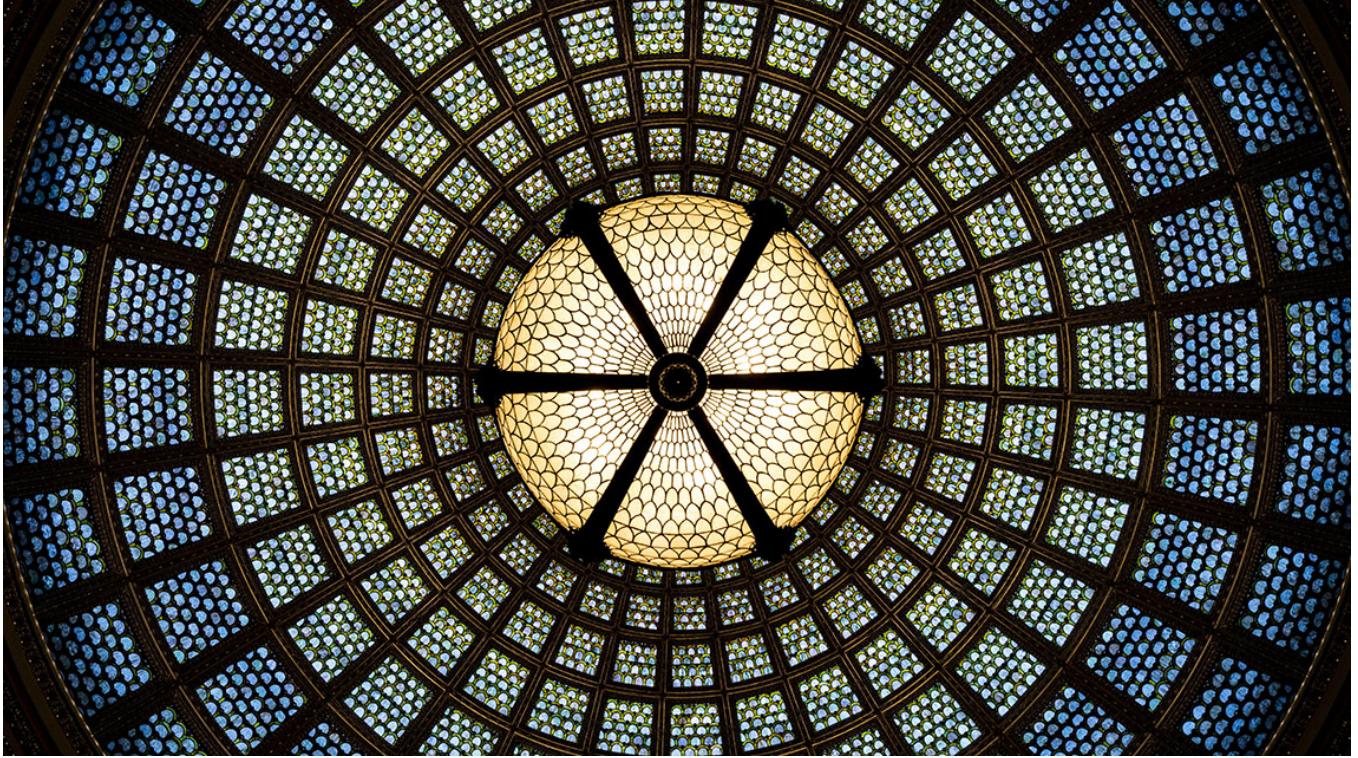


性能设计篇之"数据库扩展"

2018-05-17 陈皓



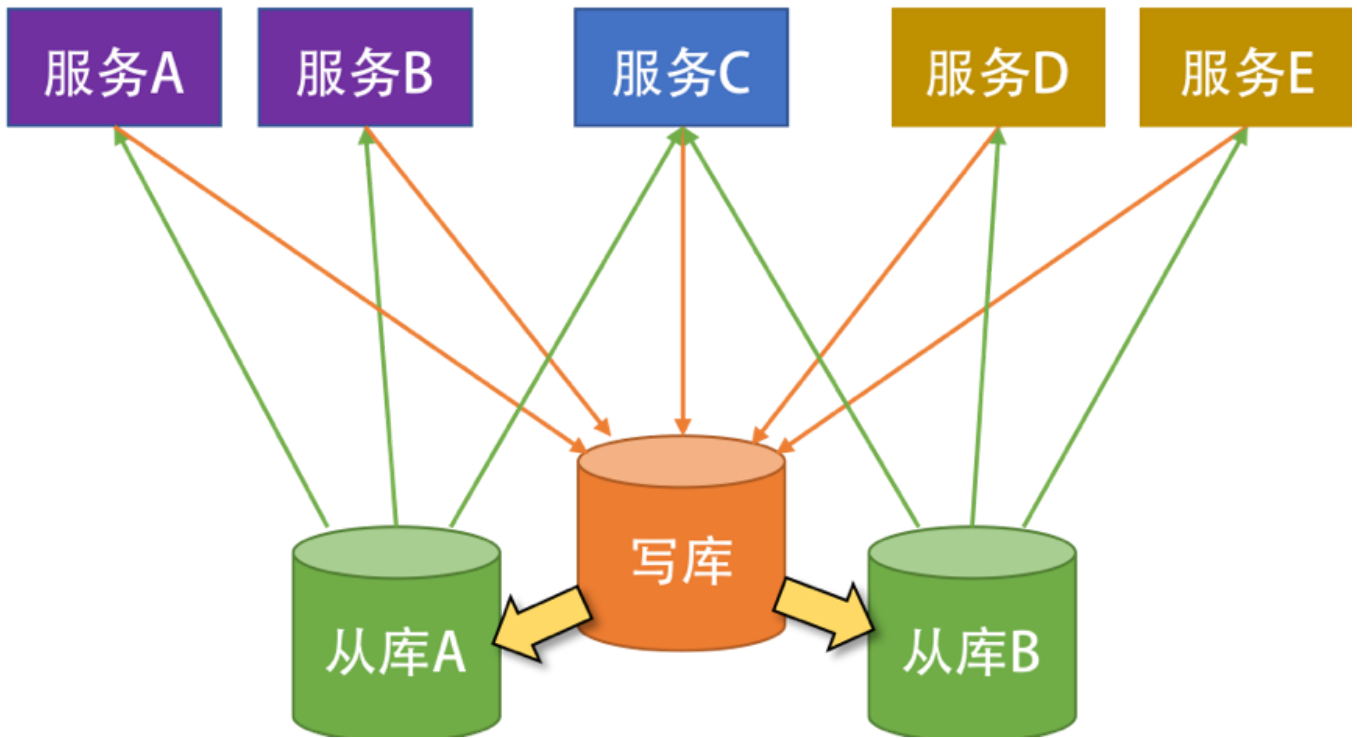
性能设计篇之"数据库扩展"

朗读人：柴巍 11'55" | 5.46M

读写分离 CQRS

读写分离是数据库扩展最简单实用的玩法了，这种方法针对读多写少的业务场景还是很管用的，而且还可以有效地把业务做相应的隔离。

如下图所示，数据库只有一个写库，有两个读库，所有的服务都写一个数据库。对于读操作来说，服务 A 和服务 B 走从库 A，服务 D 和服务 E 走从库 B，服务 C 在从库 A 和从库 B 间做轮询。



这样的方法好处是：

- 比较容易实现。数据库的 master-slave 的配置和服务框架里的读写分离都比较成熟，应用起来也很快。
- 可以很好地把各个业务隔离开来。不会因为一个业务把数据库拖死而导致所有的业务都死掉。
- 可以很好地分担数据库的读负载，毕竟读操作是最耗数据库 CPU 的操作。

这样的方法不好的地方是：

- 写库有单点故障问题。如果是写库出了性能问题，那么所有的业务一样不可用。对于交易型的业务，要得到高的写操作速度，这样的方式不行。
- 数据库同步不实时，需要强一致性的读写操作还是需要落在写库上。

综上所述，一般来说，这样的玩法主要是为了减少读操作的压力。

当然，这样的读写分离看上去有点矬，那么，我们还是为之找一个更靠谱的设计——CQRS。关于 CQRS，我在这里只做一个简单的介绍，更多的细节你可以上网自行 Google。

CQRS 全称 Command and Query Responsibility Segregation，也就是命令与查询职责分离。其原理是，用户对于一个应用的操作可以分成两种，一种是 Command 也就是我们的写操作（增，删，改），另一种是 Query 操作（查），也就是读操作。Query 操作基本上是在做数据整合显现，而 Command 操作这边会有更重的业务逻辑。分离开这两种操作可以在语义上做好区分。

- 命令 Command 不会返回结果数据，只会返回执行状态，但会改变数据。

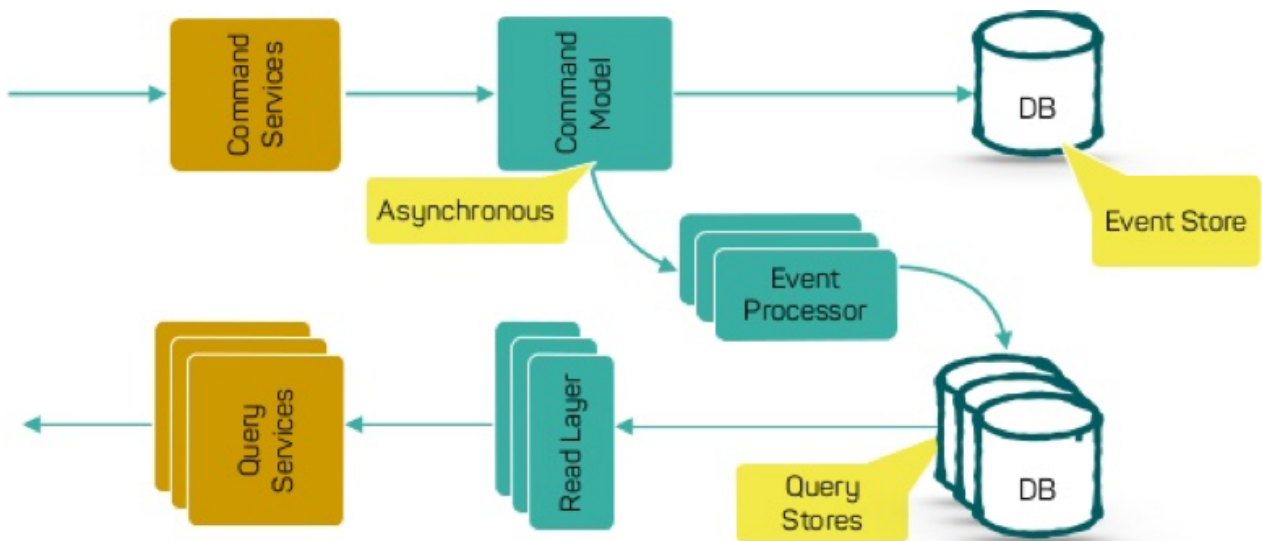
- 查询 Query 会返回结果数据，但是不会改变数据，对系统没有副作用。

这样一来，可以带来一些好处。

- 分工明确，可以负责不同的部分。
- 将业务上的命令和查询的职责分离，能够提高系统的性能、可扩展性和安全性。并且在系统的演化中能够保持高度的灵活性，能够防止出现 CRUD 模式中，对查询或者修改中的某一方进行改动，导致另一方出现问题的情况。
- 逻辑清晰，能够看到系统中的哪些行为或者操作导致了系统的状态变化。
- 可以从数据驱动（Data-Driven）转到任务驱动（Task-Driven）以及事件驱动。

如果把 Command 操作变成 Event Sourcing，那么只需要记录不可修改的事件，并通过回溯事件得到数据的状态。于是，我们可以把写操作给完全简化掉，也变成无状态的，这样可以大幅度降低整个系统的副作用，并可以得到更大的并发和性能。

文本中有 Event Sourcing 和 CQRS 的架构示意图。



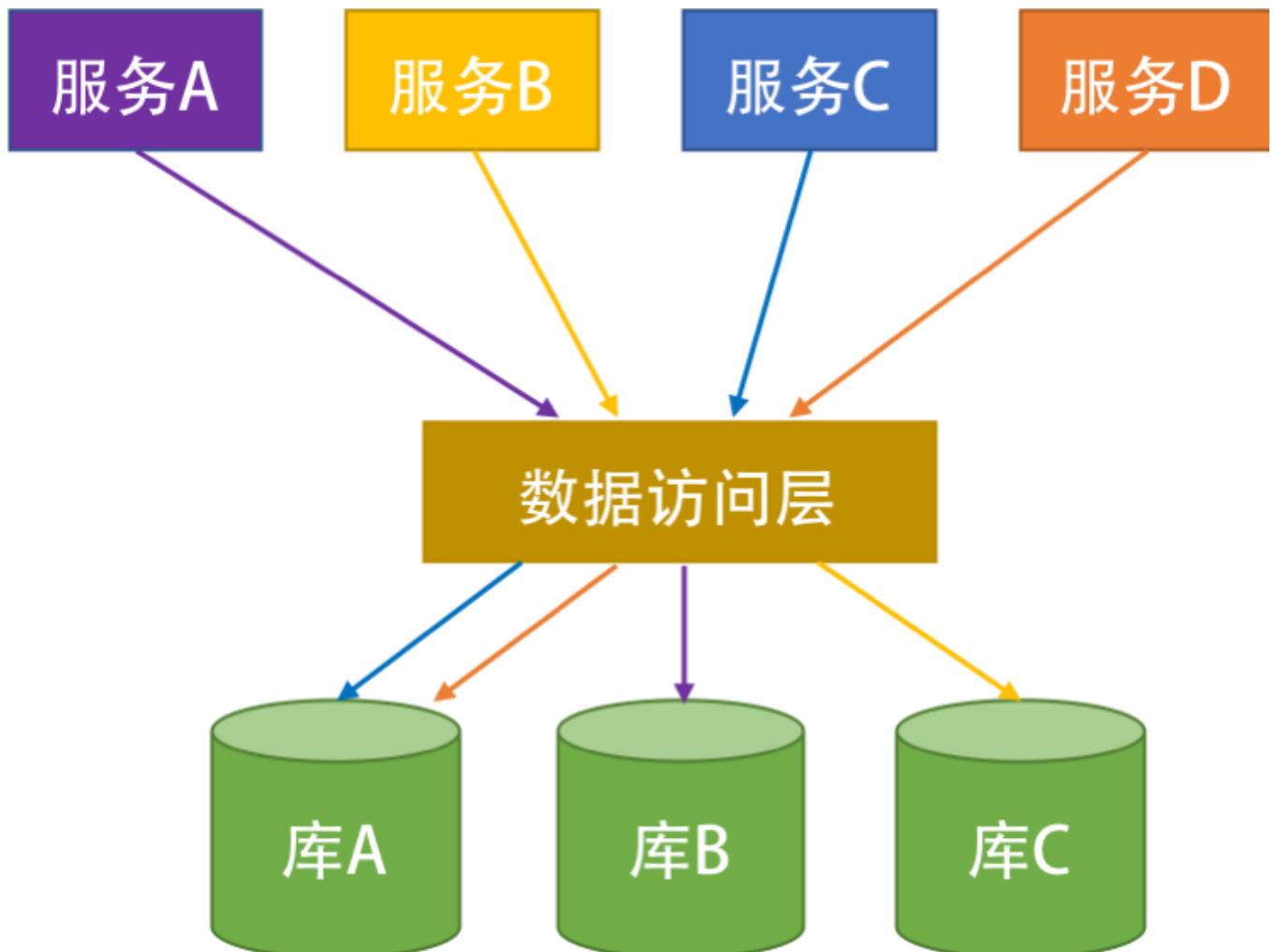
图片来源 - [CQRS and Event Sourcing Application with Cassandra](#)

分库分表 Sharding

一般来说，影响数据库最大的性能问题有两个，一个是对数据库的操作，一个是数据库中数据的大小。

对于前者，我们需要从业务上来优化。一方面，简化业务，不要在数据库上做太多的关联查询，而对于一些更为复杂的用于做报表或是搜索的数据库操作，应该把其移到更适合的地方。比如，用 ElasticSearch 来做查询，用 Hadoop 或别的数据分析软件来做报表分析。

对于后者，如果数据库里的数据越来越多，那么也会影响我们的数据操作。而且，对于我们的分布式系统来说，后端服务都可以做成分布式的，而数据库最好也是可以拆成分布式的。读写分离也因为数据库里的数据太多而变慢，于是，分库分表就成了我们必须用的手段。



上面的图片是一个分库的示例。其中有两个事，这里需要提一下，一个是关于分库的策略，一个是关于数据访问层的中间件。

关于分库的策略。我们把数据库按某种规则分成了三个库。比如，或是按地理位置，或是按日期，或是按某个范围分，或是按一种哈希散列算法。总之，我们把数据分到了三个库中。

关于数据访问层。为了不让我们前面的服务感知到数据库的变化，我们需要引入一个叫“数据访问层”的中间件，用来做数据路由。但是，老实说，这个数据访问层的中间件很不好写，其中要有解析 SQL 语句的能力，还要根据解析好的 SQL 语句来做路由。但即便是这样，也有很多麻烦事。

比如，我要做一个分页功能，需要读一组顺序的数据，或是需要做 Max/Min/Count 这样的操作。于是，你要到三个库中分别求值，然后在数据访问层这里再合计处理返回。但即使是这样，你也会遇到各种令人烦恼的事，比如一个跨库的事务，你需要走 XA 这样的两阶段提交的操作，这样会把数据库的性能降到最低的。

为了避免数据访问层的麻烦，分片策略一般如下。

- 按多租户的方式。用租户 ID 来分，这样可以把租户隔离开来。比如：一个电商平台的商家中心可以按商家的 ID 来分。
- 按数据的种类来分。比如，一个电商平台的商品库可以按类目来分，或是商家按地域来分。
- 通过范围来分。这样分片，可以保证在同一片片中的数据是连续的，于是我们数据库操作，比如分页查询会更高效一些。一般来说，大多数情况是用时间来分片的，比如，一个电商平台的订单中心是按月份来分表的，这样可以快速检索和统计一段连续的数据。
- 通过哈希散列算法来分（比如：主键 $id \% 3$ 之类的算法。）此策略的目的是降低形成热点的可能性（接收不成比例的负载的分片）。但是，这会带来两个问题，一个就是前面所说的跨库跨表的查询和事务问题，另一个就是如果要扩容需要重新哈希部分或全部数据。

上面是最常见的分片模式，但是你还应考虑应用程序的业务要求及其数据使用模式。这里请注意几个非常关键的事宜。

1. 数据库分片必须考虑业务，从业务的角度入手，而不是从技术的角度入手，如果你不清楚业务，那么无法做出好的分片策略。
2. 请只考虑业务分片。请不要走哈希散列的分片方式，除非有个人拿着把刀把你逼到墙角，你马上就有生命危险，你才能走哈希散列的分片方式。

数据库扩展的设计重点

先说明一下，这里没有讲真正数据库引擎的水平扩展的方法，我们只是在业务层上谈了一下数据扩展的两种方法。关于数据库引擎的水平扩展，你可能看一下我之前发过的《分布式数据调度的相关论文》一文中的 AWS Aurora 和 Google Spanner 的相关论文中提到的那些方法。

接下来，我们说一下从业务层上把单体的数据库给拆解掉的相关重点。

首先，你需要把数据库和应用服务一同拆开。也就是说，一个服务一个库，这就是微服务的玩法，也是 Amazon 的服务化的玩法——服务之间只能通过服务接口通讯，不能通过访问对方的数据库。在 Amazon 内，每个服务都会有一个自己的数据库，比如地址库、银行卡库等。这样一来，你的数据库就会被“天生地”给拆成服务化的，而不是一个单体的库。

我们要知道，在一个单体的库上做读写分离或是做分片都是一件治标不治本的事，真正治本的方法就是要和服务一起拆解。

当数据库也服务化后，我们才会在这个小的服务数据库上进行读写分离或分片的方式来获得更多的性能和吞吐量。这是整个设计模式的原则——先做服务化拆分，再做分片。

对于分片来说，有两种分片模式，一种是水平分片，一种是垂直分片。水平分片就是我们之前说的那种分片。而垂直分片是把一张表中的一些字段放到一张表中，另一些字段放到另一张表中。垂直分片主要是把一些经常修改的数据和不经常修改的数据给分离开来，这样在修改某个字段的数据时，不会导致其它字段的数据被锁而影响性能。比如，对于电商系统来说，商品的描述信息不常改，但是商品的库存和价格经常改，所以，可以把描述信息和库存价格分成两张表，这样可以让商品的描述信息的查询更快。

我们所说的 sharding 更多的是说水平分片。水平分片需要有以下一些注意事项。

- 随着数据库中数据的变化，我们有可能需要定期重新平衡分片，以保证均匀分布并降低形成热点的可能性。但是，重新平衡是一项昂贵的操作。若要减少重新平衡的频率，我们需要通过确保每个分片包含足够的可用空间来处理未来一段时间的变化。另外，我们还需要开发用于快速重新平衡分片的工具和脚本。
- 分片是静态的，而数据的访问则是不可预期的，可能会需要经常性地调整我们的分片，这样一来成本太高。所以，我们最好使用一个索引表的方式来进行分片。也就是说，把我们数据的索引动态地记录在一个索引表中。这样一来，我们就可以非常灵活地调度我们的数据了。当数据调度到另一台结点上时，我们只需要去索引表里改一下这个数据的位置就好了。
- 如果程序必须要从多个分片检索数据的查询，则可以使用并行任务从各个分片上提取此数据，然后聚合到单个结果中。但是，此方法不可避免地会在一定程度上增加解决方案数据访问逻辑的复杂性。
- 数据分片后，我们很难在分片之间保持引用完整性和一致性，也就是所谓的跨分片的事务，因此应尽量减少会影响多个分片中的数据的操作。如果应用程序必须跨分片修改数据，那么我们需要评估一致性以及评估是否采用两阶段提交的方式。
- 配置和管理大量分片可能是一个挑战。在做相应的变更时，一定要先从生产线上拉出数据，然后根据数据计划好新的分片方式，并做好相当的测试工作。否则，这个事出了问题会是一个灾难性的问题。

小结

好了，我们来总结一下今天分享的主要内容。首先，我介绍了单主库多从库的读写分离，并进一步用 CQRS 把语义区分成命令和查询。命令的执行可以变成事件溯源方式，从而得到更大的并发和性能。随后我讲了分库分表的策略及其数据访问层所做的抽象。最后，我指出了数据库扩展的设计重点。下篇文章中，我们讲述秒杀。希望对你有帮助。

也欢迎你分享一下你的数据库做过哪些形式的扩展？设计中有哪些方面的考量？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
 - [认识故障和弹力设计](#)
 - [隔离设计 Bulkheads](#)
 - [异步通讯设计 Asynchronous](#)
 - [幂等性设计 Idempotency](#)
 - [服务的状态 State](#)
 - [补偿事务 Compensating Transaction](#)
 - [重试设计 Retry](#)
 - [熔断设计 Circuit Breaker](#)
 - [限流设计 Throttle](#)
 - [降级设计 degradation](#)
 - [弹力设计总结](#)
- 管理设计篇
 - [分布式锁 Distributed Lock](#)
 - [配置中心 Configuration Management](#)
 - [边车模式 Sidecar](#)
 - [服务网格 Service Mesh](#)
 - [网关模式 Gateway](#)
 - [部署升级策略](#)
- 性能设计篇
 - [缓存 Cache](#)
 - [异步处理 Asynchronous](#)
 - [数据库扩展](#)
 - [秒杀 Flash Sales](#)
 - [边缘计算 Edge Computing](#)



版权归极客邦科技所有，未经许可不得转载

精选留言



唐稳

👍 2

CQRS应该用在没有事务强一致性要求的场合，才能充分发挥其作用。不过微服务架构似乎更倾向于设计出最终一致性的程序。

2018-05-17

作者回复

嗯。另外，仔细想想，强一致性这种场景真的不多。

2018-05-22



W_T

👍 2

按照哈希散列分片，实现方案最简单，只需要在操作数据库的时候特殊处理就可以了。按照业务分片，为了减少跨分片操作，在请求的前端就需要明确业务字段的值，所以并不是所有场景都适用，这些方案各有利弊。不过有一点我还是赞同的，不到万不得已，不要用哈希散列分片，不然等到以后要重新分片的时候代价巨大。

2018-05-17

作者回复

业务分片，其实直接就数据分库，服务拆分，走向微服务得了。

2018-05-22



吞枣

👍 1

感觉分库分表是分布式数据库到来之前的临时方案，另外感觉老外们好像并不怎么会采用分库分表，是这样吗？

2018-05-22



_CountingStars

1

索引表也越来越多大 需要分片怎么办呢

2018-05-17

作者回复

索引表没有业务属性，就是kv，没有join，没有group，所以非常容易用哈希分片

2018-05-22



Geek_22d08b

0

请问如果采用阿里云华为云的话，那么多技术要实现是不是只要购买阿里云他们相应的产品，然后配置下就可以了，就没程序员什么事了？

2018-05-19



A:春哥大魔王

0

cqrs设计中，如果写操作在逻辑上有一定的查询结果判断，该如何对query和command进行更好的设计呢？

2018-05-18

作者回复

统一算写操作

2018-05-22



chaoqiang

0

请只考虑业务分片。请不要走哈希散列的分片方式

对这句话不太理解，走哈希分片虽然是有跨表查询隐患，后续数据量再次暴涨也需要重新哈希，比较恶心，但也可以解决热点问题，而且互联网公司的用户数据大部分场景下都是有热点的吧，为什么皓叔这么反对呢？实际场景中会遇到什么更痛的点嘛？能否更详细地讲讲呢

2018-05-17