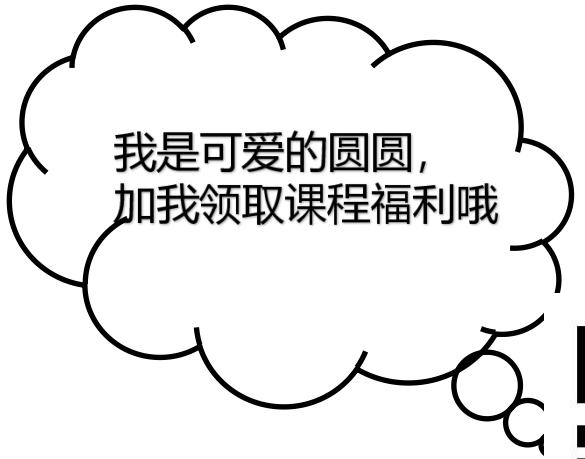
从用户系统设计中学习数据库与缓存

Design User System - Database & Cache



讲师：东邪

扫描上方二维码加圆圆
快速获取面试资料/课程福利

关注公众号，了解大厂资讯

本章课程大纲

- 使用 4S 分析法分析用户系统
- 缓存是什么 Cache
- 缓存和数据库如何配合 Cache & Database
- 登陆系统如何做 Authentication Service
- 好友关系的存储与查询 Friendship Service
- 以 Cassandra 为例了解 NoSQL 型数据库
- 关系型与非关系型数据库的适用场景比较 SQL vs NoSQL
- 拓展真题：
  - NoSQL 单向好友关系
  - 如何按照 email / username / phone / id 同时检索 User
  - 共同好友查询
  - Linkedin 六度好友关系·

Design User System

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 AuthenticationService 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

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)
```

“%s” 是 Python 的语法，表示把后面的 user_id 填入字符串的这个位置

Memcached 如何优化 DB 的查询

下面哪些写法是不对的？

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

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

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

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

Cache 和 Database 的操作都不保证一定成功

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

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

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

以上三个选项，如果第一个操作成功了，第二个操作失败了，都会导致数据库和缓存中的数据不一致（inconsistent）。我们称之为“脏数据”（Dirty Data）

问：有没有可能第一个失败，第二个成功？

Cache 和 Database 的操作都不保证一定成功

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

D选项中，不会因为第一个操作成功，而第二个操作不成功造成数据不一致。因为 cache.delete 是删除缓存中的数据，而不是修改缓存中的数据，第二个操作失败以后，信息相当于没有被修改，虽然操作失败了，但是没有造成缓存与数据库的数据不一致。

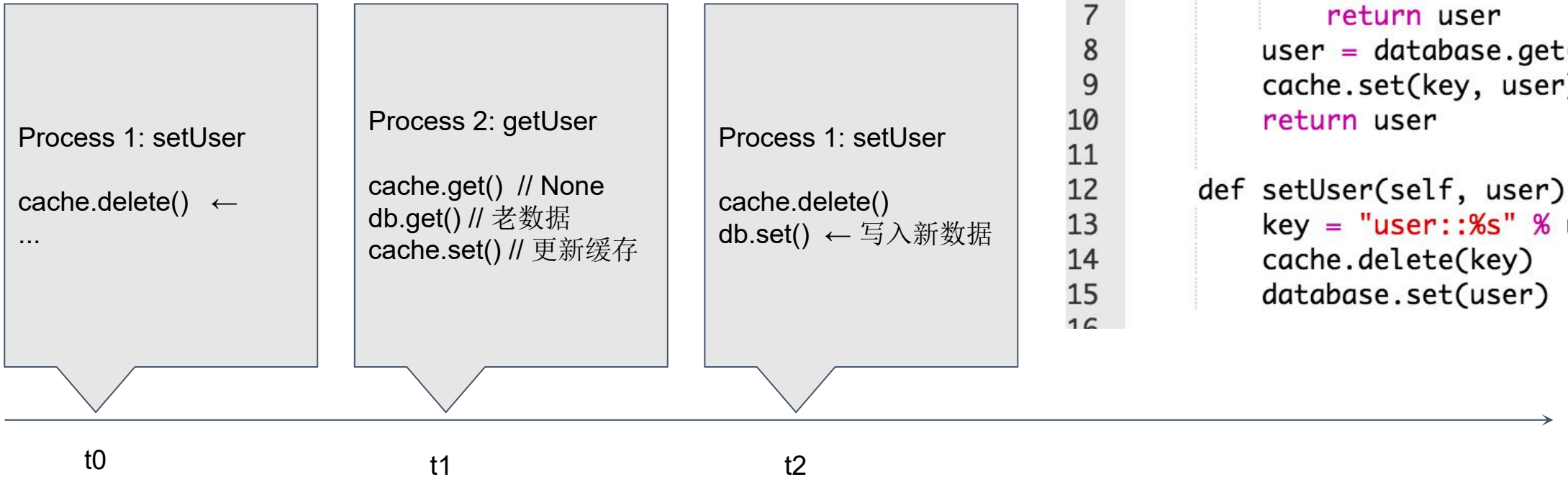
但是 D 这个选项仍然存在问题，请问在什么情况下会造成数据不一致？

多线程多进程下的数据不一致

在 setUser 执行到14行和15行之间的时候

另外一个进程执行了 getUser

此时 cache 里的数据是旧数据




```
1  class UserService:
2
3      def getUser(self, user_id):
4          key = "user::%s" % user_id
5          user = cache.get(key)
6          if user:
7              return user
8          user = database.get(user_id)
9          cache.set(key, user)
10         return user
11
12     def setUser(self, user):
13         key = "user::%s" % user.id
14         cache.delete(key)
15         database.set(user)
```

解决办法

可以给数据库和缓存的两个操作加锁么？

解决办法

可以给数据库和缓存的两个操作加锁么？

不行，数据库和缓存是两台机器，两套系统，并不支持加锁

如果是用一些第三方分布式锁，会导致存取效率降低，得不偿失

Best practice: database.set(key, user); cache.delete(key)

```
1  class UserService:
2
3      def getUser(self, user_id):
4          key = 'user::%s' % user_id
5          user = cache.get(user)
6          if user:
7              return user
8
9          user = database.get(user_id)
10         cache.set(key, user)
11         return user
12
13     def setUser(self, user):
14         key = 'user::%s' % user.id
15         database.set(user)
16         cache.delete(key)
```

问题1：在多线程多进程的情况下依旧会出问题

在getUser执行到第9行和第10行之间时

另外一个进程执行了 setUser()，cache 里会放入旧数据

问题2：db set 成功，cache delete 失败

好处：上面这两种情况发生概率都远低于 cache.delete + db.set

为什么？

如何“解决”一致性问题

巧妙利用 cache 的 ttl（time to live / timeout） 机制

任何一个 cache 中的 key 都不要永久有效，设置一个短暂的有效时间，如 7 天

那么即便在极低概率下出现了数据不一致，也就最多不一致 7 天

即，我们允许数据库和缓存有“短时间”内的不一致，但最终会一致。

如果写很多怎么办？

在每次数据修改的时候，我们会在 cache 中 delete 这个数据

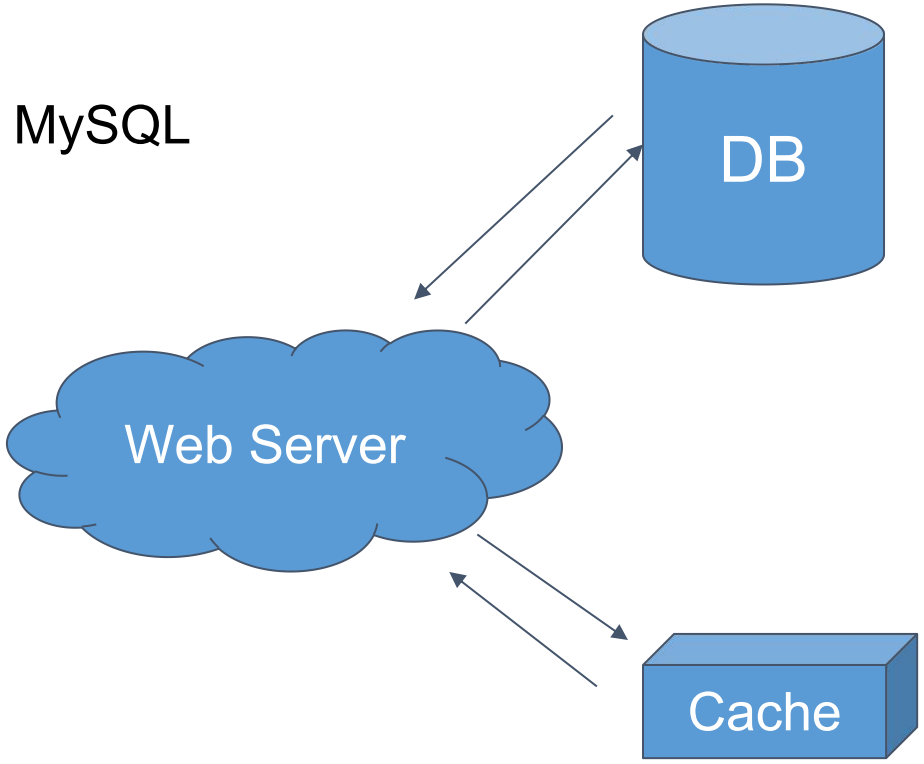
如果写很多，甚至写多读少，那么此时 cache 是没有任何优化效果的

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

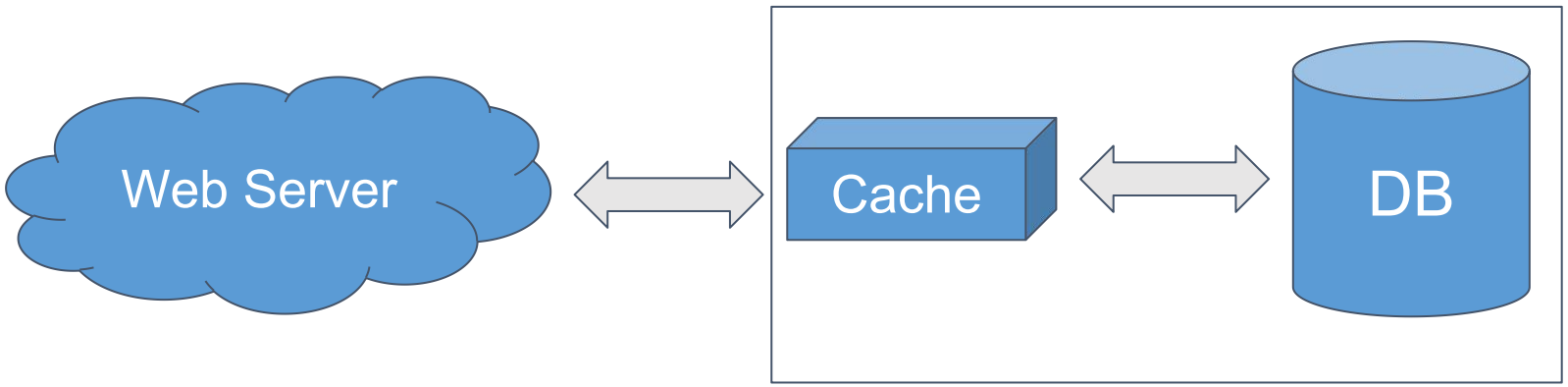
服务器只和 Cache 沟通

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）

缺点：Redis 支持单纯的 key-value 存储结构，无法适应复杂的应用场景

所以通常业界使用 Cache Aside 的方式较多，Cache 和 DB 都可以自由的搭配组合



Authentication Service

登录系统

Session

Cookie

Session 会话

- 用户 Login 以后，为他创建一个 session 对象
- 并把 session_key 返回给浏览器，让浏览器存储起来
- 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
- 此时服务器拿到 cookie 中的 session_key，在 Session Table 中检测是否存在，是否过期
- Cookie：HTTP 协议中浏览器和服务器的沟通机制，服务器把一些用于标记用户身份的信息，传递给浏览器，浏览器每次访问任何网页链接的时候，都会在 HTTP 请求中带上所有的该网站相关的 Cookie 信息。Cookie 可以理解为一个 Client 端的 hash table。

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 三问

1. Session 记录过期以后，服务器会主动删除么？
2. 只支持在一台机器登陆和在多台机器同时登陆的区别是什么？
3. Session 适合存在什么数据存储系统中

Friendship Service

好友关系的存储与查询

单向好友关系

双向好友关系

单向好友关系

- 例子：Twitter、Instagram、微博
- **存在 SQL 数据库时**
  - 查询x所有的关注对象：
  - select * from friendship where from_user_id=x
  - 查询x所有的粉丝：
  - select * from friendship where to_user_id=x
- **存在 NoSQL 数据库时**
  - 见拓展练习1

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

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

双向好友关系

- 例子：微信，Facebook，WhatsApp
- **方案1：存储为一条数据**
  - select * from friendship
  - where smaller_user_id = x or bigger_user_id=x
  - 问：为什么需要区分 smaller / bigger？
  - SQL 可以按照这种方案
  - NoSQL 很多不支持 Multi-index 不能使用这种方案
- **方案2：存储为两条数据**
  - select * from friendship where from_user_id=x
  - NoSQL 和 SQL 都可以按照这种方案
- 问：两种方案哪种更好？

方案1	
smaller_user_id	bigger_user_id
1	2
1	3
2	3

方案2	
from_user_id	to_user_id
1	2
2	1
1	3
3	1
2	3
3	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
- 第二层：column_key
- 第三层：value

- Cassandra 的 Key = row_key + column_key
- 同一个 row_key + column_key 只对应一个 value

结构化信息如何存储？

- 将其他需要同时存储的数据，序列化（Serialize）到 value 里进行存储
  - 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

无法进行 Range Query

常用：user_id

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)

可以是复合值，如 timestamp + user_id

SQL vs NoSQL

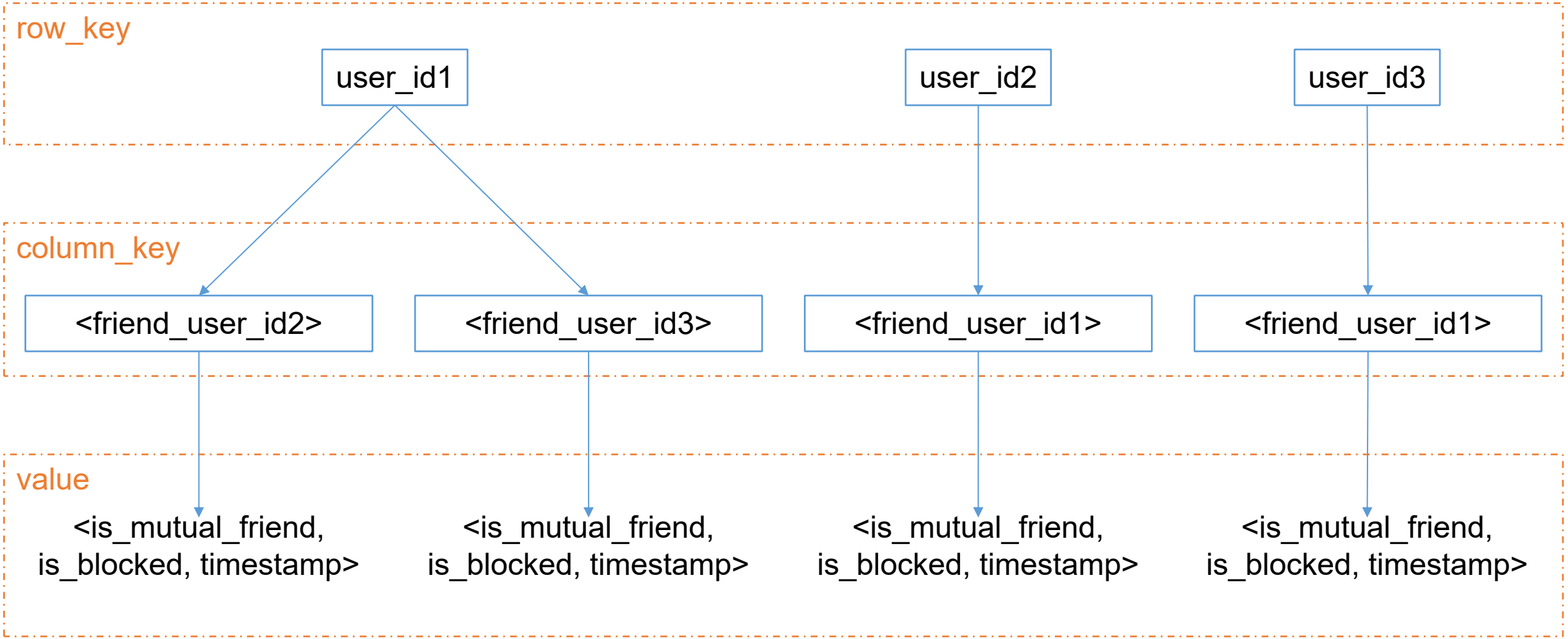
- SQL的column是在Schema中预先指定好的，不能随意添加
- 一条数据一般以 row 为单位（取出整个row作为一条数据）

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)

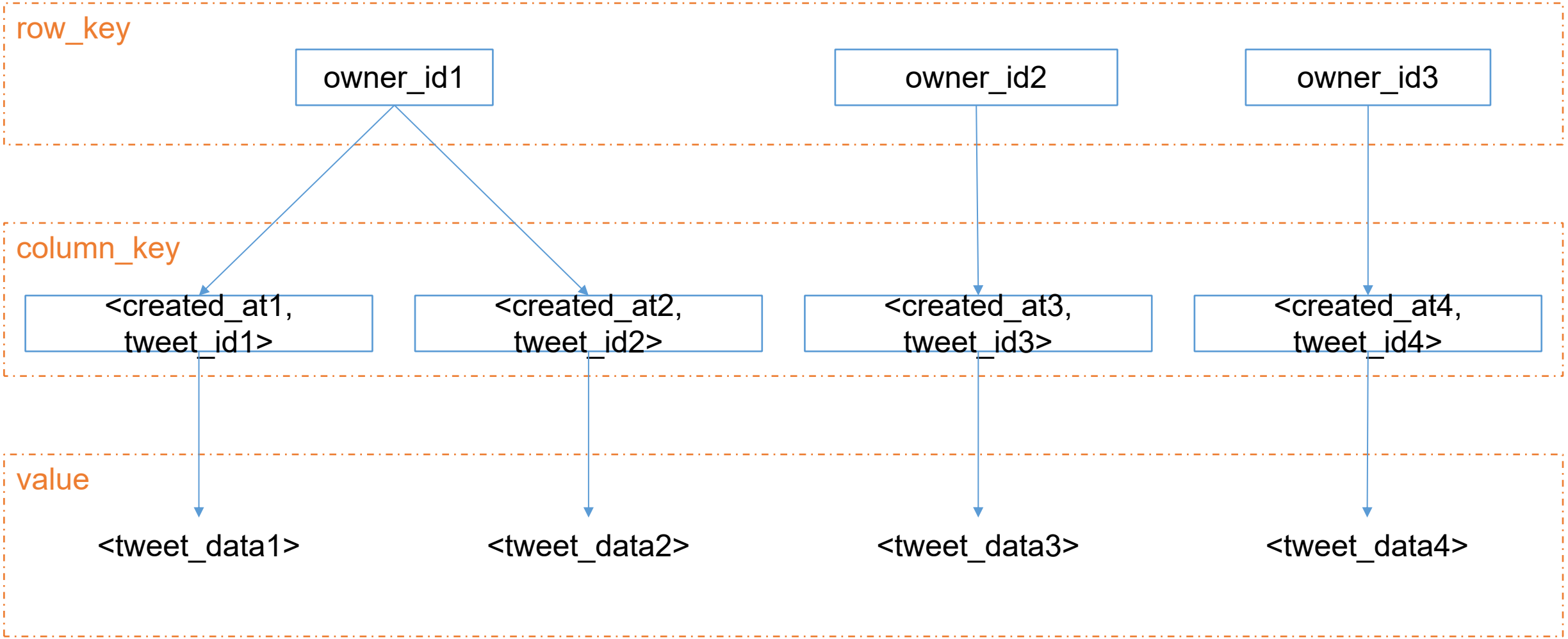
NoSQL	column_key1	column_key2	column_key3	column_key4	…
row_key1	value0	value1	...		
row_key2	...	...			
...					

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



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

你想在什么地方偷懒很大程度决定了选什么数据库

SQL：结构化数据，自由创建索引

NoSQL：分布式，Auto-scale，Replica

数据库选择原则4

一般一个网站会同时用多种数据库系统

不同的表单放在不同的数据库里

User Table 存在哪儿？

大部分公司选择了 SQL

原因：信任度，Multi-Index

Friendship 存在哪儿？

大部分公司选择了 NoSQL

原因：数据结构简单，都是 key-value 的查询/存储需求

NoSQL效率更高

拓展练习1：NoSQL 存单向好友关系

使用 Cassandra 存储单向好友关系，支持如下操作：

1. 查询某个人的关注列表
2. 查询某个人的粉丝列表
3. 查询 A 是否关注了 B

请设计出需要哪些表单和对应的表单结构

拓展练习2：NoSQL 存储 User

如果使用不支持 Multi-index 的 NoSQL 存储 User

如何同时支持按照 email, username, phone, id 来检索用户？

拓展练习3：共同好友

共同好友（Mutual Friends）是社交网站上常见的功能

请设计这个功能：列出 A 和 B 之间的共同好友

拓展练习4：LinkedIn 六度关系

LinkedIn 上有一个功能是显示你和某人之间的几度关系

（通过多少个朋友能认识）

请设计这个功能

问：可以使用宽度优先搜索（BFS）算法么？

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]
- Coach Base Architecture
  - http://horicky.blogspot.in/2012/07/couchbase-architecture.html
- Least Frequently Used Cache (LFU)
  - http://dhruvbird.com/lfu.pdf

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/