后语：除了水平切分，数据库架构设计还经常遇到哪些问题

不知不觉，数据库水平切分的系列文章已经走过半年，很感谢朋友们一路的陪伴，也感谢gitchat这个平台，能够把自己曾经实践过的水平切分架构方案，梳理总结和沉淀，系统的记录成文字，和大家一起分享。

本文是这个专题的最后一讲，在设定这个专题之初，就已经想好，这一讲的内容了：

- 一部分，**总结**：特别担心大家没有完整的订阅这一系列文章，无法系统的看到全貌，特地留了一篇总结性质的文章，把1-5篇的知识点系统性的回顾。
- 一部分，**挖坑**：互联网数据库架构，水平切分只是其中的一部分，除此之外，还会遇到很多问题，而这些问题并不是一篇文章就能讲清楚的，如果未来有缘，再和大家深入交流探讨。

一、《前言：数据库水平切分通用实践》

从《前言：数据库水平切分通用实践》这篇文章，能够了解到：

- 数据库**分组架构**的概念，特点，解决的问题域，最佳实践。
- 数据库**分片架构**的概念，特点，解决的问题域，最佳实践。
- 数据库**垂直拆分**的概念，特点，解决的问题域，最佳实践。

二、《从用户中心开始，聊“单KEY”类业务数据库水平切分架构实践》

从《从用户中心开始，聊“单KEY”类业务数据库水平切分架构实践》这篇文章，能够了解到：

水平切分方式

- 范围法
- 哈希法

水平切分后碰到的问题

- 通过uid属性查询能直接定位到库，通过非uid属性查询不能定位到库。

非uid属性查询的典型业务

- 用户侧，前台访问，单条记录的查询，访问量较大，服务需要高可用，并且对一致性的要求较高。
- 运营侧，后台访问，根据产品、运营需求，访问模式各异，基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性的要求不高，对一致性的要求也没这么严格。

用户侧与运营侧架构设计思路

- 针对用户侧，应该采用“建立非uid属性到uid的映射关系”的架构方案。
- 针对运营侧，应该采用“前台与后台分离”的架构方案。

用户前台侧，“建立非uid属性到uid的映射关系”最佳实践

- 索引表法：数据库中记录login_name->uid的映射关系。
- 缓存映射法：缓存中记录login_name->uid的映射关系。
- login_name生成uid。
- login_name基因融入uid。

运营后台侧，“前台与后台分离”最佳实践

- 前台、后台系统web/service/db分离解耦，避免后台低效查询引发前台查询抖动。
- 可以采用数据冗余的设计方式。
- 可以采用“外置索引”（例如ES搜索系统）或者“大数据处理”（例如HIVE）来满足后台变态的查询需求。

三、《从帖子中心开始，聊“1对多”类业务数据库水平切分架构实践》

从《从帖子中心开始，聊“1对多”类业务数据库水平切分架构实践》这篇文章，能够了解到：

“1对多”类业务，在架构上，采用元数据与索引数据分离的架构设计方法

- 帖子服务，元数据满足uid和tid的查询需求。
- 搜索服务，索引数据满足复杂搜索寻求。

对于元数据的存储，在数据量较大的情况下，有三种常见的切分方法：

- tid切分法，按照tid分库，同一个用户发布的帖子落在不同的库上，通过 - uid来查询要遍历所有库。
- uid切分法，按照uid分库，同一个用户发布的帖子落在同一个库上，需要通过索引表或者缓存来记录tid与uid的映射关系，通过tid来查询时，先查到uid，再通过uid定位库。
- 基因法，按照uid分库，在生成tid里加入uid上的分库基因，保证通过uid和tid都能直接定位到库。

四、《从好友关系开始，聊“多对多”类业务数据库水平切分架构实践》

从《从好友关系开始，聊“多对多”类业务数据库水平切分架构实践》这篇文章，能够了解到：

好友业务是一个典型的多对多关系，又分为强好友与弱好友

数据冗余是一个常见的多对多业务数据水平切分实践

冗余数据的常见三种方案

- 服务同步冗余
- 服务异步冗余
- 线下异步冗余

数据冗余会带来一致性问题，高吞吐互联网业务，要想完全保证事务一致性很难，常见的实践是最终一致性

最终一致性的常见实践是，尽快找到不一致，并修复数据，常见三种方案

- 线下全量扫描法
- 线下增量扫描法
- 线上实时检测法

五、《从订单中心开始，聊“多KEY”类业务数据库水平切分架构实践》

从《从订单中心开始，聊“多KEY”类业务数据库水平切分架构实践》这篇文章，能够了解到：

任何复杂难题的解决，都是一个化繁为简，逐步击破的过程。

对于像订单中心一样复杂的“多key”类业务，在数据量较大，需要对数据库进行水平切分时，对于后台需求，采用“前台与后台分离”的架构设计方法：

- 前台、后台系统web/service/db分离解耦，避免后台低效查询引发前台查询抖动。
- 采用前台与后台数据冗余的设计方式，分别满足两侧的需求。
- 采用“外置索引”（例如ES搜索系统）或者“大数据处理”（例如HIVE）来满足后台变态的查询需求。

对于前台需求，化繁为简的设计思路，将“多key”类业务，分解为“1对多”类业务和“多对多”类业务分别解决：

- 使用“基因法”，解决“1对多”分库需求：使用 buyer_uid 分库，在oid中加入分库基因，同时满足oid和 buyer_uid 上的查询需求。
- 使用“数据冗余法”，解决“多对多”分库需求：使用 buyer_uid 和 seller_uid 来分别分库，冗余数据，满足 buyer_uid 和 seller_uid 上的查询需求。
- 如果 oid/buyer_uid/seller_uid 同时存在，可以使用上述两种方案的综合方案，来解决“多key”业务的数据库水平切分难题。

六、数据库架构设计其他问题

上述1-5篇文章，仅仅只是展开描述了“水平切分”这一个话题，在数据库架构设计过程中，除了水平切分，至少还会遇到这样一些问题：

- **可用性**：不管是主库实例，还是从库实例，如果数据库实例挂了，如何不影响数据的读和写。
- **读性能**：互联网业务大多是读多写少的业务，如果提升数据库的读性能是架构设计中必须考虑的问题。
- **一致性**：数据一旦冗余，就可能出现一致性问题，如何解决主库与从库之间的不一致，如何解决数据库与缓存之间的不一致，也是需要重点设计的。
- **扩展性**：如何在不停服务的情况下扩充数据表的属性，实施数据迁移，实施存储引擎的切换，架构设计上都是十分有讲究的。
- **分布式SQL语句**：单库情况下，所有SQL语句的执行都没问题问题，一旦实施了水平切分，如何实现SQL的集函数，分页，非patition key上的查询都成了大问题。

上面这些问题，都不是简单三言两语能够说清楚的，大家对哪个话题感兴趣，欢迎留言，未来和大家深聊。

一篇文章的1块钱，和一系列文章的几十块钱，获取的知识内容是一样的，差别只在于，迟到了半年，希望，这半年不算晚。

希望大家继续支持GitChat，希望大家继续支持“架构师之路”，任何数据库架构问题欢迎评论提问，下次咱们再见，希望对得起这1块钱。