从用户系统中理解数据库与缓存

Design User System - Database & Memcache

课程版本: v5.1　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

今日课程大纲

- Design User System
  - Memcached
  - Authentication
  - SQL vs NoSQL
  - Friendship
- How to scale?
  - Sharding
  - Consistent Hashing（第五节课）
  - Replica（第五节课）

Design User System

设计用户系统

实现功能包括注册、登录、用户信息查询

* 好友关系存储

4S - Scenario, Service, Storage, Scale

- Scenario 场景
  - 注册、登录、查询、用户信息修改
    - 哪个需求量最大？
  - 支持 100M DAU
  - 注册，登录，信息修改 QPS 约
    - 100M * 0.1 / 86400 ~ 100
    - 0.1 = 平均每个用户每天登录+注册+信息修改
    - Peak = 100 * 3 = 300
  - 查询的QPS 约
    - 100 M * 100 / 86400 ~ 100k
    - 100 = 平均每个用户每天与查询用户信息相关的操作次数(查看好友，发信息，更新消息主页)
    - Peak = 100k * 3 = 300 k
- Service 服务
  - 一个 AuthService 负责登录注册
  - 一个 UserService 负责用户信息存储与查询
  - 一个 FriendshipService 负责好友关系存储

QPS 与 系统设计的关系

为什么要分析 QPS?

QPS 的大小决定了数据存储系统的选择

4S - Storage: QPS 与 常用数据存储系统

- MySQL / PosgreSQL 等 SQL 数据库的性能
  - 约 1k QPS 这个级别
- MongoDB / Cassandra 等 硬盘型NoSQL 数据库的性能
  - 约 10k QPS 这个级别（
- Redis / Memcached 等 内存型NoSQL 数据库的性能
  - 100k ~ 1m QPS 这个级别
- 以上数据根据机器性能和硬盘数量及硬盘读写速度会有区别

- 思考：
- 注册，登录，信息修改，300 QPS，适合什么数据存储系统？
- 用户信息查询适合什么数据存储系统？

用户系统特点

读非常多，写非常少

一个读多写少的系统，一定要使用 Cache 进行优化

Storage: Cache

- Cache 是什么？
  - 缓存，把之后可能要查询的东西先存一下
    - 下次要的时候，直接从这里拿，无需重新计算和存取数据库等
  - 可以理解为一个 Java 中的 HashMap
  - key-value 的结构
- 有哪些常用的 Cache 系统/软件？
  - Memcached（不支持数据持久化）
  - Redis（支持数据持久化）
- Cache 一定是存在内存中么？
  - 不是
  - Cache 这个概念，并没有指定存在什么样的存储介质中
  - File System 也可以做Cache
  - CPU 也有 Cache
- Cache 一定指 Server Cache 么？
  - 不是，Frontend / Client / Browser 也可能有客户端的 Cache

Mem-Cache

存在内存中的Cache
是一个“概念”

Memcached

一款负责帮你Cache在内存里的“软件”

非常广泛使用的数据存储系统

Memcached 使用例子


```
cache.set("this is a key", "this is a value")
cache.get("this is a key")
>> "this is a value"

cache.set("foo", 1, ttl=60)
cache.get("foo")
>> 1

# wait for 60 seconds
cache.get("foo")
>> null

cache.set("bar", "2")
cache.get("bar")
>> "2"

# for some reason like out of memory
# "bar" may be evicted by cache
cache.get("bar")
>> null
```

Memcached 如何优化 DB 的查询

```python
class UserService:

    def getUser(self, user_id):
        key = "user::%s" % user_id
        user = cache.get(key)
        if user:
            return user
        user = database.get(user_id)
        cache.set(key, user)
        return user

    def setUser(self, user):
        key = "user::%s" % user.id
        cache.delete(key)
        database.set(user)
```

Memcached 如何优化 DB 的查询

下面哪种写法是对的：

A: database.set(user); cache.set(key, user);

B: cache.set(key, user); database.set(user);

C: cache.delete(key); database.set(user);

D: database.set(user); cache.delete(key);

Memcached 如何优化 DB 的查询

cache.**set**(key, user)

// 此时 database 挂了

database.set(user) → 失败

此时会造成cache和database的数据不一致。导致下一次 get 的时候得到的是 cache 里的脏数据。

cache.**delete**(key)

// 此时 database 挂了

database.set(user) → 失败

这种情况下，不会出现数据不一致，只是cache里被删除了key，下次需要重新load。

用户会收到修改信息失败的提示，用户自己可以选择再点一次“保存”进行重试。

Authentication Service

- 用户是如何实现登陆与保持登陆的？
- 会话表 Session
- 用户 Login 以后
  - 创建一个 session 对象
  - 并把 session_key 作为 cookie 值返回给浏览器
  - 浏览器将该值记录在浏览器的 cookie 中
  - 用户每次向服务器发送的访问，都会自动带上该网站所有的 cookie
  - 此时服务器检测到cookie中的session_key是有效的，就认为用户登陆了
- 用户 Logout 之后
  - 从 session table 里删除对应数据
- 问题：Session Table 存在哪儿？
  - A: 数据库
  - B: 缓存
  - C: 都可以

Session Table		
session_key	string	一个 hash 值，全局唯一，无规律
user_id	Foreign key	指向 User Table
expire_at	timestamp	什么时候过期

Session Table 存在哪儿？

一般来说，都可以，即便存在 Cache 里

断电了相当于让所有用户都 logout 也没啥大不了

存在数据库里肯定更好

如果访问多的话，就用 Cache 做优化即可

小结

- 对于 User System 而言
  - 写很少
  - 读很多
- 写操作很少，意味着
  - 从QPS的角度来说，一台 MySQL 就可以搞定了
- 读操作很多，意味着
  - 可以使用 Memcached 进行读操作优化

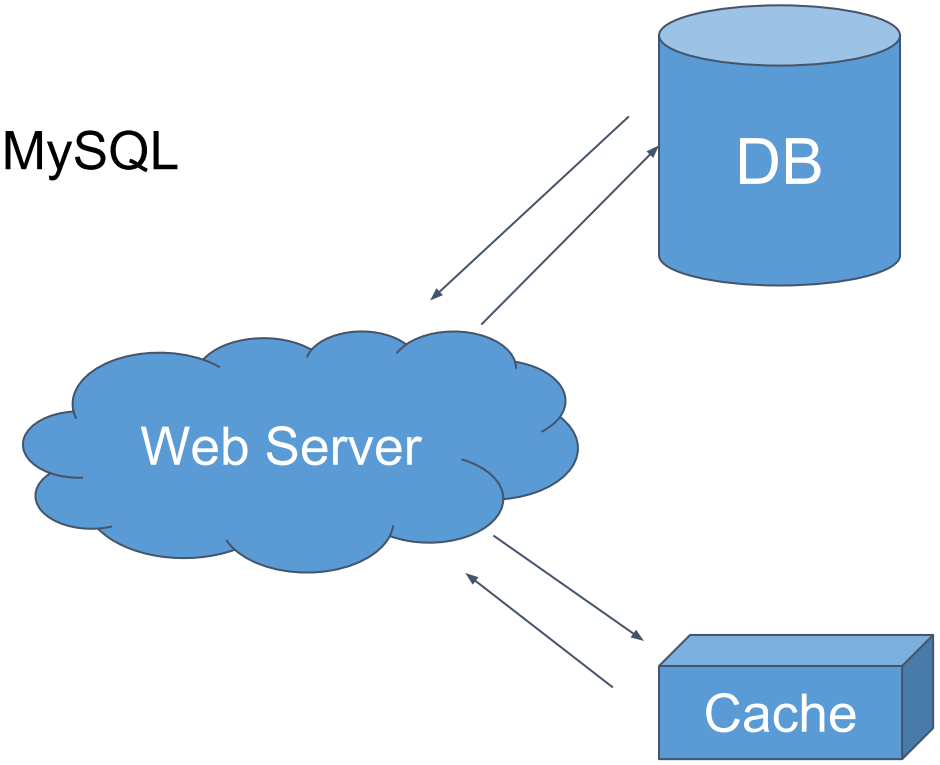
- 进一步的问题，如果读写操作都很多，怎么办？
  - 方法一：使用更多的数据库服务器分摊流量
  - * 方法二：使用像 Redis 这样的读写操作都很快的 Cache-through 型 Database
    - * Memcached 是一个 Cache-aside 型的 Database，Client 需要自己负责管理 Cache-miss 时数据的 loading

Cache Aside

服务器分别与 DB 和 Cache 进行沟通

DB 和 Cache之间不直接沟通

业界典型代表：Memcached + MySQL

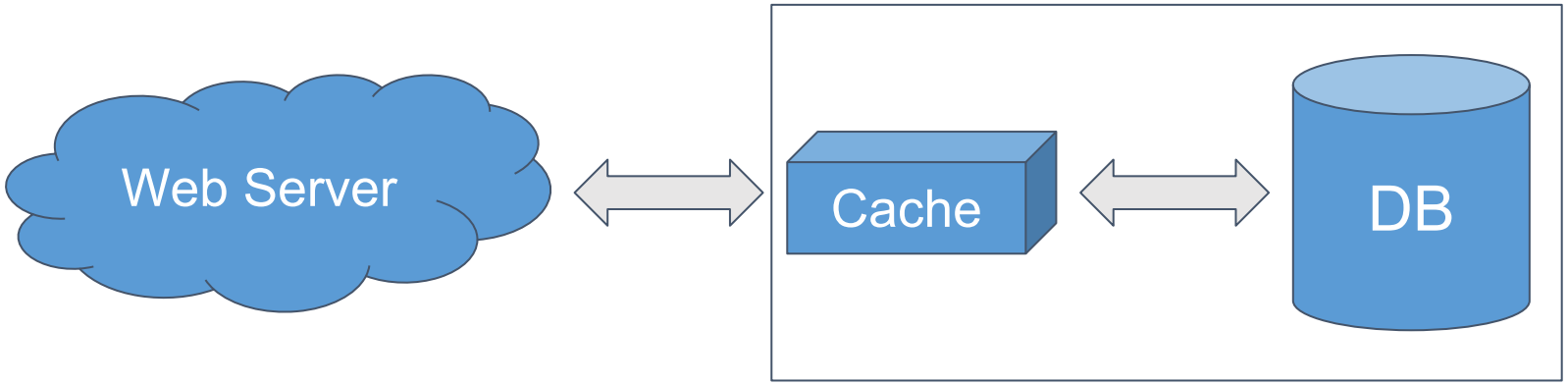


Cache Through

服务器只和 Cache 沟通

Cache 负责 DB 去沟通，把数据持久化

业界典型代表：Redis（可以理解为 Redis 里包含了一个 Cache 和一个 DB）



User System 里程碑

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端)/ Cookie（存在浏览器端）

Take a break

5 minutes

Friendship Service

- 单向好友关系（Twitter、Instagram、微博）

Friendship Table		
from_user_id	Foreign key	用户主体
to_user_id	Foreign key	被关注的人

- 双向好友关系（WhatsApp、Facebook 、微信）
  - 方案1：存为两条信息，A关注了B，B关注了A
  - 方案2：存为一条信息，但查询的时候需要查两次
- 好友关系所涉及的操作非常简单，基本都是 key-value：
  - 求某个 user 的所有关注对象
  - 求某个 user 的所有粉丝
  - A 关注 B → 插入一条数据
  - A 取关 B → 删除一条数据

SQL vs NoSQL

Friendship Table适合什么数据库？

SQL 和 NoSQL 的选择标准是什么？

数据库选择原则1

大部分的情况，用SQL也好，用NoSQL也好，都是可以的

数据库选择原则2

需要支持 Transaction 的话不能选 NoSQL

数据库选择原则3

你想不想偷懒很大程度决定了选什么数据库

SQL更成熟帮你做了很多事儿

NoSQL很多事儿都要亲力亲为(Serialization, Secondary Index)

数据库选择原则4

如果想省点服务器获得更高的性能，NoSQL就更好

硬盘型的NoSQL比SQL一般都要快10倍以上

Friendship Service

- 如果存在SQL

Friendship Table		
smaller_user_id	Foreign key	双向好友关系中id小一点的,index=true
bigger_user_id	Foreign key	双向好友关系中id大一点的,index=true

- 查询好友关系时
  - 对于给定的 user_id
  - select bigger_user_id from friendship where smaller_user_id = $user_id
  - select smaller_user_id from friendship where bigger_user_id = $user_id
- 如果存在 NoSQL
- 很多 NoSQL 一般来说不支持 Multi Indexes
- 所以需要拆分为两条数据
  - A的好友有B: key=A, value=B
  - B的好友有A: key=B, value=A

存在 SQL 时的例子

方案 1：好友关系存一份（1和2是好友，2和3是好友，3和1是好友）

Friendship Table	
smaller_user_id	bigger_user_id
1	2
1	3
2	3

查询2的好友：

```
SELECT * from friendship
WHERE bigger_user_id = 2 OR
 smaller_user_id = 2;
```

方案2：好友关系存两份（1和2是好友，2和3是好友，3和1是好友）

Friendship Table	
from_user_id	to_user_id
1	2
2	1
1	3
3	1
2	3
3	2

查询2的好友：

```
SELECT * from friendship
WHERE from_user_id=2
```

以 Cassandra 为例剖析典型的 NoSQL 数据结构

- Cassandra 是一个三层结构的 NoSQL 数据库
  - http://www.lintcode.com/problem/mini-cassandra/
- 第一层：row_key
  - 又称为 hash_key
  - 也就是我们传统所说的 key-value 中的 那个key
  - 任何的查询都需要带上这个key，无法进行range query
  - 最常用的row_key: user_id
- 第二层：column_key
  - 是排序的，可以进行range query
  - 可以是复合值，比如是一个 timestamp + user_id 的组合
- 第三层：value
  - 一般来说是 String
  - 如果你需要存很多的信息的话，你可以自己做 Serialization
    - 什么是 Serialization: 把一个 object / hash 序列化为一个 string，比如把一棵二叉树序列化
    - http://www.lintcode.com/problem/binary-tree-serialization/

Row Key

又称为 Hash Key, Partition Key

Cassandra 会根据这个 key 算一个 hash 值

然后决定整条数据存储在哪儿

Column Key

insert(row_key, column_key, value)

任何一条数据，都包含上面三个部分

你可以指定 column_key 按照什么排序

Cassandra 支持这样的“范围查询”：

query(row_key, column_start, column_end)

SQL vs NoSQL

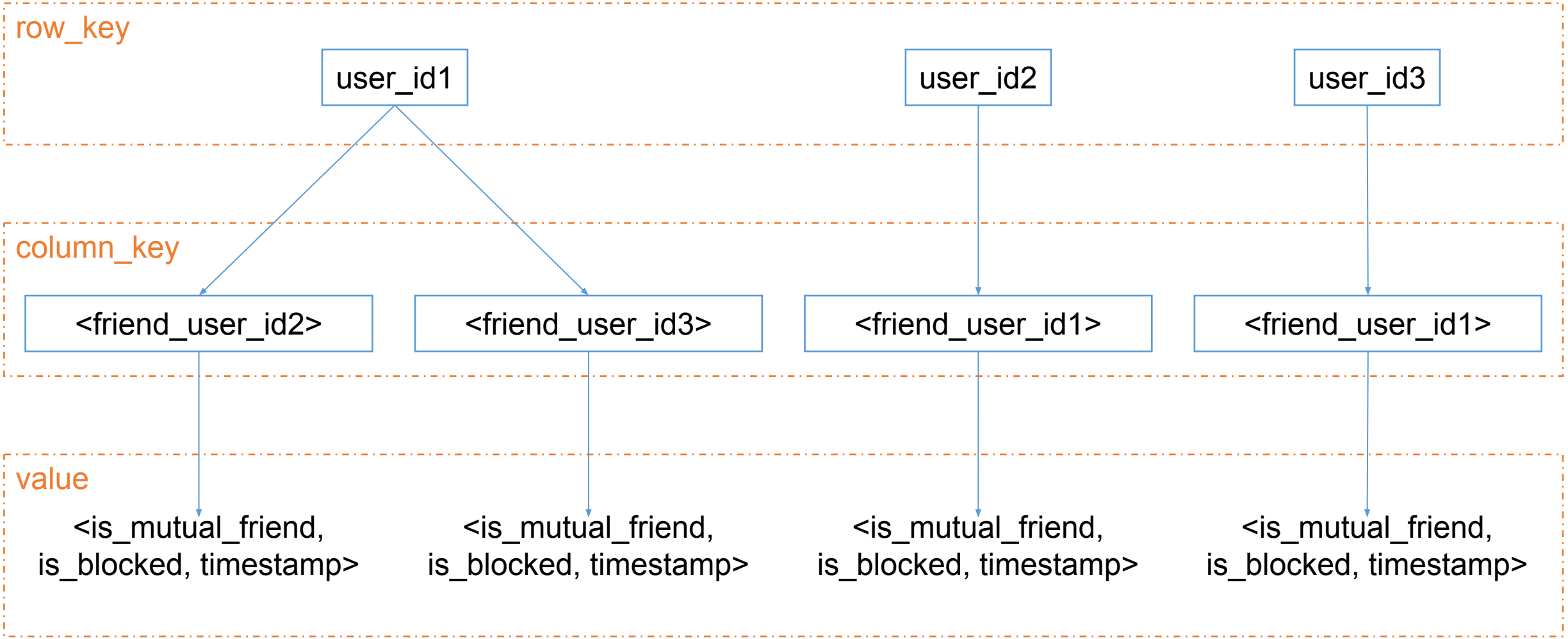
- SQL的column是在Schema中预先指定好的，不能随意添加
- 一条数据一般以 row 为单位（取出整个row作为一条数据）

SQL	id	username	email	password	first_name	…
row1						
row2						
…						

- NoSQL的column是动态的，无限大，可以随意添加
- 一条数据一般以 grid 为单位，row_key + column_key + value = 一条数据
- 只需要提前定义好 column_key 本身的格式（是一个 int 还是一个 int+string）

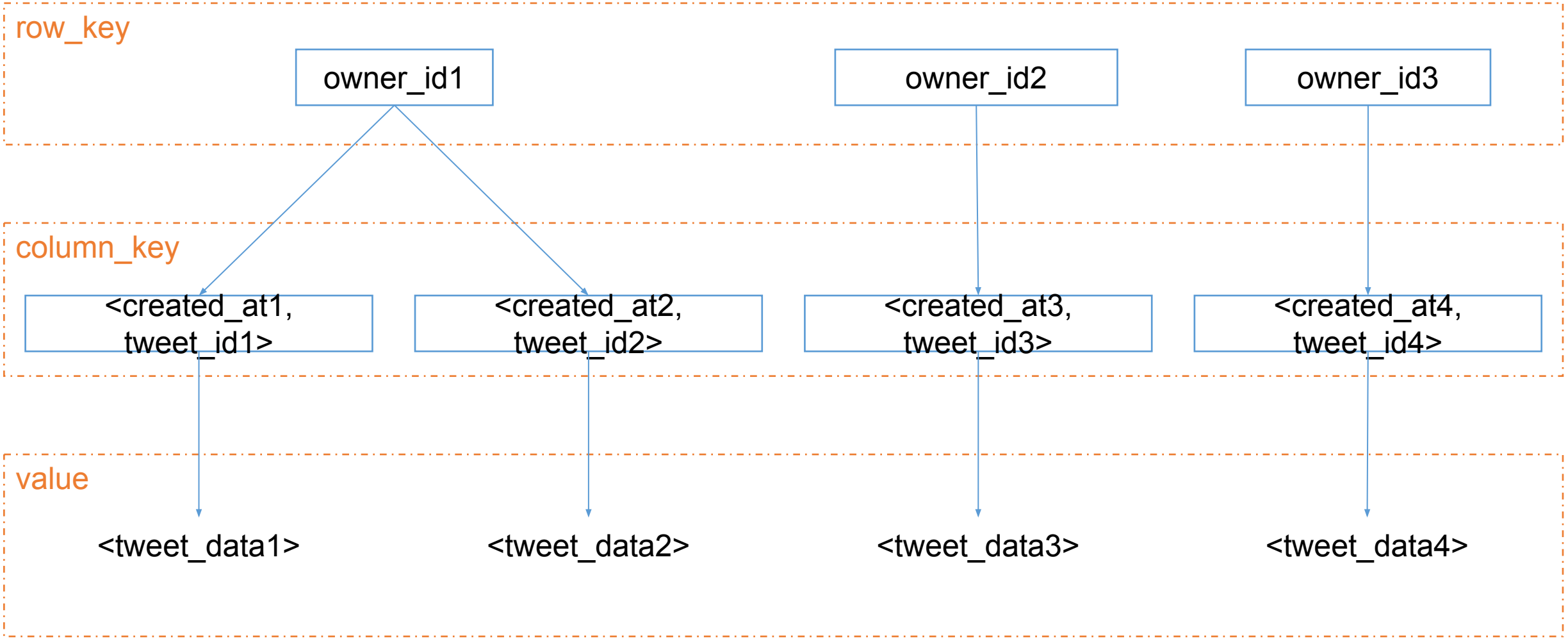
NoSQL	column_key1	column_key2	column_key3	column_key4	…
row_key1	value0	value1	...		
row_key2	...	...			
...					

以 Cassandra 为例看看 Friendship Table 如何存储



如果要查询最近1天关注的好友怎么办？

例子2：Cassandra 如何存储 NewsFeed



Cassandra 例子和总结

Cassandra为代表的NoSQL 存储数据相当于存储一个一个的三元组，比如存储好友关系时：

1和2是好友，2和3是好友会存成下面4条数据：

row_key	column_key	value
1	2	null
2	1	null
2	3	null
3	2	null

User System 里程碑

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端)/ Cookie（存在浏览器端）
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储

Interviewer: How to scale?

系统设计的日经题

除了QPS，还有什么需要考虑的？

100M 的用户存在一台 MySQL 数据库里也存得下，Storage没问题

通过 Cache 优化读操作后，只有 300QPS 的写，QPS也没问题

还有什么问题？

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. Sharding
2. Replica

Sharding / Replica

- 数据拆分 Sharding
  - 按照一定的规则，将数据拆分成不同的部分，保存在不同的机器上
  - 这样就算挂也不会导致网站 100% 不可用
- 数据备份 Replica
  - 通常的做法是一式三份（重要的事情“写”三遍）
  - Replica 同时还能分摊读请求

Sharding in SQL vs NoSQL

SQL自身不带 Sharding 功能，需要码农亲自上手

Cassandra为例的NoSQL大多数都自带 Sharding

这就是为什么程序员要发明 NoSQL ---- 可以偷懒！

数据拆分 Sharding

Vertical Sharding

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
  - push_preference
  - avatar
- 我们知道 email / username / password 不会经常变动
- 而 push_preference, 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 User Profile Table
  - 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

- 提问：Vertical Sharding 的缺点是什么？他不能解决什么问题？

横向切分
Horizontal Sharding

核心部分！

Scale 的核心考点！

一个粗暴的想法

假如我们来拆分 Friendship Table

我们有10台数据库的机器

于是想到按照 from_user_id % 10 进行拆分

这样做的问题是啥？

假如10台机器不够用了

我现在新买了1台机器

原来的%10，就变成了%11

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，牵一发动全身

2. 迁移期间，服务器压力增大，容易挂

3. 容易造成数据的不一致性

怎么办？

一致性 Hash 算法

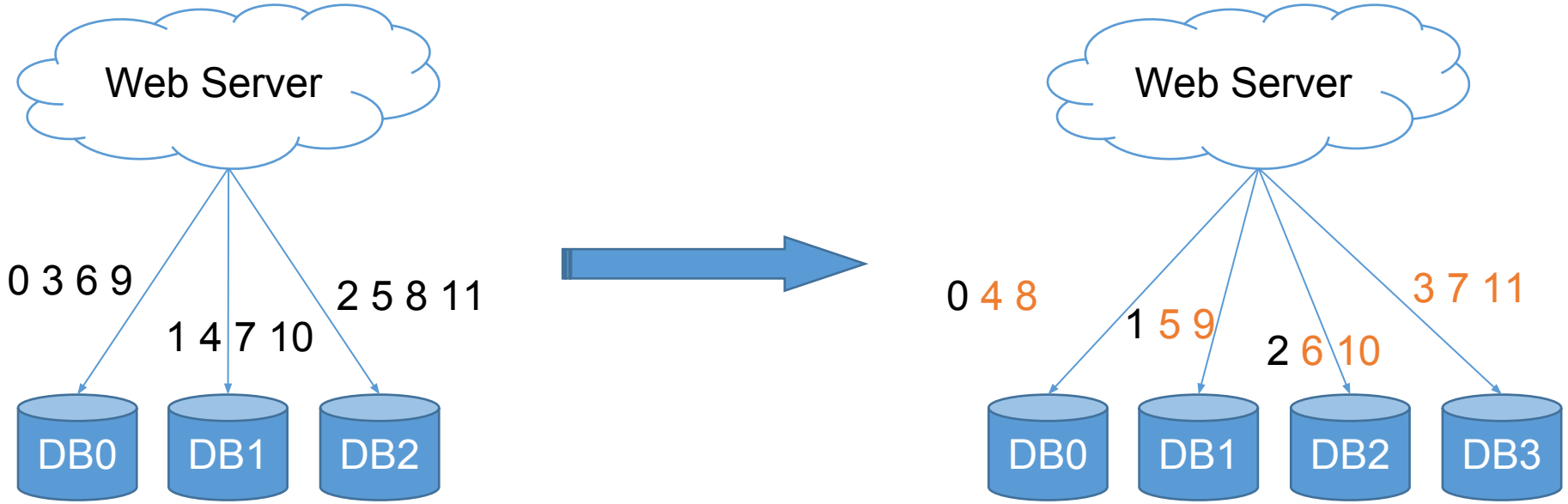
Consistent Hashing

我们将在第五节课中，为大家介绍该算法，且听下回分解

下面几页PPT供大家预习

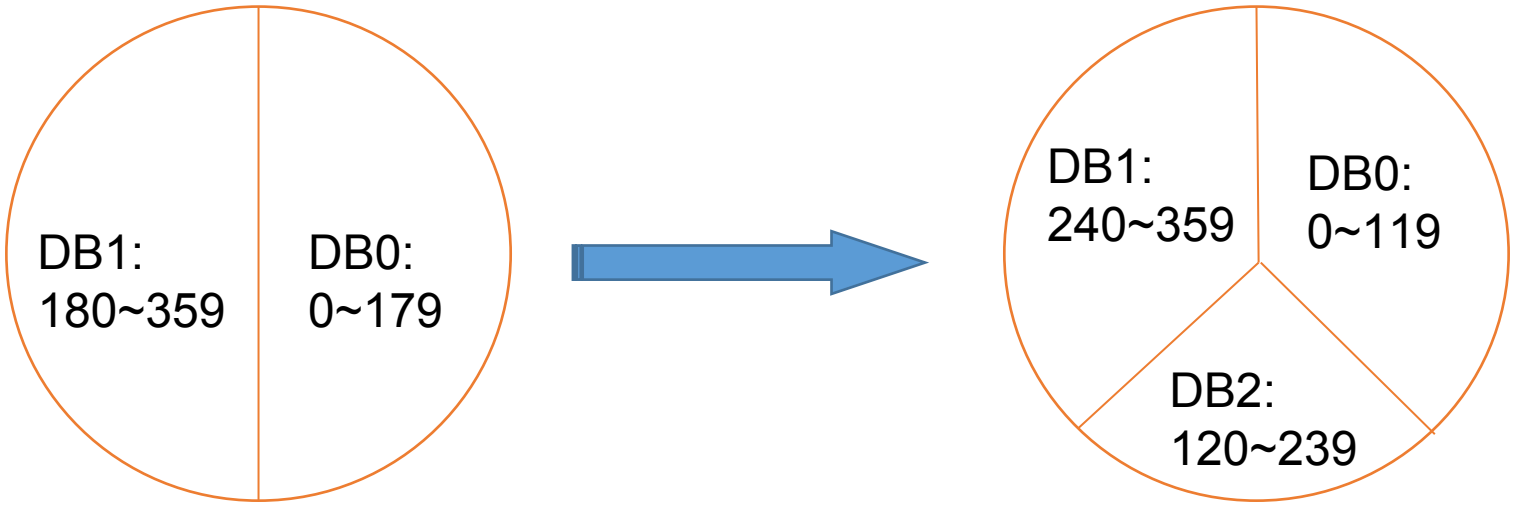
Consistent Hashing

- 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 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



Consistent Hashing

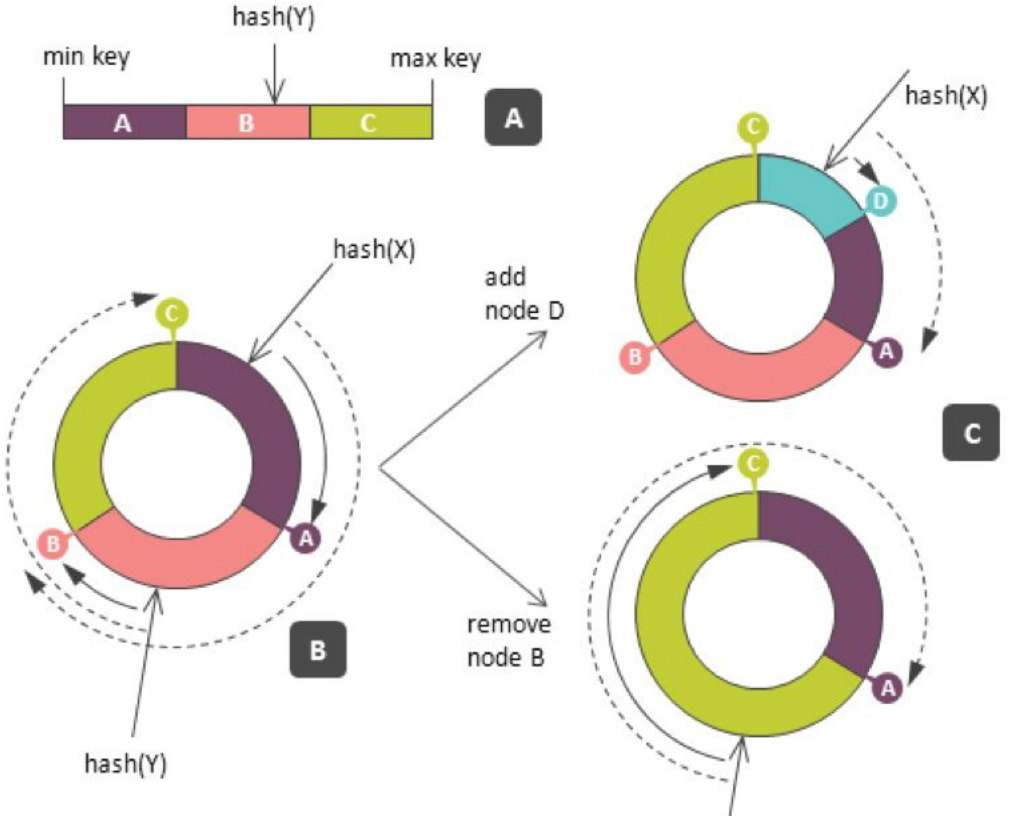
- 一个简单的一致性Hash算法
  - 将 key 模一个很大的数，比如 360
  - 将 360 分配给 n 台机器，每个机器负责一段区间
  - 区间分配信息记录为一张表存在 Web Server 上
  - 新加一台机器的时候，在表中选择一个位置插入，匀走相邻两台机器的一部分数据
- 比如 n 从 2 变化到 3，只有 1/3 的数据移动



Consistent Hashing

- http://www.lintcode.com/problem/consistent-hashing/
- 每一次加入一台新机器
- 只有1台或者2台数据库的数据会被迁移
  - 因为要占据环上的一段区间

- 这是一个比较简单的 Consistent Hashing 的算法
- 提问:这种简单方法中，有什么缺陷？

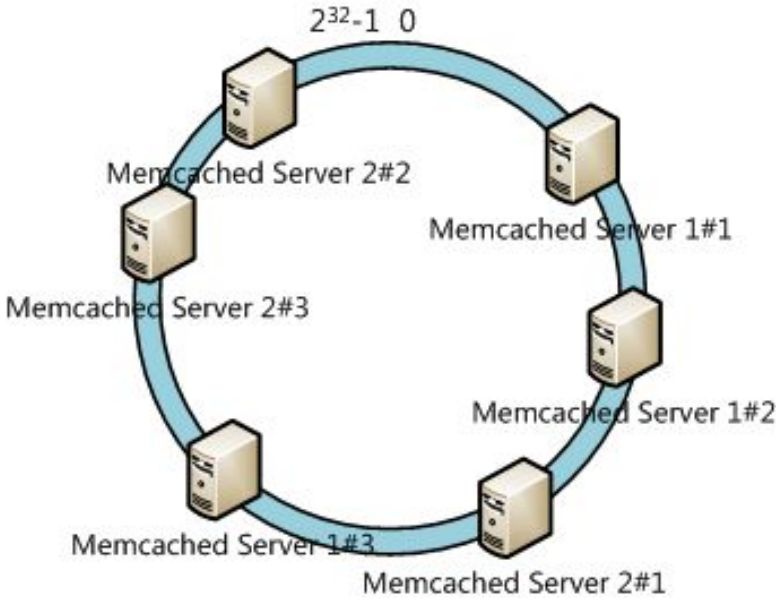


Consistent Hashing —— 简单版本的缺陷

- 增加一台服务器之后，该新服务器上的数据全从周围的1~2台服务上来
  - 这1~2台服务器的短时间内读压力增大很多，影响到正常服务
- 每次只能从1~2台机器匀数据
  - 数据分配无法做到绝对均匀

Consistent Hashing —— 更实用的方法

- 将整个 Hash 区间看做环
- 这个环的大小从 0~359 变为 0~2^{64}-1
- 将机器和数据都看做环上的点
- 引入 Micro shards / Virtual nodes 的概念
  - 一台实体机器对应 1000 个 Micro shards / Virtual nodes
- 每个 virtual node 对应 Hash 环上的一个点
- 每新加入一台机器，就在环上随机撒 1000 个点作为 virtual nodes
- 需要计算某个 key 所在服务器时
  - 计算该key的hash值——得到0~2^{64}-1的一个数，对应环上一个点
  - 顺时针找到第一个virtual node
  - 该virtual node 所在机器就是该key所在的数据库服务器
- 新加入一台机器做数据迁移时
  - 1000 个 virtual nodes 各自向**顺**时针的一个 virtual node 要数据
  - 例子：http://www.jiuzhang.com/qa/2067/



Consistent Hashing

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

Replica in SQL vs NoSQL

MySQL 数据库一般如何做 Replica?

Master - Slave

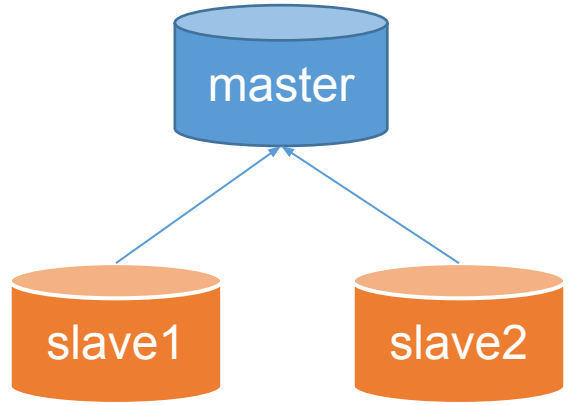
Master responsible for write / read

Slave responsible for read only

对于大部分系统来说基本够用

Master - Slave

- 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 一般如何做 Replica?

重要的数据存三份——顺时针找3台机器

SQL vs NoSQL

SQL需要自己做Sharding，自己管理Replica

——也就是要写好多代码

而对于NoSQL这一切基本都是自动的

——也就是可以偷懒

User System 里程碑

- 使用 Cache 优化数据库的读操作
- Session (存在服务器端)/ Cookie（存在浏览器端）
- SQL vs NoSQL的一些基本原则
- Friendship 在 Cassandra (NoSQL) 中如何存储
- SQL / NoSQL 数据库如何进行 Sharding
- 一致性 Hash 算法 Consistent Hashing
- SQL 如何进行 Replica – Master-slave
- NoSQL 数据库如何进行 Replica —— 顺时针存三份

附录：扩展阅读

- Dynamo DB —— 理解分布式数据库(NoSQL)的原理
  - http://bit.ly/1mDs0Yh [Hard] [Paper]
- Scaling Memcache at Facebook —— 妈妈再也不担心我的 Memcache
  - http://bit.ly/1UlpbGE [Hard] [Paper]
- Consistent Hashing
  - http://bit.ly/1KhqPEr [Medium] [Blog]
  - http://bit.ly/1XU9uZH [Medium] [Blog]
- Coach Base Architecture
  - http://horicky.blogspot.in/2012/07/couchbase-architecture.html
- Least Frequently Used Cache (LFU)
  - http://dhruvbird.com/lfu.pdf
- 作业：http://www.lintcode.com/ladder/8/

附录:NoSQL，也就是所谓的分布式数据库

- 分布式数据库解决的问题
  - Scalability
- 分布式数据库还没解决很好的问题
  - Query language
  - Secondary index
  - ACID transactions
  - Trust and confidence

附录：Storage, Network

	IOPS	Latency	Throughput	Capacity
Memory	(10M)	100ns	10GB/s	100GB
Flash	(100K)	(10us)	1GB/s	1TB
Hard Drive	100	10ms	100MB/s	1TB

	Rack	Datacenter	远距离
P99 Latency	<1ms	1ms	100ms +
Bandwidth	1GB/s	100MB/s	10MB/s -

NoSQL 和 SQL 的选取，面试的时候怎么答？

http://www.jiuzhang.com/qa/1836/

Friendship 的存储和查询的相关问题

http://www.jiuzhang.com/qa/1878/

Consistent Hashing 的相关问题

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

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

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