去IOE关键技术分析

演讲人：吕海波（VAGE）

内容介绍

- 互联网企业去O关键技术分析： Sharding
- 去，还是不去！
- O的替补

互联网企业去O关键技术分析：Shanding

Sharding，是Sharde-nothing的缩写，有些地方也就horizontal partitioning/horizontal split，也就是数据库切片。简单点说，就是将一张大表，拆分到N个数据库中。咱们后面，用一个更形象的词称它：拆表。

对并发比较高的数据库进行去O时，一定要考虑拆表。哪么，问题来了：

为什么要拆表

非O数据库目前的并发机制，不如Oracle更加强大。这个短板，通过拆表来弥补最为方便。

（注：本文中非O数据库主要以MySQL为例，不代表所有非O数据库）

决定并发的因素：锁。锁粒度的大小等锁机制决定了并发程度。

去0关键技术 -- 拆表 ：架构师读书问题

（有一个著名的讨论线程同步的问题，叫哲学家就餐问题，我也东施效颦，来个架构师读书问题）：

假设5个架构师要读同一本纸制书：

书只有一本，人有5名，为了避免打架，方法就是将书装进一个盒子中锁上，有一把钥匙可以打开这个盒子得到书，这把钥匙我们称为Book Key（简称BK）。

5名架构师谁先得到BK，就能打开盒子，得到书，然后开始读书。而其他人就只能等着。这种方式资源使用率有限，并发不高。

去0关键技术 -- 拆表 ：架构师读书问题

如何提高并发呢，“细化锁的粒度”。

比如5名架构师分别是A、B、C、D和E。这本《Oracle内核技术揭密》共有7章，假设5名架构师分别想看不同的章，A想看第4章，共享池原理。B想看第3章，Buffer Cache原理，C想看第6章，Redo原理，等等。

那么问题就很简单了，将书拆开，按章节拆成多本。一共7章，就拆成7本，分别装在7个盒子中，每个盒子都有一把钥匙，可以打开盒子，得到书的某一章。这样就有了7个盒子、7把钥匙。这7个钥匙分别编号为BK1，2，3.....7。

A想看第4章，他只需要得到BK4，打开锁得到盒子中的第4章，就可以了。A可以拿着第4章自己看个够，不影响B同一时间看第3章，C在同一时间看第5章等等。这样，A、B、C、D和E 5名架构师可以同时读书了。并发得到提高，系统资源利用的更加充分。

去0关键技术 -- 拆表 ：架构师读书问题

细化锁的粒度可以有效的提高并发，但这并不是一件容易的工作。

比如，下面这个问题，如果有多名架构师要读同一章怎么办？如何进一步提升“架构师读书问题”的并发？

这个问题很简单，将每一章再拆成页，A看第1页，B看第5页，等等。

去0关键技术 -- 拆表 ：架构师读书问题

但这个问题只是看似简单。《Oracle内核技术揭密》共有400页，将书拆成页后，每一页都要有一个锁保护，锁的数量就是400。

看到没，锁从1个，拆成章后变成6个，拆成页后变成400个，越来越多。锁并不是凭空而来的东西，它要占内存，处理它要耗CPU资源。

将书拆成页后，现在你看书的步骤是这样的：
查找第一页的锁
判断是否可以得到它
如果可以得到，持有它
开始阅读第一页
释放第一页的锁，查看有无其他人在等待，有的话通知他可以持有了

查找第二页的锁
判断是否可以得到它
如果可以得到，持有它
开始阅读第二页
释放第二页的锁，查看有无其他人在等待，有的话通知他可以持有了

.....

去0关键技术 -- 拆表 ：架构师读书问题

这就是锁太细、太多的问题。我们今天不讨论锁的设计，只是说明一个问题，并发的提高，不是想提你就提。

回过头说拆表的问题。
Oracle在各种锁机制（或者说并发机制）上，还是下足了工夫的，比很多其他数据库要好。比如说，MySQL。

MySQL在将重做日志数据写到Redo log buffer（重做日志缓存）中时，一直要持有log_sys->mutex。

Oracle为了提高并发，首先重做日志缓存（在Oracle中叫Log Buffer）可以有多个（SGA总大小在百G以上时，Oracle默认会有6个以上的Lob Buffer Pool）。而且Oracle中，相关日Log Buffer的锁有三个：redo allocation latch，redo copy latch和redo writing latch。

去0关键技术 -- 拆表 ：架构师读书问题

Oracle的技术细节我们不讲了，仍以“读书问题”为例子简单说下，每个人要读书时，先要拿到独占锁：redo allocationg latch，然后声明自己要读的范围，比如说A说我要读1到10页，B说要读11页到18页，等等。这个操作虽然是独占的，但只声明范围，这是一个很快的操作。

得到范围后，每个人再持有redo copy latch锁，开始同时阅读同一本书不同的页数。

除了log_sys->mutex外，MySQL在开始事务时，要增加trx_sys结构中的max_trx_id。trx_sys是一个全局结构，它的改变要在kernel_mutex的保护下完成。
Oracle中事务开始、结构过程中，都不必持有任何唯一的锁。而且，MySQL中Select也是事务，也就是说Select也要在kernel_mutex保护下，改变全局结构trx_sys.max_trx_id。Oracle的select，只在硬解析时才会有全局、独占的锁，Oracle的Softer Parse（软软解析）时，Select没有任何全局级的锁。

Oracle也有一个全局的数据结构：SCN。它也需要频繁的做自增操作，它是一个48位整数。Oracle将它分为16位的主、和32位的辅两部分。通过自增的是32位部分，这一部分的自增是通过Lock Free方式进行。很多时候同一DML会有同样的SCN，这时Oracle会再增加一个叫SUB SCN的16位整数，加以区分。SUB SCN同样以Lock Free方式自增。

去0关键技术 -- 拆表 ：架构师读书问题

很多人都问过我一个问题，MySQL可以支持多大的表，有Oracle的大吗？

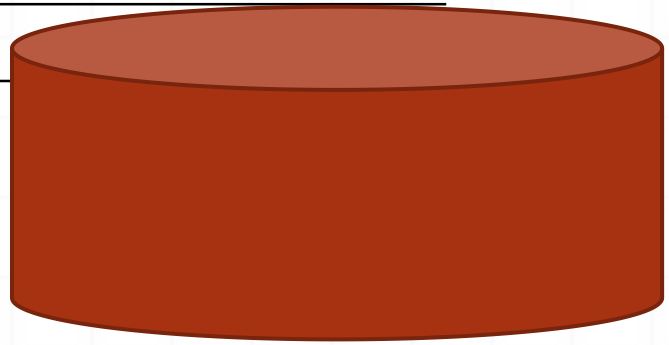
我想，从容量上来说，这些数据库都差不多。但是从并发的角度上看，最好不要让MySQL有太大的表。大表就要拆分到多个库中。当然，Oracle表大到一定程度，也可以用拆分的方法提高性能。

去0关键技术 -- 拆表的一般方法

- HASH
- RANG
- 路由库

去0关键技术 -- 拆表的一般方法：HASH

User_id	User_name	Balance	.....
1	甲	10000	.....
2	乙	10000	.....
.....	.....	.....	.....
100	丙	10000	.....
.....	.....	.....	.....
1000	丁		
.....	.....		

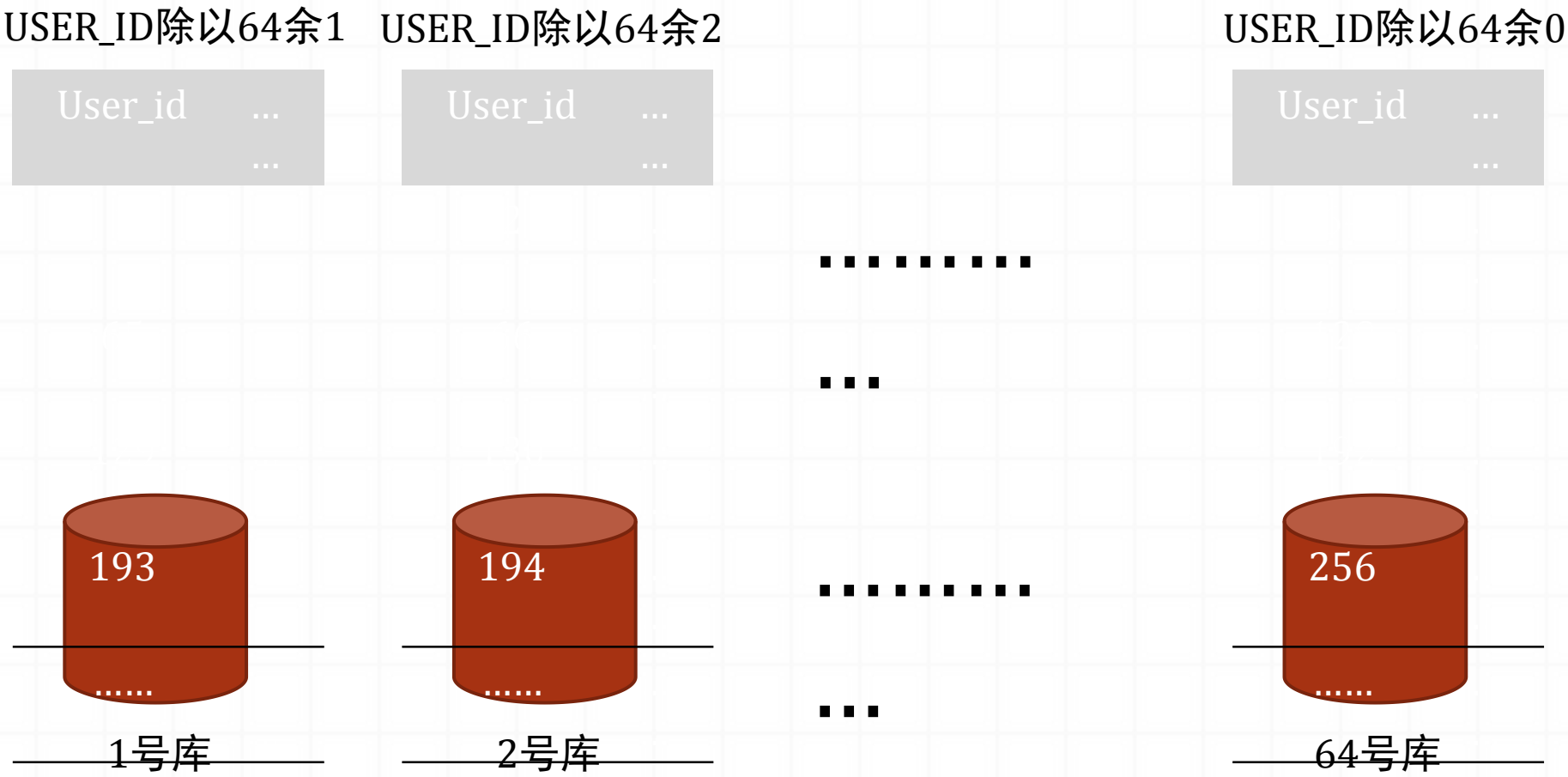


假设表被拆分为64份，分别分布到64个库中。

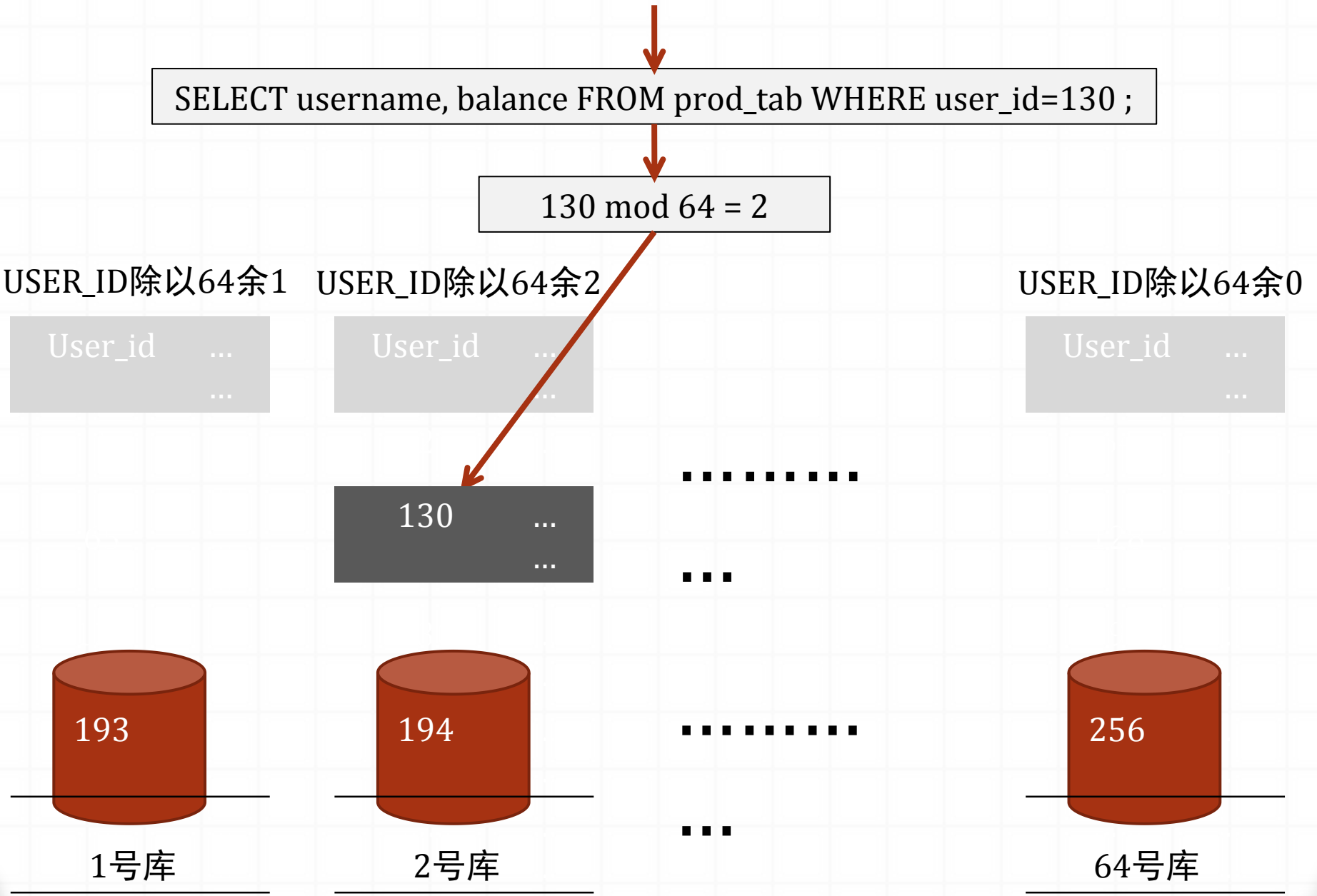
最简单的HASH方法就是，以user_id 除以 64 的余数为HASH值，决定某个用户的数据保存到几号库中。

去0关键技术 - 拆表的一般方法：HASH

以user_id 除以 64 的余数为HASH值，决定某个用户的数据保存到几号库中。



去0关键技术 -- 拆表的一般方法：HASH



去O关键技术 -- 拆表的一般方法：RANG

按USER_ID的范围拆分：

USER_ID在1到10万

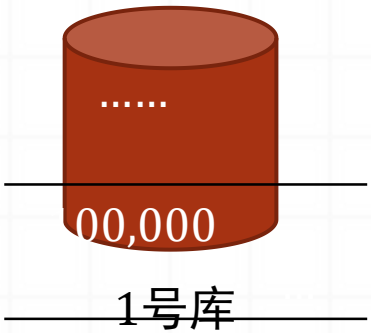
User_id ...
...

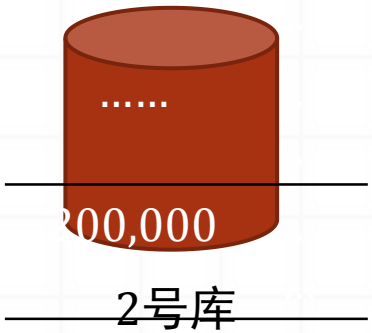
USER_ID在10万零1到20万

User_id ...
...

.....

...





.....

...

去0关键技术 -- 拆表的一般方法：路由表

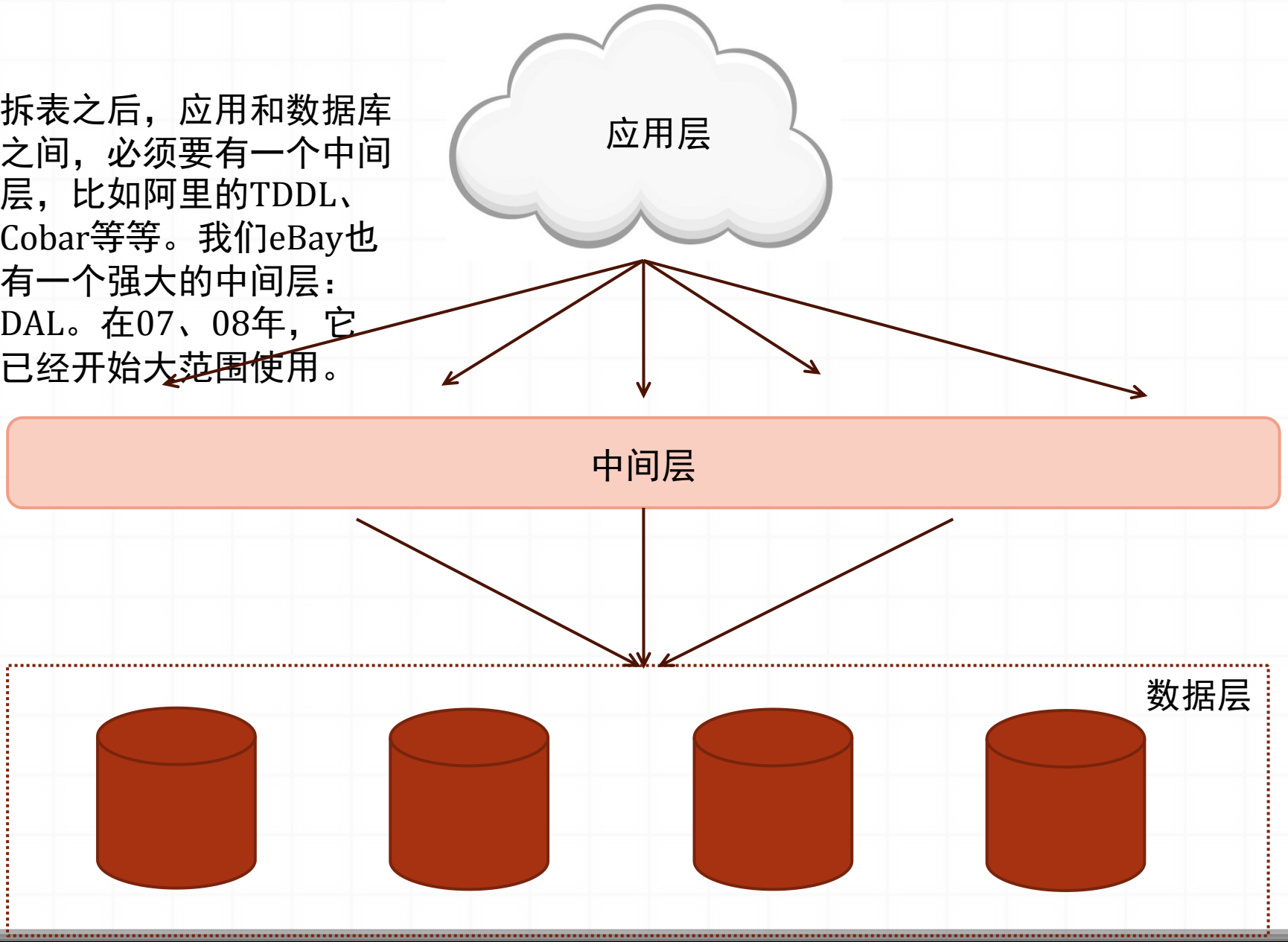
将USER_ID的分布法则写到一张表中，每次访问数据时，比如要查询USER_ID为130的行，先查询路由表，得到130存在哪个库中，再去到指定库中查询

去0关键技术 -- 拆表的一般方法：路由表

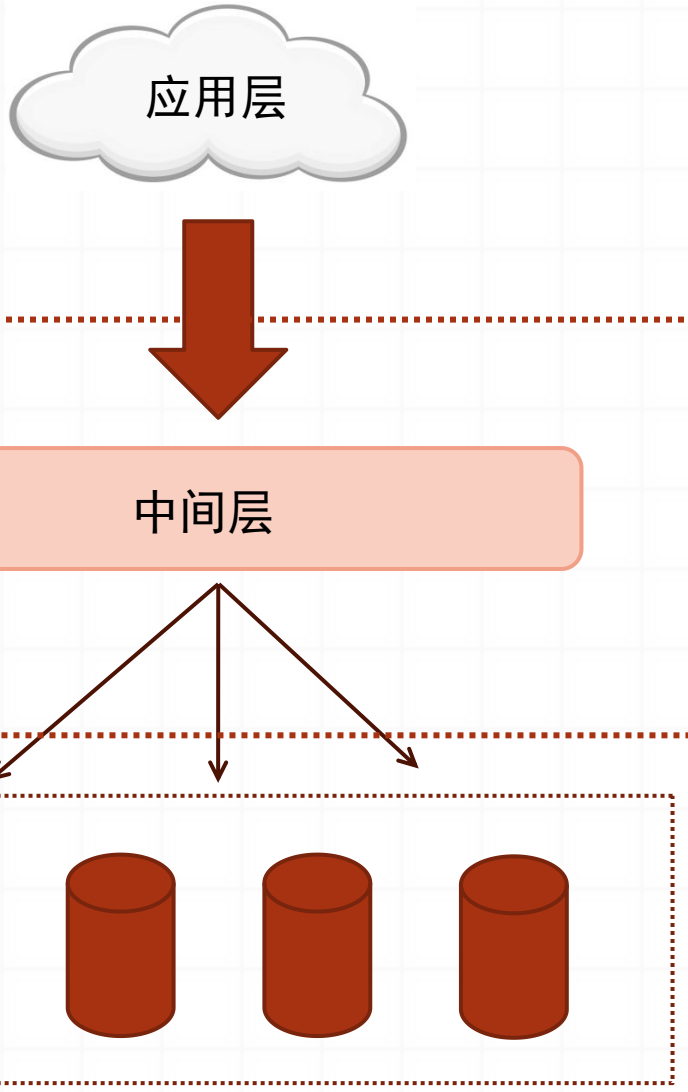
推荐阅读：
http://www.dbafan.com/blog/?p=212

去0关键技术 -- 拆表的一般方法：中间层

拆表之后，应用和数据库之间，必须要有一个中间层，比如阿里的TDDL、Cobar等等。我们eBay也有一个强大的中间层：DAL。在07、08年，它已经开始大范围使用。



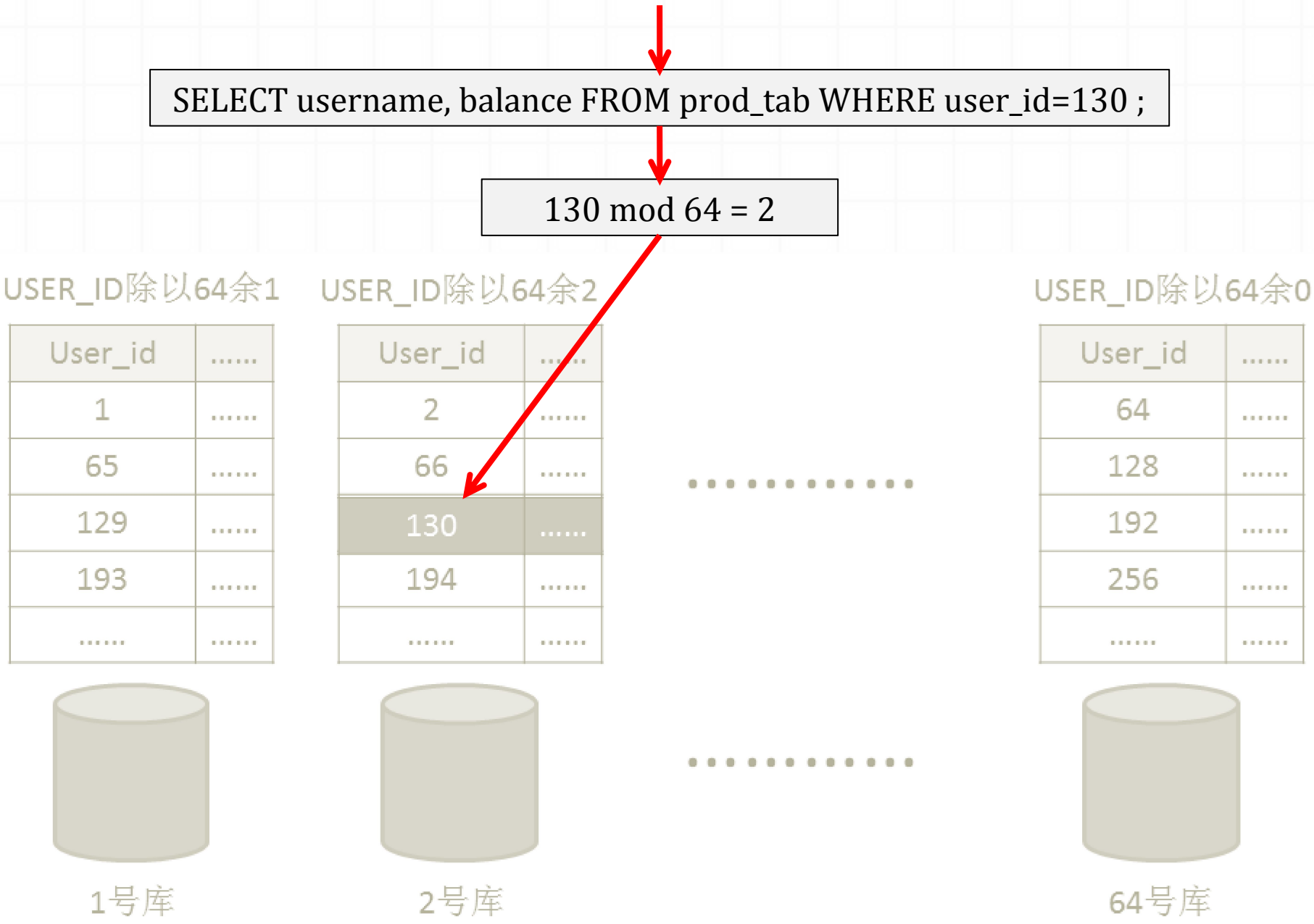
去0关键技术 -- 拆表的一般方法：中间层



应用连接中间层。在应用眼中，它认为中间层就是数据库。它不知道数据库被拆分成多个。

中间层根据ID列，将应用传递过来的SQL发送到对应数据库。

去0关键技术 -- 拆表的一般方法：中间层



去0关键技术 -- 拆表的一般方法：中间层

调用中间层提供的接口函数，连接数据库;

SELECT name, balance FROM prod_tab WHERE user_id=130 ;

根据user_id=130判断出用户要查询的数据在2号数据库中;

连接2号数据库;

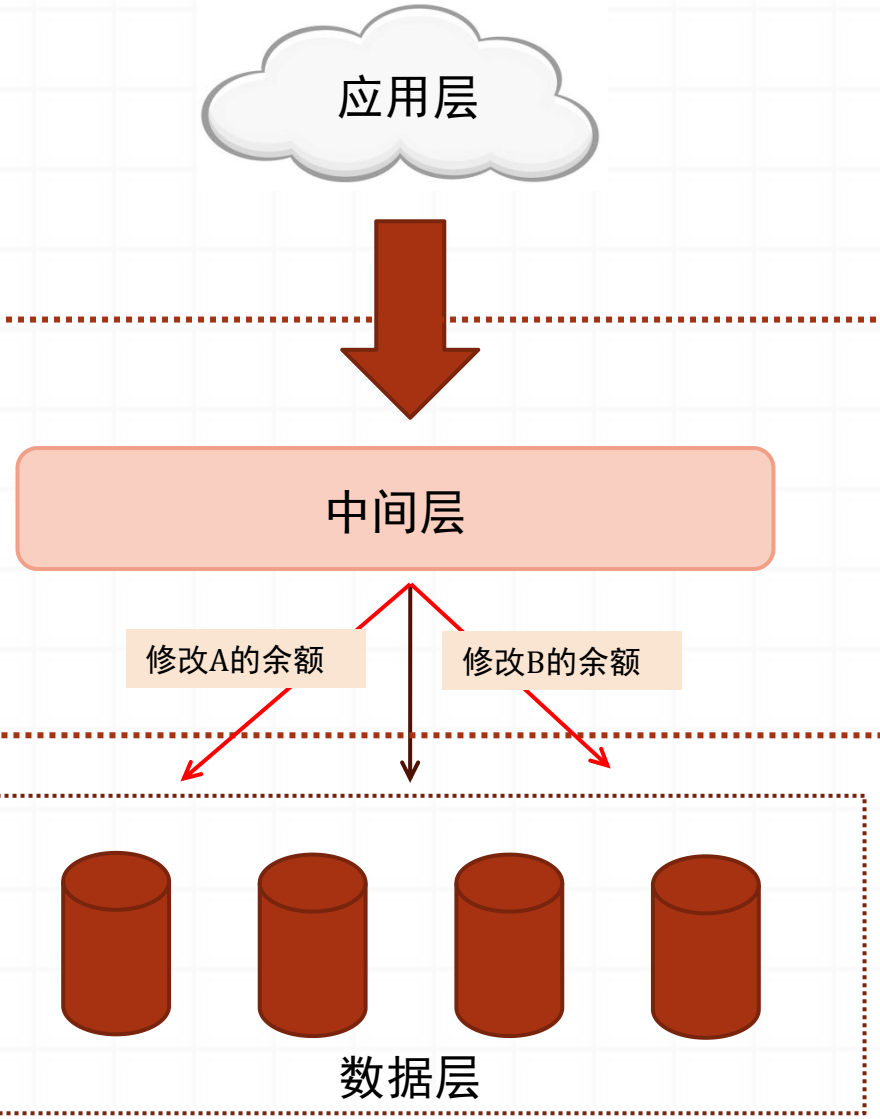
SELECT name, balance FROM prod_tab WHERE user_id=130 ;

2号数据库开始执行查询

去0关键技术 -- 拆表的一般方法：中间层

中间层在处理查询、和包含一条DML语句的事务时，逻辑都是简单的，处理多条DML的事务是复杂的。

去0关键技术 -- 拆表的一般方法：中间层



连接数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;

连接1号数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
结束事务 Commit;
连接4号数据库;
开始事务;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;

两个数据库分别开始自己的事务

去0关键技术 -- 拆表的一般方法：中间层

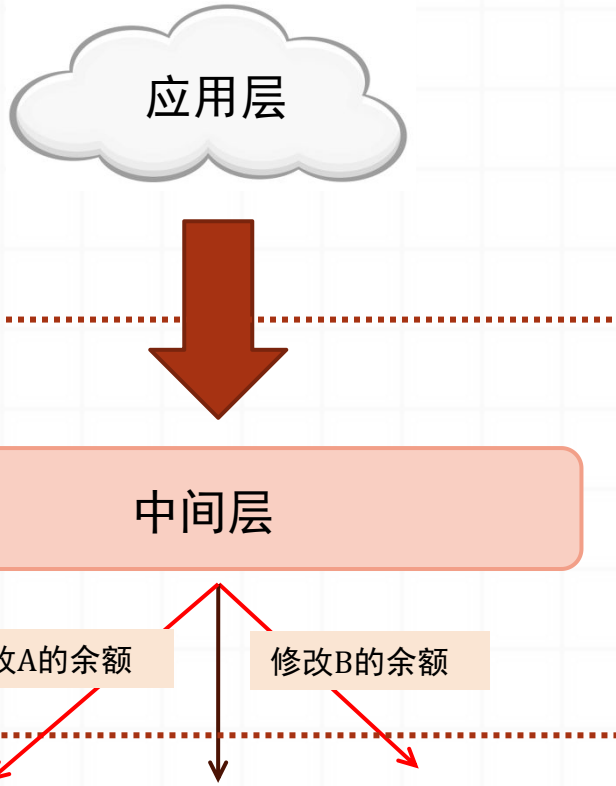
连接数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;

连接1号数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
结束事务 Commit;

连接4号数据库;
开始事务;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;

两个数据库分别开始自己的事务

应用层的一个事务，由于涉及两个数据库，因此在中间层被分成了两个事务。为了保证两个事务的原子性等ACID特性。只能使用两阶段提交或三阶段提交，但因网络造成的时延，这是高并发性系所不能容忍的。这时候就只能选择“最终一致性”模型。

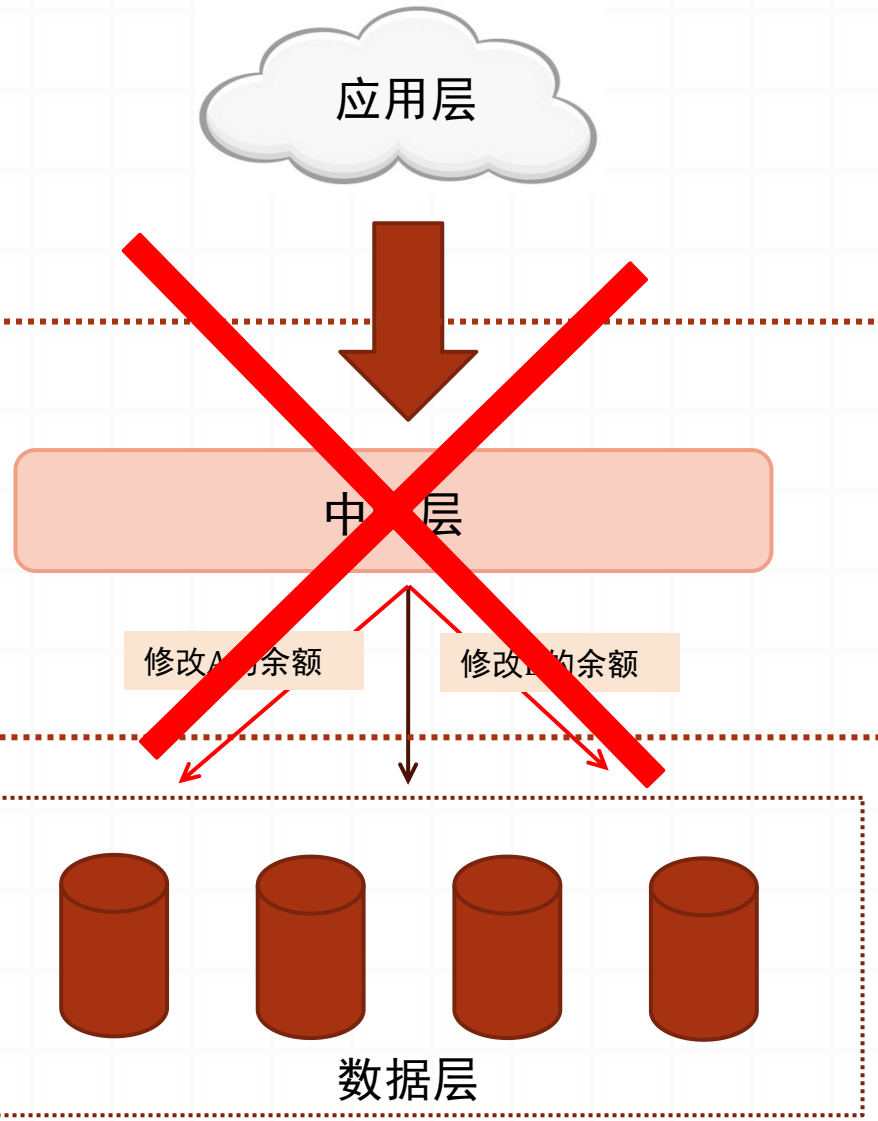


连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完
Message_queue：取出消息
Update 某帐户
结束事务 Commit;
Message_queue：取出消息
Update 某帐户
结束事务 Commit;

连接1号数据库;
开始事务;
……

CAP理论：鱼和熊掌不能兼得。研究人员发现，不能在同一顿饭中，即吃熊掌又吃鱼。

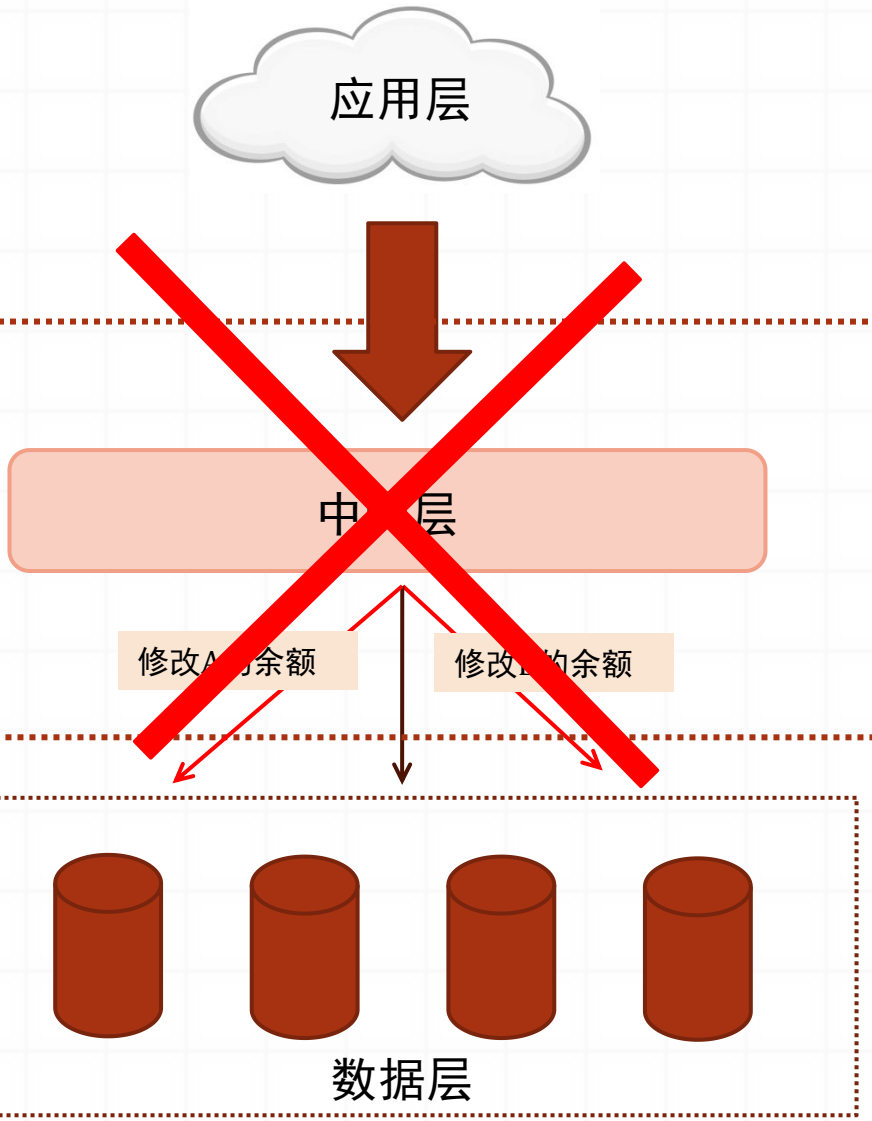
最终一致性：午餐熊掌，晚饭鱼。



连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完
Message_queue：取出消息
Update 某帐户
结束事务 Commit;
Message_queue：取出消息
Update 某帐户
结束事务 Commit;

连接1号数据库;
开始事务;
……

两个数据库分别开始自己的事务



连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完
Message_queue：取出消息
Update 某帐户
结束事务 Commit;

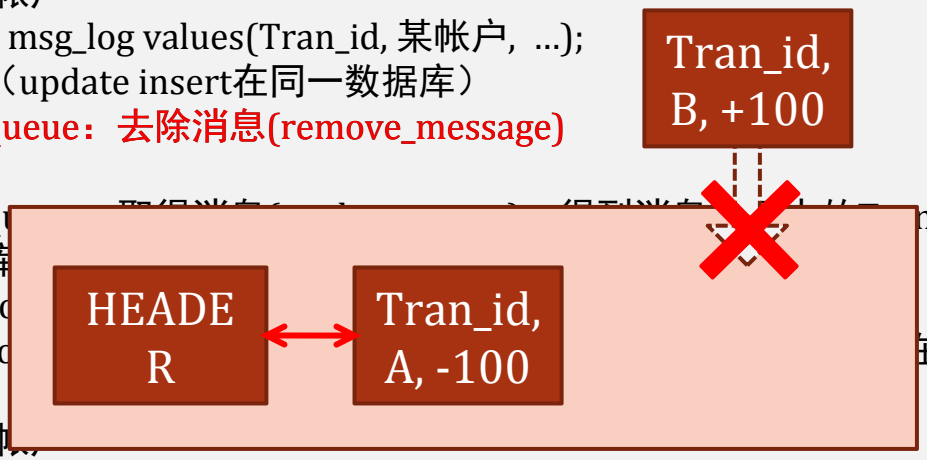
Message_queue：取出消息
Update 某帐户
结束事务 Commit;

连接1号数据库;
开始事务;
……

两个数据库分别开始自己的事务

连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完

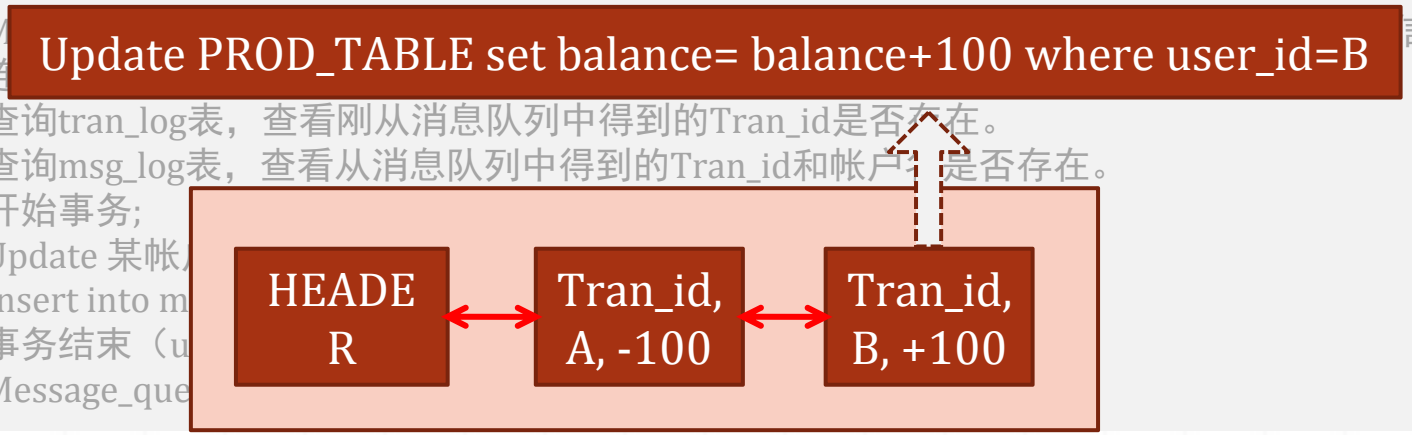
Message_queue：取得消息(peek_message)，得到消息队列中的Tran_id、帐户名和其他信息。
连接数据库;
查询tran_log表，查看刚从消息队列中得到的Tran_id和帐户名是否存在。
查询msg_log表，查看从消息队列中得到的Tran_id和帐户名是否存在。
开始事务;
Update 某帐户
Insert into msg_log values(Tran_id, 某帐户，...);
事务结束（update insert在同一数据库）
Message_queue：去除消息(remove_message)



Message_qu n_id、帐户名和其他信息。
连接数据库
查询tran_lo
查询msg_lo 在。
开始事务;
Update 某帐;
Insert into msg_log values(Tran_id, 某帐户，...);
事务结束（update insert在同一数据库）
Message_queue：去除消息(remove_message)

连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完

Message_queue：取得消息(peek_message)，得到消息队列中的Tran_id、帐户名和其他信息。
连接数据库;
查询tran_log表，查看刚从消息队列中得到的Tran_id和帐户名是否存在。
查询msg_log表，查看从消息队列中得到的Tran_id和帐户名是否存在。
开始事务;
Update 某帐户
Insert into msg_log values(Tran_id, 某帐户，...);
事务结束（update insert在同一数据库）
Message_queue：去除消息(remove_message)



连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完

Message_queue：取得消息(peek_message)，得到消息队列中的Tran_id、帐户名和其他信息。
连接数据库;
查询tran_log表，查看刚从消息队列中得到的Tran_id和帐户名是否存在。
查询msg_log表，查看从消息队列中得到的Tran_id和帐户名是否存在。
开始事务;
Update 某帐户
Insert into msg_log values(Tran_id, 某帐户，...);
事务结束（update insert在同一数据库）
Message_queue：去除消息(remove_message)

Message_queue：取得消息(peek_message)，得到消息队列中的Tran_id、帐户名和其他信息。
连接数据库;
查询tran_log表，查看刚从消息队列中得到的Tran_id是否存在。
查询msg_log表，查看从消息队列中得到的Tran_id和帐户名是否存在。
开始事务;
Update 某帐户
Insert into msg_log values(Tran_id, 某帐户，...);
事务结束（update insert在同一数据库）
Message_queue：去除消息(remove_message)

去O关键技术 -- 拆表的一般方法：中间层

当然，这种拆分不一定都用于去O。如果不想去O的话，随着单库规模越来越接近极限，也可以使用同样的架构分拆Oracle。

去0关键技术 -- 去，还是不去！

去，还是不去。这个问题没有TO BE OR NOT TO BE 哪么复杂，需要考虑的重要问题，也就下面这三点：

1. 成本

2. 可控性

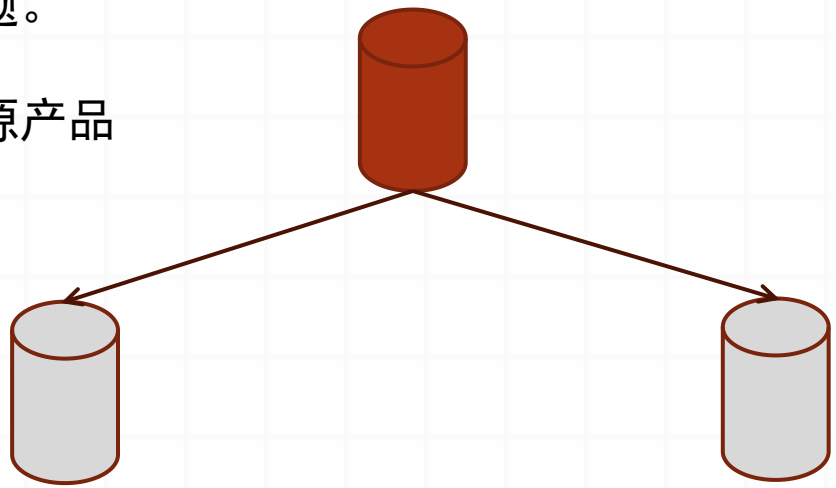
3. 高层或领导意志

去O关键技术 -- 去，还是不去：成本

数据规模越大，使用O式架构的成本将降低，和去O后成本逐渐近似，甚至有可能会低于去O架构的成本。

我们有一套700T的OLTP库，共拆分为将近30个Oracle。如果使用去O架构的话，为了数据安全，每个数据库采用一拖二模式（一主两备）。按一台主机在做完RAID后有效空间5TB算，700除5，共需要140个主机，每个主机都有两个备份（一拖二），主机实际数量是420台。
420台主机，也是一笔庞大的费用。和使用Oracle的架构，其实差不多。
另外，前面我们讲过，开源产品并发性目前还没有Oracle强，为了提高并发，单个库不要太大，假设我每个库200G，700TB容量，共需要拆分成3500个数据库。这么多数据库，元数据管理是一个很大问题。

当数据规模没有这么大时，开源产品将有效降低成本。



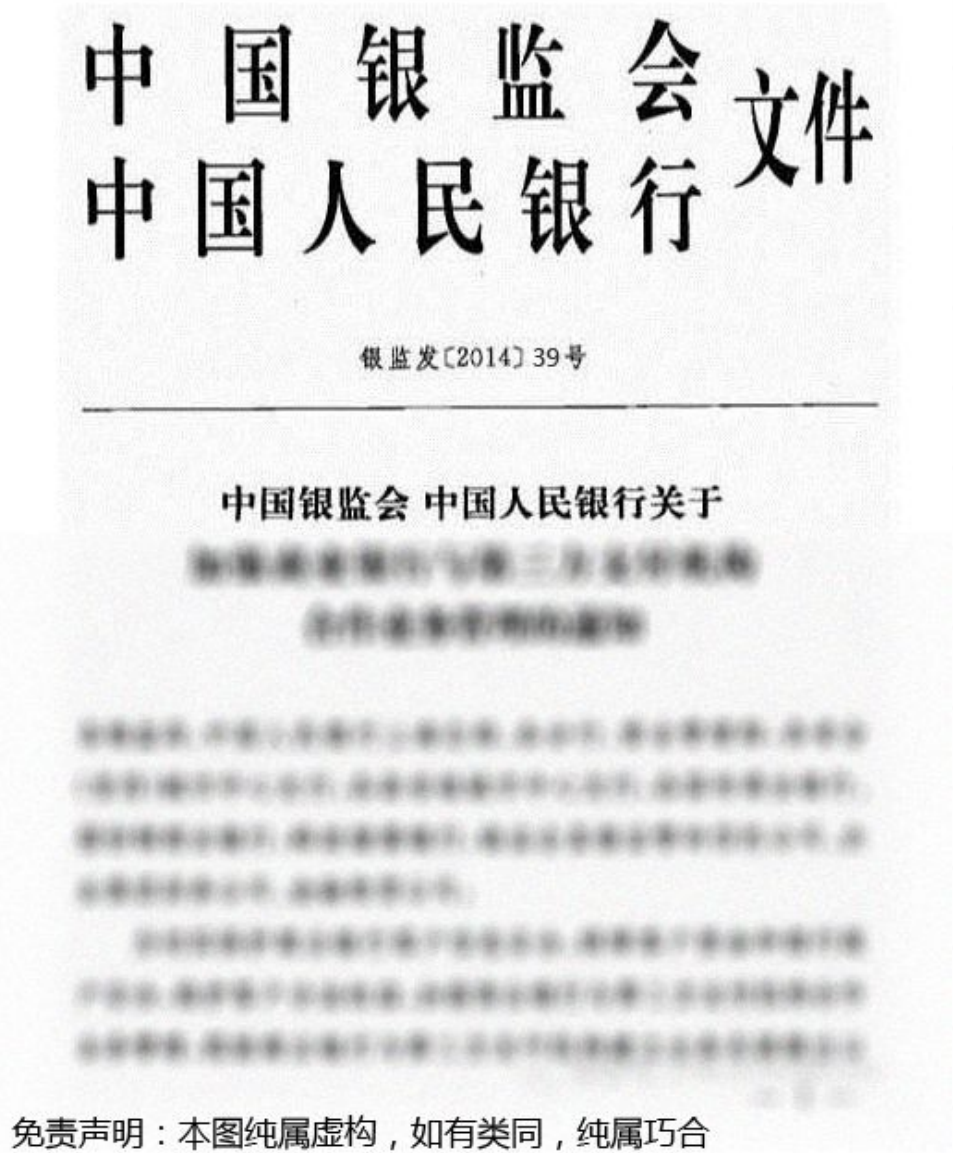
去0关键技术 -- 去，还是不去：可控性

开源产品，可控性的确要强于闭源的Oracle。但这是建立在对开源产品有深入研究基础上的。如果技术实力达不到，要注意开源产品服务远不如Oracle完善。从这个角度上说，选择开源，不单是“勇敢者的游戏”，更是“智者的游戏”。

开源的两大优点，一个是便宜、一个是有源码，如果你只抓住第一个优点“便宜”，哪你也只能是一名便宜的DBA。因此，从个人职业生涯发展角度，抓住“源码”这个方向，也更有意义。

Oracle一直被称为可控性不高的数据库，各种各样的原理，像雾里看花、水中望月一样，只能猜测，因为没有源码，而无法一窥真像。其实Oracle也有“源码”的，就是汇编。想了解什么原理，反汇编一下，一切也都清清楚楚。

去0关键技术 -- 去，还是不去：高层或领导意志



中国银监会
中国人民银行 文件

银监发〔2014〕39号

中国银监会 中国人民银行关于

[redacted]

免责声明：本图纯属虚构，如有类同，纯属巧合

去0关键技术 -- 去，还是不去：高层或领导意志

[摘要]IBM会对工信部和政府承认的第三方安全检测机构开放软件代码，以方便政府做信息安全合规检测。

去0关键技术 --
去，还是不去：让合适的技术，解决合适的问题

不必纠结去还是不去，一定要去0，或者一定要坚守0的阵地，都是“着相”了。

技术优劣之争，某种数据库好于另种数据库，某种编程语言好于另一种编程语言，这是技术人员在一定层次必有的论点。做为架构，要让合适的技术，解决合适的问题。当然，这并不简单，需要深入的理解各种技术。

对于OLTP型数据库：
数据量超大、并发高：Oracle+拆表
数据量小、并发高：非0数据库+拆表

去0关键技术 -- O的替补

如果决定权在你，你更熟悉哪种数据库，或者，你更喜欢哪种数据库，就用哪种数据库替补。

无论MySQL和PostGre SQL，都有他们的优缺点。尽量使用你熟悉、你可以掌控的技术。

如果决定权不在你，哪就接受现实吧。

硅谷一线IT公司，机不可失啊

eBay招聘Senior Oracle DBA,工作地点上海张江德国中心大厦

国际化团队
硅谷一线互联网公司
海量、高并发数据环境
机不可失

THANKS

吕海波（VAGE）