

数据库拆分与一致性哈希算法

Scaling Database & Consistent Hashing

课程版本 v6.0 本节主讲人 东邪



扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: [ninechapter](#)

微博: <http://www.weibo.com/ninechapter>

知乎: <http://zhuanlan.zhihu.com/jiuzhang>

官网: <http://www.jiuzhang.com>

Interviewer: How to scale?

当访问量越来越大以后, 如何让你的系统 Scale ?
How to scale system \approx How to scale database

除了QPS, 还有什么需要考虑的？

假如用户的 QPS 有 1k

Database 和 Web Server 假设也均能够承受 1k 的 QPS

还有什么情况需要考虑？

单点失效

Single Point Failure

万一这一台数据库挂了
短暂的挂: 网站就不可用了
彻底的挂: 数据就全丢了

所以你需要做两件事情

1. 数据拆分 Sharding (又名 Partition)
2. 数据复制 Replica

数据拆分 Sharding

又名 Partition

方法:按照一定的规则,将数据拆分开存储在不同的实体机器上。

意义:

1. 挂了一台不会全挂
2. 分摊读写流量

数据复制 Replica

方法：一式三份(重要的事情写三遍)

意义：

1. 一台机器挂了可以用其他两台的数据恢复
2. 分摊读请求

Sharding in SQL vs NoSQL

数据拆分与数据库的类型无关

无论是 SQL 还是 NoSQL 都可以进行数据拆分

大部分的 NoSQL 已经帮你写好了拆分算法

数据拆分 Sharding

纵向拆分 Vertical Sharding

横向拆分 Horizontal Sharding

纵向切分 Vertical Sharding

User Table 放一台数据库

Friendship Table 放一台数据库

Message Table 放一台数据库

...

稍微复杂一点的 Vertical Sharding

- 比如你的 User Table 里有如下信息
 - email
 - username
 - password
 - nickname // 昵称
 - avatar // 头像
- 我们知道 email / username / password 不会经常变动
- 而 nickname, avatar 相对来说变动频率更高
- 可以把他们拆分为两个表 User Table 和 UserProfile Table
 - UserProfile 中用一个 user_id 的 foreign key 指向 User
 - 然后再分别放在两台机器上
 - 这样如果 UserProfile Table 挂了, 就不影响 User 正常的登陆

table存储的内容即使没有很大，但是table频繁需要被用到查询（QPS高），是不是也会是vertical sharding的缺点？

水平切片，一次访问只需访问一次数据库；垂直切片，一次需要访问多个数据库（并不能将访问压力平摊到多个数据库上，这是重点）。

提问

Vertial Sharding 的缺点是什么？
不能解决什么问题？



@No One

User分成两张表之后QPS不就double了吗？这样还有意义吗？

qps未必double，得看具体访问什么数据，单表能不能满足。

垂直拆分：

优点：系统之间整合或扩展容易。拆分后业务清晰，拆分规则明确。数据维护简单。

缺点：事务处理复杂。受每种业务不同的限制存在单库性能瓶颈，不易数据扩展跟性能提高。部分业务表无法join，只能通过接口方式解决，提高了系统复杂度。

如果数据非常非常大，能不能先 Vertical Sharding 然后再 Horizontal Sharding？比如先把User Profile 分成和一个纵向的服务，然后再按照 userid 来横向 sharding？

垂直分表不能提升性能，还会降低性能（原本一个table读完数据，现在需要在多个table中读取，效率降低了）。

建议只水平分表。

横向切分

Horizontal Sharding

核心部分！

Scale 的核心考点！

猜想1：新数据放新机器，旧数据放旧机器

比如一台数据库如果能放下 1T 的数据
那么超过1T之后就放在第二个数据库里
以此类推

问：这种方法的问题是什么？

[单选题] 【系统设计2020】新数据放新机器，老数据放老机器的问题是什么

- A. 数据访问不均匀 43.38% 选择
- B. 数据存储不均匀 13.68% 选择
- C. 取数据时无法确定数据存在哪儿 42.94% 选择

答错了，您选择的答案是 C

正确答案： A

解析：
根据数据的新旧程度来拆分的话，新数据的访问次数比旧数据的访问次数是要明显多的，会导致数据访问不均匀的问题。
这种方法并不会导致存储不均匀，最多只有最新的一台机器的数据相对少一些，其他的机器都还是均匀的。也不会导致不知道数据去哪台机器取，比如根据 id 来拆分，0~99在1号机器，1~199在2号机器的话，根据 id 可以算出对应的机器是哪个。

关闭

猜想2: 对机器数目取模

假如我们来拆分 Friendship Table

我们有 3 台数据库的机器

于是想到按照 $\text{from_user_id} \% 3$ 进行拆分

这样做的问题是啥？

假如 3 台机器不够用了

我现在新买了1台机器

原来的%3, 就变成了%4

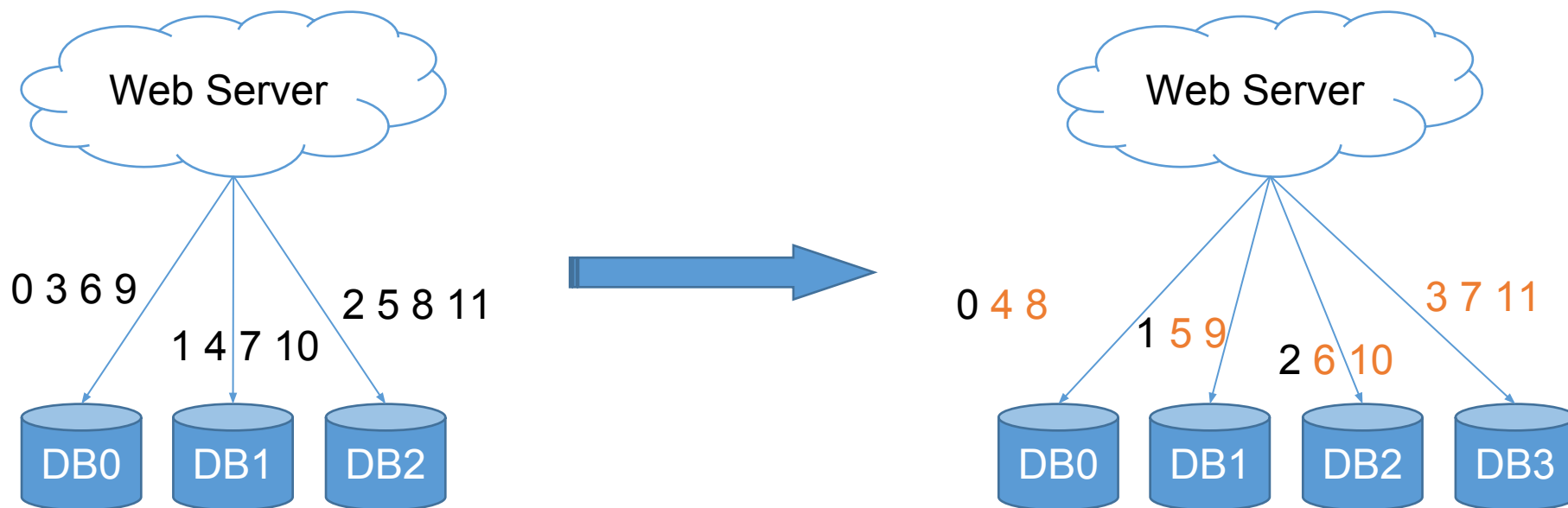
几乎所有的数据都要进行位置大迁移

过多的数据迁移会造成的问题

1. 慢, 容易造成数据的不一致性
2. 迁移期间, 服务器压力增大, 容易挂

直接 %n (机器总数)的方法

- % n 的方法是一种最简单的 Hash 算法
 - 但是这种方法在 n 变成 n+1 的时候, 每个 key % n 和 % (n+1) 结果基本上都不一样
 - 所以这个 Hash 算法可以称之为: 不一致 hash



怎么办？

一致性 Hash 算法

Consistent Hashing

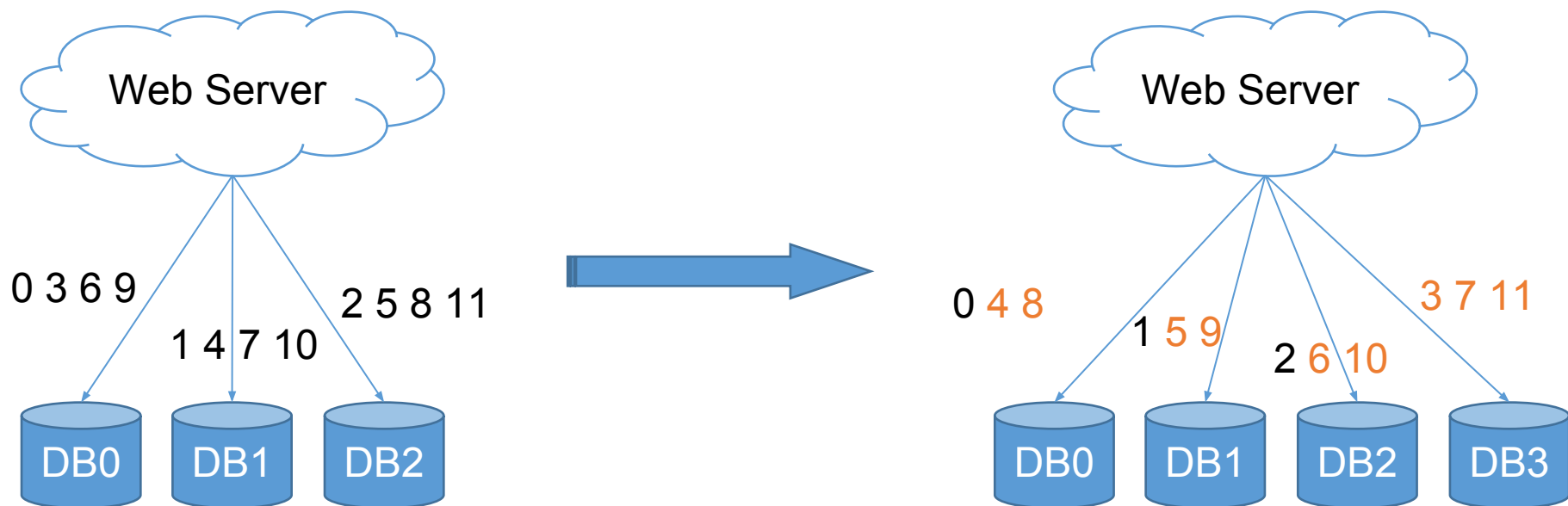
一致性哈希算法 Consistent Hashing

Horizontal Sharding 的秘密武器

无论是 SQL 还是 NoSQL 都可以用这个方法进行 Sharding

注: 大部分 NoSQL 都帮你实现好了这个算法, 帮你自动 Sharding
很多 SQL 数据库也逐渐加入 Auto-scaling 的机制了, 也开始帮你做
自动的 Sharding

- 我们先来说说为什么要做“一致性”Hash
 - $\% n$ 的方法是一种最简单的 Hash 算法
 - 但是这种方法在 n 变成 $n+1$ 的时候, 每个 $\text{key} \% n$ 和 $\% (n+1)$ 结果基本上都不一样
 - 所以这个 Hash 算法可以称之为: 不一致 hash

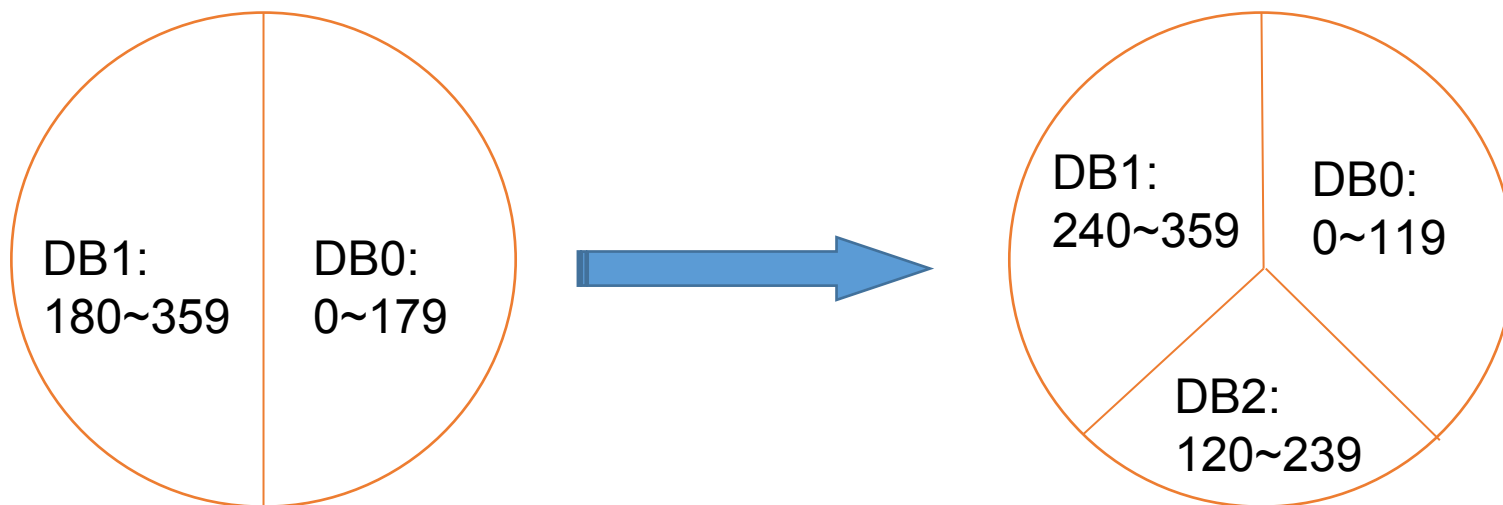


- 一个简单的一致性Hash算法
 - 将 key 模一个很大的数, 比如 360
 - 将 360 分配给 n 台机器, 每个机器负责一段区间
 - 区间分配信息记录为一张表存在 Web Server 上
 - 新加一台机器的时候, 在表中选择一个位置插入, 匀走相邻两台机器的一部分数据
- 比如 n 从 2 变化到 3, 只有 1/3 的数据移动

请问在进行数据迁移的时候, 服务器是不是就要停止服务了?

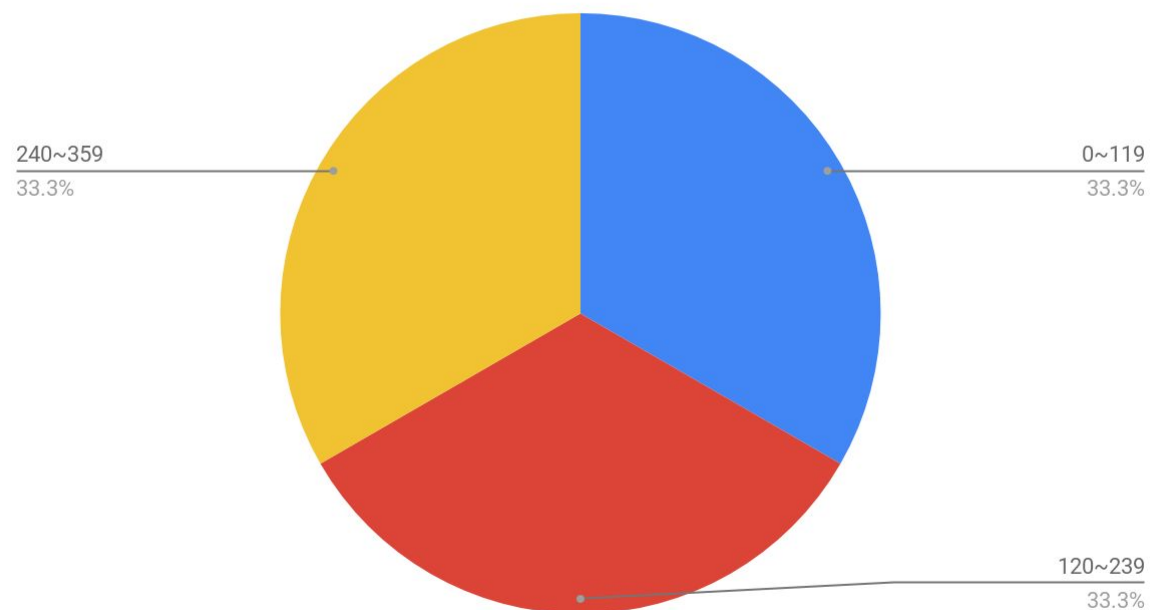
可以进行热迁移的。

可以通过实时备份, 不停服务迁移数据库, 备份完成且没有用户访问的额瞬间将主库切过去就可以了。

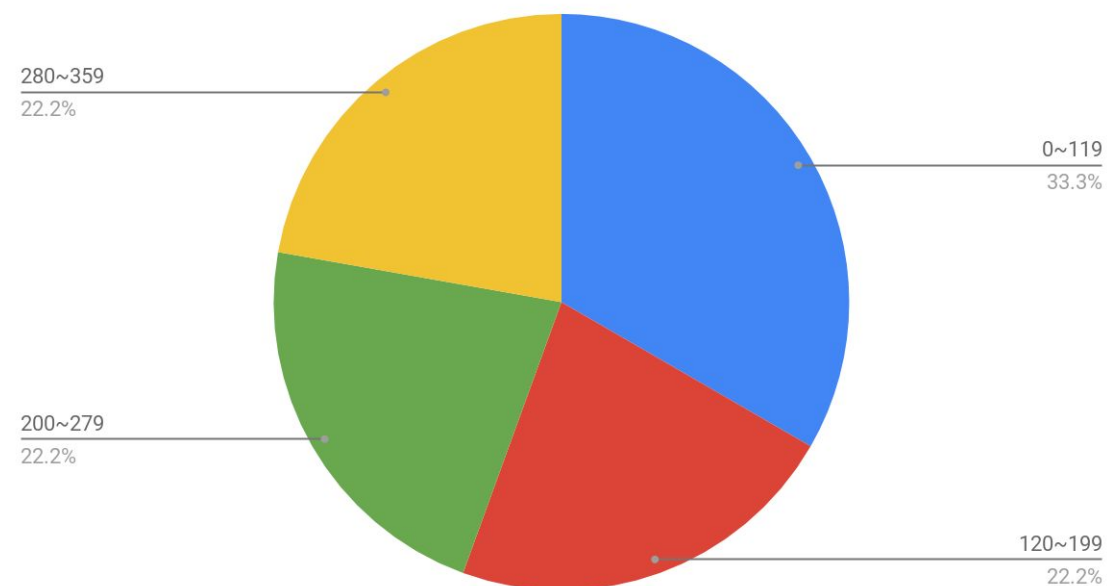


3台机器变4台机器的例子

3台机器时



4台机器时



提问:这种方法有什么缺陷?

缺陷1 数据分布不均匀

因为算法是“将数据最多的相邻两台机器均匀分为三台”
比如，3台机器变4台机器时，无法做到4台机器均匀分布

缺陷2 迁移压力大

新机器的数据只从两台老机器上获取
导致这两台老机器负载过大

哈希函数 Hash Function

<http://www.lintcode.com/problem/hash-function>

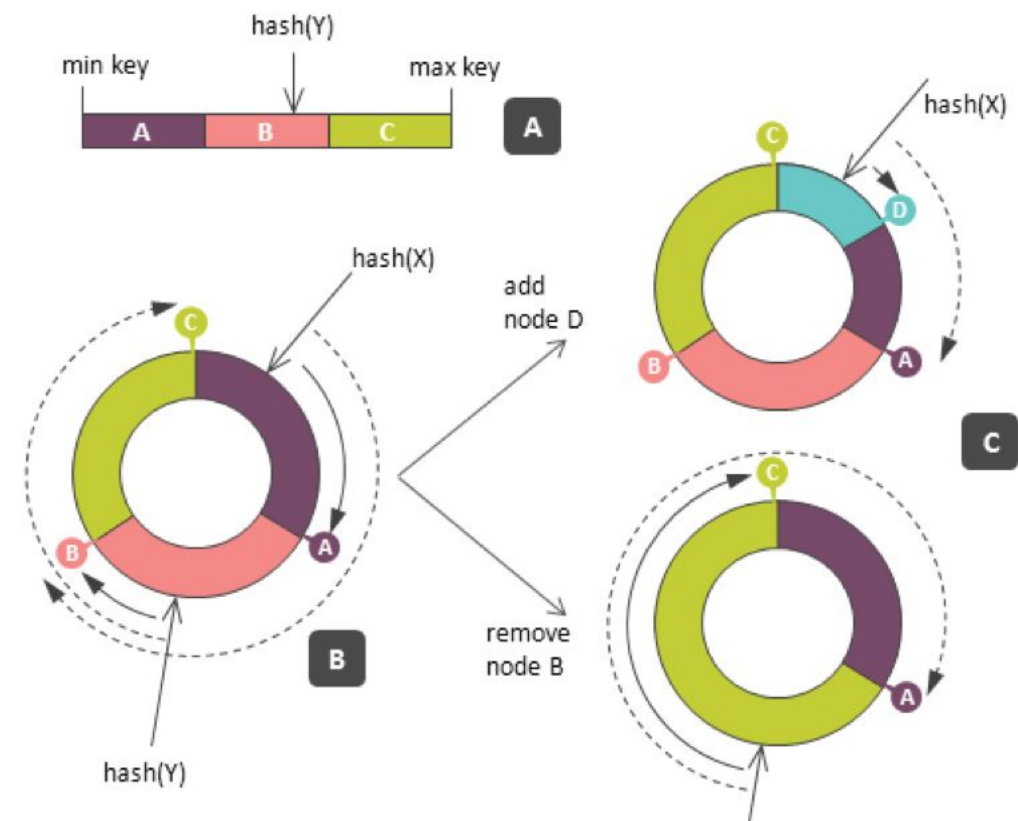
将任意类型的 key 转换为一个 $0 \sim \text{size}-1$ 的整数

在 Consistent Hashing 中, 一般取 $\text{size} = 2^{64}$


很多哈希算法可以保证不同的 key 算得相同的 hash 值的概率等于
宇宙爆炸的概率

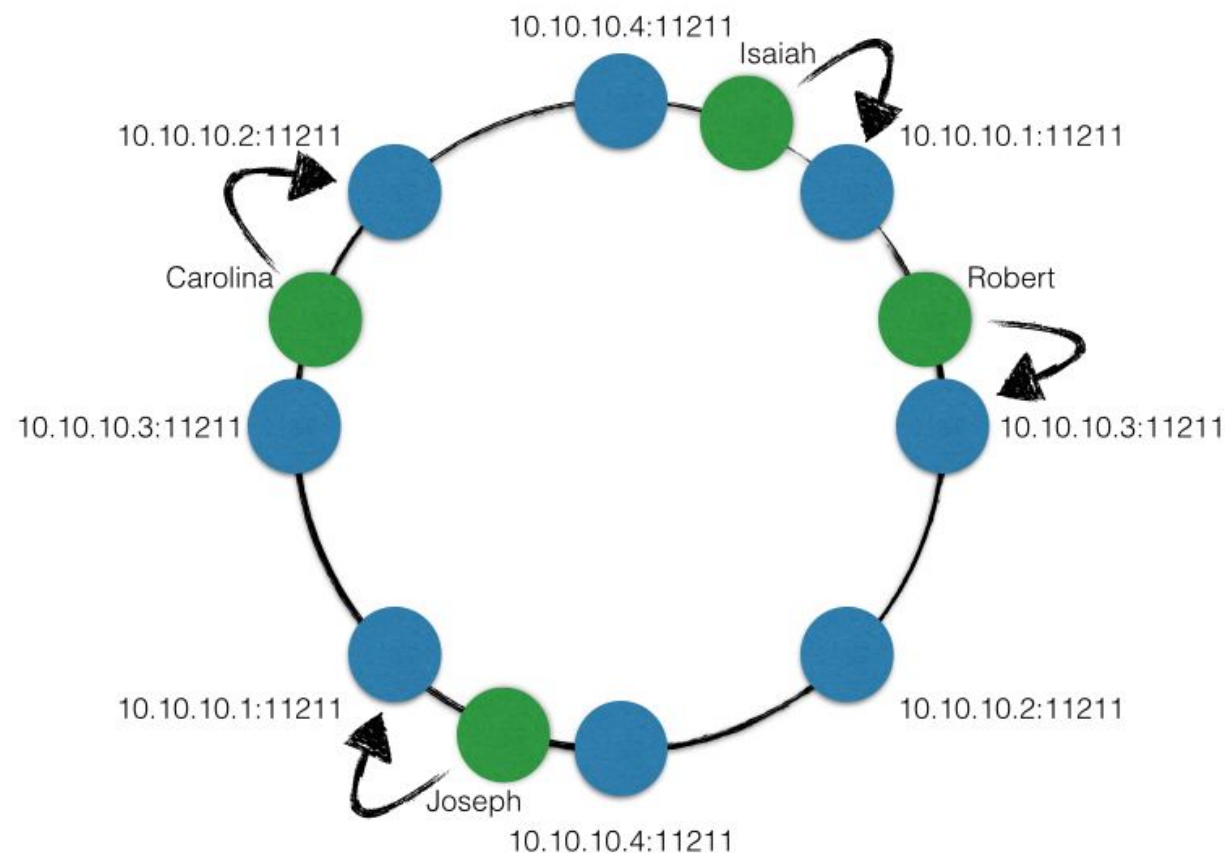
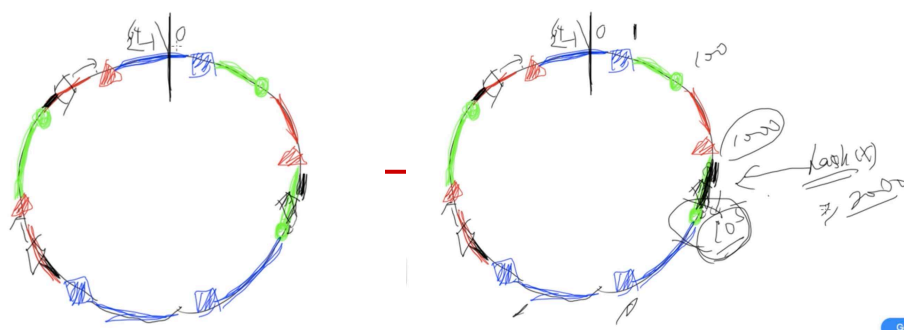
一致性哈希算法 Consistent Hashing

- 将取模的底数从 360 拓展到 2^{64}
- 将 $0 \sim 2^{64}-1$ 看做一个很大的圆环(Ring)
- 将数据和机器都通过 hash function 换算到环上的一个点
 - 数据取 key / row_key 作为 hash key
 - 机器取MAC地址, 或者机器固定名字如 database01, 或者固定的 IP 地址
- 每个数据放在哪台机器上, 取决于在 Consistent Hash Ring 上**顺时针**碰到的下一个机器节点



虚拟节点 Virtual Node

- 引入分身的概念 (Virtual nodes)
- 一个实体机器 (Real node) 对应若干虚拟机器 (Virtual Node)
 - 通常是 1000 个
 - 分身的KEY可以用**实体机器的KEY+后缀**的形式
 - 如 database01-0001
 - 好处是直接按格式去掉后缀就可以得到实体机器
- 用一个数据结构存储这些 virtual nodes
 - 支持快速的查询比某个数大的最小数
 - 即顺时针碰到的下一个 virtual nodes
 - 哪种数据结构可以支持？ 



新增一台机器

创建对应的 1000 个分身 db99-000 ~ 999

加入到 virtuals nodes 集合中

问: 该从哪些机器迁移哪些数据到新机器上? 

Consistent hashing 和 Load Balancing 不是一个意思。Consistent hashing 会让同样的 Request 去同一个 Server, Load Balancing 可能会让同样的 Request 去不同的 Server。

Web Server 上只存代码不存数据, 是 stateless 的, 更适合 Load Balancing。而 Database 上存数据, 且我们希望每个 Database server 能分开存储所有数据, 因此 Database 是 stateful 的, 更适合 Consistent Hashing。当然如果你想把 Consistent Hashing 用在 Web Server 也是可以的, 这样的好处是有时候 Web Server 上会有一些 Local Cache 用与降低 cache miss, 那么同样数据的请求经过同一个 Web Server 会更好。

Consistent Hashing

<http://www.lintcode.com/problem/consistent-hashing-ii/>

实现一遍, 印象更深刻

Replica 数据备份

问: Backup 和 Replica 有什么区别?



Backup

- 一般是周期性的, 比如每天晚上进行一次备份
- 当数据丢失的时候, 通常只能恢复到之前的某个时间点
- Backup 不用作在线的数据服务, 不分摊读

Replica

- 是实时的, 在数据写入的时候, 就会以复制品的形式存为多份
- 当数据丢失的时候, 可以马上通过其他的复制品恢复
- Replica 用作在线的数据服务, 分摊读


思考: 既然 Replica 更牛, 那么还需要 Backup 么? 

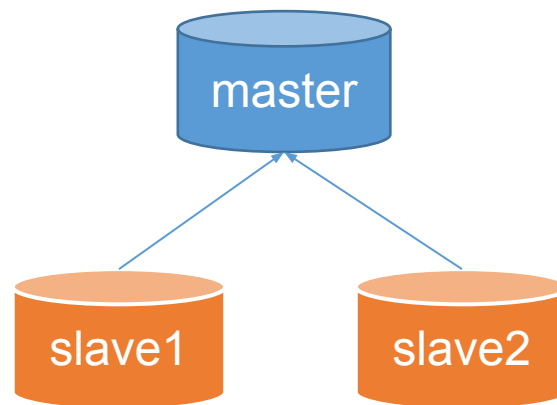
MySQL Replica

以MySQL为代表SQL型数据库, 通常“自带” Master Slave 的
Replica 方法

Master 负责写, Slave 负责读

Slave 从 Master 中同步数据

- 原理 Write Ahead Log 
 - SQL 数据库的任何操作, 都会以 Log 的形式做一份记录
 - 比如数据A在B时刻从C改到了D
 - Slave 被激活后, 告诉master我在了
 - Master每次有任何操作就通知 slave 来读log
 - 因此Slave上的数据是有“延迟”的
- Master 挂了怎么办?
 - 将一台 Slave 升级 (promote) 为 Master, 接受读+写
 - 可能会造成一定程度的数据丢失和不一致



NoSQL Replica

以 Cassandra 为代表的 NoSQL 数据库
通常将数据“顺时针”存储在 Consistent hashing 环上的三个 virtual nodes 中

SQL

“自带”的 Replica 方式是 Master Slave

“手动”的 Replica 方式也可以在 Consistent Hashing 环上顺时针存三份

NoSQL

“自带”的 Replica 方式就是 Consistent Hashing 环上顺时针存三份

“手动”的 Replica 方式:就不需要手动了, NoSQL就是在 Sharding 和 Replica 上帮你偷懒用的!

实战1: User Table 如何 Sharding

如果我们在 SQL 数据库中存储 User Table
那么按照什么做 Sharding ?

怎么取数据就怎么拆数据

How to shard data based on how to query data

绝大多数请求: `select * from user_table where user_id=xxx`

问如果我需要按照 username 找用户怎么办？



- User Table Sharding 之后, 多台数据库无法维护一个全局的**自增ID**怎么办?
 - 手动创建一个 UUID 来作为用户的 user_id
- 创建用户时还没有用户的 user_id, 如何知道该去哪个数据库创建呢?
 - Web Server 负责创建用户的 user_id, 如用 UUID 来作为 user_id
 - 创建之后根据 consistent_hash(user_id) 的结果来获得所在的实体数据库信息
- 更进一步的问题: 如果 User Table 没有 sharding 之前已经采用了自增ID该怎么办?
 - UUID 通常是一个字符串, 自增 id 是一个整数, 并不兼容
 - 单独用一个 UserIdService 负责创建用户的 ID, 每次有新用户需要创建时, 问这个 Service 要一个新的 ID。这个 Service 是全局共享的, 专门负责创建 UserId。负责记录当前 UserId 的最大值是多少了, 然后每次 +1 即可。这个 Service 会负责加锁来保证数据操作的原子性(Atomic)。
 - 因为创建用户不是一个很大的 QPS, 因此这个做法问题不大

1. 为什么全局维护 auto increment id 很难? 2. 为什么不能用原有的 auto-increment id 去做 consistent hashing sharding? 3. 似乎通过 userid service 也可以维护全局自增?

"类似多线程竞争读写同一个共享变量, 加锁效率低, 不加锁就会混乱。多个数据库肯定是部署在多台机器上, 维护这个 auto increment id 效率很低 (毕竟需要网络延迟), 但是是可以实现的。"

老师说的最后一点没太明白。如果是像 message table 这种 QPS 很大的数据, 然后需要迁移到 distributed db, 这个时候用 userservice 来保持全局自增 id 又效率很低, 此时应该怎么办呢?

使用 uuid, 可以根据 毫秒级时间戳+一段随机数 生成 uuid, 保证每一 ms 内生成相同的 uuid 基本概率尽量接近 0 就好。各个服务器各自同时生成各自的就好, 不用加锁, 保证效率。

没有太听懂“双向好友关系是否只能存储为一条数据”

“a关注了b，数据库中存储的是：
user1 -> user2
· | -
a -> b
b -> a
这里是存了两条数据，也就是通常的方式。

但是只存储一条的话就是：
user1 -> user2
· | -
a -> b
当需要查询b的好友有哪些的时候
user2列就需要全部遍历 + user1二分遍历，复杂度太高”

没有很明白为什么不能按照大id和小id去存双向好友关系，为什么不能保证用sharding key查到好友数据呢？

现在存储A和B是好友关系 ($A < B$)，然后查询B的好友有哪些？

怎么查询？在小id列查询的话是查询不到的，在大id列查询的话，查询复杂度是 $O(n)$ 啊。

单DB都是 $O(n)$ ，sharding后也是 $O(n)$ ，效率太低了。

实战2: Friendship Table 如何 Sharding

双向好友关系是**否**还能只存储为一条数据？

单向好友关系按照什么 sharding？

一条记录可以被sharding两次么？还是说这里follower和following是两张表单，那这两个表单存的不是一模一样么？A followed B 既需要存在follower并以to_user_id作为sharding key，这条关系又需要存在following里并以from_user_id作为sharding_key

“存储的数据是一样的，表结构不一样

```
fans_table
to_user_id(sharding key)from_user_id
...
follow_table
from_user_id(sharding key)to_user_id
...
```


实战3: Session Table 如何 Sharding

Session Table 主要包含 session_key(session token), user_id, expire_at 这几项



实战4: News Feed / Timeline 按照什么 Sharding ?

News Feed = 新鲜事列表
Timeline = 某人发的所有帖子



还记得 News Feed 是什么么? News Feed 是指我关注的人的信息流汇总, 也就是你朋友圈里看到的数据。News Feed Table 大概有如下一些 columns:

```
id - primary key, 系统自动生成的unique id
owner_id - Foreign key, 放在谁的 news feed 里
post_id - Foreign key, 帖子的 id
posted_by_user_id - Foreign key, 谁发的
created_at - timestamp, 啥时候发的, 用于排序
...
```

Timeline 是某个人发的所有的帖子的汇总。就是点到某个人的朋友圈以后看到的他/她发的所有帖子。Timeline 其实就是 Post Table 按照时间排序, 里大致有如下一些 columns:

```
id - primary key, 系统自动生成的 unique id
user_id - foreign key, 谁发的
content - text, 发了啥
created_at - timestamp, 啥时候发的
...
```

实战5: LintCode Submission 按照什么 Sharding

Submission 包含了, 谁(user)什么时候(timestamp)提交了哪个题(problem)得到了什么判定结果(status)

需求1: 查询某个题的所有提交记录

```
select * from submission_table where problem_id=1001;
```

需求2: 查询某个人的所有提交记录

```
select * from submission_table where user_id=101;
```

单纯按照 user_id 或者 problem_id sharding 都无法同时满足两个查询需求。

解决办法(两个表单):

submission_table 按照 user_id sharding, 因为按照 user_id 的查询操作相对频繁一些。

在 submission_table 之外再建立一张表单, 记录某个题有哪些提交记录, 表单包含 <problem_id, user_id, submission_id, ...> 等信息, 以 problem_id 作为 sharding key。

Consistent Hashing

<http://bit.ly/1XU9uZH>



<http://bit.ly/1KhqPEr>

FAQ:

<http://www.jiuzhang.com/qa/1828/>

<http://www.jiuzhang.com/qa/1417/>

<http://www.jiuzhang.com/qa/980/>