如说，在 [第 9 讲](#) 中我们讲过，对于索引更新这样的功能，我们往往会使用 Double Buffer 机制，以及全量索引 + 增量索引的机制来实现读写分离。通过这样的机制，我们才能保证搜索引擎、广告引擎、推荐引擎等系统能在高并发的应用场景下，依然具备实时的索引更新能力。

设计思想四：分层处理

在大规模检索系统中，不同数据的价值是不一样的，如果我们都使用同样的处理技术，其实会造成系统资源的浪费。因此，将数据分层处理也是非常基础且重要的一种设计。

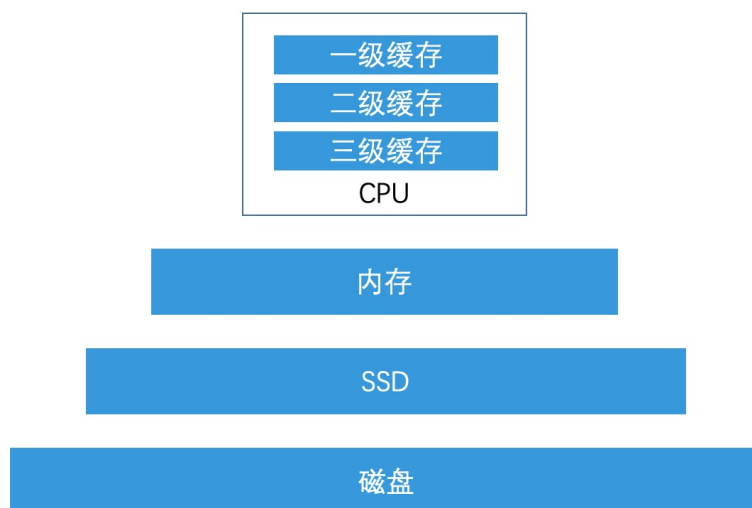
最典型的例子就是在 [第 12 讲](#) 中，我们提到的非精准 Top K 检索 + 精准 Top K 检索的设计思路了。简单回顾一下，如果我们对所有的检索结果集都进行耗时复杂的精准打分，那会带来大量的资源浪费。所以，我们会先进行初步筛选，快速选出可能比较好的结果，再进行精准筛选。这样的分层打分机制，其实非常像我们在招聘过程中，先进行简历筛选，再进行面试的处理流程。可见，这种设计思路有着非常广泛的使用场景。



分层检索+分层打分示意图

包括我们前面提到的将索引放在内存中而不是磁盘上，这其实也是一种分层处理的思路。我们对索引进行分层，把最有价值的索引放在内存中，保证检索效率。而价值较低的大规模索引，则可以存在磁盘中。

如果你再仔细回想一下就会发现，这其实和第 7 讲中，LSM 树将它认为最有价值的近期数据放在内存中，然后将其他数据放在磁盘上的思路是非常相似的。甚至是硬件设计也是这样，CPU 中的一级缓存和二级缓存，也是基于同样的分层设计理念，将最有价值的数



硬件分层架构设计

此外，还有一个典型的分层处理的设计实例就是，为了保证系统的稳定性，我们往往需要在系统负载过高时，启用自动降级机制。而好的自动降级机制，其实也是对流量进行分层处理，将系统认为有价值的流量保留，然后抛弃低价值的流量。

总结来说，如果我们在应用场景中，对少数的数据进行处理，能带来大量的产出，那分层处理就是一种可行的设计思路。这其实就是**二八原则：20% 的资源带来了 80% 的产出**。

重点回顾

今天，我们讨论了索引和数据分离、减少磁盘 IO、读写分离、以及分层处理这四种设计思想。你会发现，这些设计思想说起来并不复杂，但几乎无处不在。为了方便你使用，我总结了一下，它们在使用上的一些通用经验。

首先，索引和数据分离能让我们遵循“奥卡姆剃刀原则”，聚焦于索引的优化技术上。但我们还要注意，索引和数据分离可能会带来数据的不一致性，因此需要合理使用。

其次，减少磁盘 IO 有两种主要思路：一种是尽可能将索引加载到最快的内存中；另一种是对磁盘进行批量顺序读写。并且，即便存储介质技术在不断变化，但我们只要保持关注这两个优化方向，就可以设计出高性能的系统。

接着，在读写分离的设计中，使用 Master - Slave 的架构设计也是一种常见方案，无论是 MySQL 还是 Redis 都采用了这种设计方案。而基于倒排索引的检索系统，也有着类似的 Double Buffer 和全量索引结合增量索引的机制。

最后，分层处理其实就是遵循二八原则，将核心资源用来处理核心的数据，从而提升整体系统的性能。

当你要设计或实现高并发系统时，你可以试试使用这些思想来指导你设计和审视系统，相信这会让你的系统变得更加高效。

课堂讨论

对于今天我们讨论的这些设计思想，你有什么案例可以分享出来吗？

欢迎在留言区畅所欲言，说出你的案例和思考。如果有收获，也欢迎把这一讲分享给你的朋友。

学习计划

五一计划

晒学习姿势
「免费」领课程



【点击】图片，立即参加 >>>

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (1)

写留言



那一刻

2020-05-04

对于老师总结的这几部分，我的想法如下，不恰当的地方，烦请老师指正。

1. 索引和数据分离

我想到了存储计算分离的设计，比如消息队列Pulsar就是采用这样设计，存储消息的职责从Broker分离出来，交给专门存储集群，这样Broker就变成无状态的节点，灵活的集群调度。Bigquery和aws Aurora也是如此的设计。...

展开 ∨

