架构之名

真的需要分布式数据库吗

演讲人：吕海波（VAGE）

内容介绍

- 为什么需要分布式数据库
- 分布式数据库的一般方法：分散数据
- 分布式数据的关键：逻辑聚合层
- 高性能分布式事物的关键：最终一致性
- 节约成本的方法：适当分布式

为什么需要分布式数据库

是因为数据量太大吗？

对于OLAP应用来说，或许是这样的，但对OLTP来说，数据量通常不是问题。

例如全国14亿，无论男女老幼，每人申请10个支付宝帐号，也才总共140亿行记录。140亿行记录的数据量，可以算是大表，但并没有大到一定要分布式才能存储。而事实上支付宝的核心数据库是由几十节点数据库组成的一套分布式数据库。

为什么需要分布式数据库

- 硬件能力：单机处理能力有限。
- 软件瓶颈：数据库软件程序遇到竞争。

硬件能力

以整机256核的服务器来算，每个事务假设需要20ms(0.02秒)，一个核心每秒处理50个事务，256核就是256*50，整机每秒共能处理**12800**个事务。

而双11时峰值事务数达每秒数万笔事务。已经超出了单台服务器的峰值，因此，分布式是必然的解决方案。

软件瓶颈

软件瓶颈并不是指程序写的不好、不够优化。目前主流的NoSQL、关系数据库，软件代码的优化方面，都是做的非常好的。

我这里所说的“软件瓶颈”，主要指并发控制机制。

并发控制机制，就如同一个城市的交通体系。并不是一味的路宽，就能减少阻塞。比如繁忙的十字路口，路再宽，对于减少阻塞帮助不大，不如修建立体交通（立交桥），这样南来北往互相不再阻塞。

软件瓶颈

有些数据库的并发机制像这样：简单、直接、有效。

但由于南来的、北往的互相阻塞，有时候可能会造成拥堵。

多数NoSQL数据库都是这样，如Mongo DB。在进行DML时，它的锁是Database级别的

注：2015年下半年发布的3.0版已经锁级别降低。

软件瓶颈

有些数据库的并发机制稍微复杂：

比如MySQL和PostGre SQL，在DML语句执行期间，它们的锁都分成多个层次，对象级别、行级锁、内存页级别。

软件瓶颈

成熟商业数据库的并发机制更加复杂：

比如Oracle，锁的层次、锁的类型要比NoSQL和MySQL类的开源数据库丰富许多，并发性更高。

软件瓶颈：全局锁

全局锁，就是一旦某个进（线）程持有，其他进（线）程的同类操作全部被阻塞。

看一个数据库并发机制的好坏，就看它的全局锁数量和持有时间。全局锁越少、持有时间越短，并发机制就越好，并发性越高

软件瓶颈：MySQL全局锁举例

在代码层面，全局锁其实是一个全局变量，以trx_sys为例，可以在storage/innobase/include/trx0sys.h中，找到它的声明：

```
/** The transaction system */
extern trx_sys_t*   trx_sys;
```

“extern”声明它是一个全局变量。在事务开始时，线程要增加trx_sys中的max_trx_id，和设置trx_sys中的view_list。

因为trx_sys是一个全局结构变量，所有线程共享一份，因此它的改变要在它的全局锁保护下完成。保护它的全局锁就是下面红字显示的“Mutex”。

（同样是在trx0sys.h中，可以找到trx_sys_t结构的原型：）

```
603 struct trx_sys_t{
604
605     ib_mutex_t      mutex;      /*!< mutex protecting most fields in
606                     this structure except when noted
607                     otherwise */
608     ulint      n_prepared_trx; /*!< Number of transactions currently
609                     in the XA PREPARED state */
…………
```

软件瓶颈：MySQL全局锁举例

以事务开始时设置trx_sys中的view_list为例，这段示例代码在storage/innobase/read/read0read.cc中，read_view_open_now_low函数的主要作用是设置trx_sys中的view_list。

```
386     mutex_enter(&trx_sys->mutex);
387
388     view = read_view_open_now_low(cr_trx_id, heap);
389
390     mutex_exit(&trx_sys->mutex);
```

在read_view_open_now_low函数之前，mutex_enter函数设置全局锁trx_sys->mutex。这将阻塞MySQL中所有涉及事务的操作，对innodb引擎来说，基本上就是阻塞所有操作了。read_view_open_now_low函数之后的mutex_exit是释放Mutex。哪么这个函数通常会耗用多长时间呢？

软件瓶颈：MySQL全局锁举例

SYSTEM TAP测试脚本：

```
1 #!/usr/bin/stap
2
3  global mutex_time, mutex_cycle;
4
5  probe begin {
6      printf("Begin.\n");
7  }
8
9  probe process("/usr/local/mysql/bin/mysqld").statement("read_view_open_now@read0read.cc:387") {
10     mutex_time=gettimeofday_us();
11     mutex_cycle=get_cycles();
12 }
13
14 probe process("/usr/local/mysql/bin/mysqld").statement("read_view_open_now@read0read.cc:389") {
15     printf("hold time:%d, hold cycle:%d \n",
               gettimeofday_us()-mutex_time, get_cycles()-mutex_cycle);
16 }
```

在我的测试机中，以Select操作为例，read_view_open_now_low函数将耗时35微秒左右，8、9万个指令周期。

应用题：从甲地到乙地必须经过路口A。路口A每次只能通过一个人，每次穿过路口A需耗时35微秒。问：每秒最多可以有多少人从甲地到乙地。

不考虑其他路段所耗时间，28571人次。

软件瓶颈：全局锁

测试结果是在我自己的个人电脑上得到的，但即使是高端PC Server服务器，也不会比这个结果快多少。

高端服务器的优点是核多、核间通信效率高、L1等CACHE多，等等。在大数据量、高并发环境下，高端服务器优势明显，如果只是衡量单线程下一个简单的C++函数的执行时间，高端服务器比我的个人电脑也快不了多少。

而且，Select在执行时，并不是只有这一处全局锁。单说trx_sys变量，还有一次全局锁操作，就是在“trx_start_if_not_started”函数中，将trx_sys中的事务ID加1时。

另外在SQL解析期间、逻辑读期间、事务结束期间，都需要全局锁。

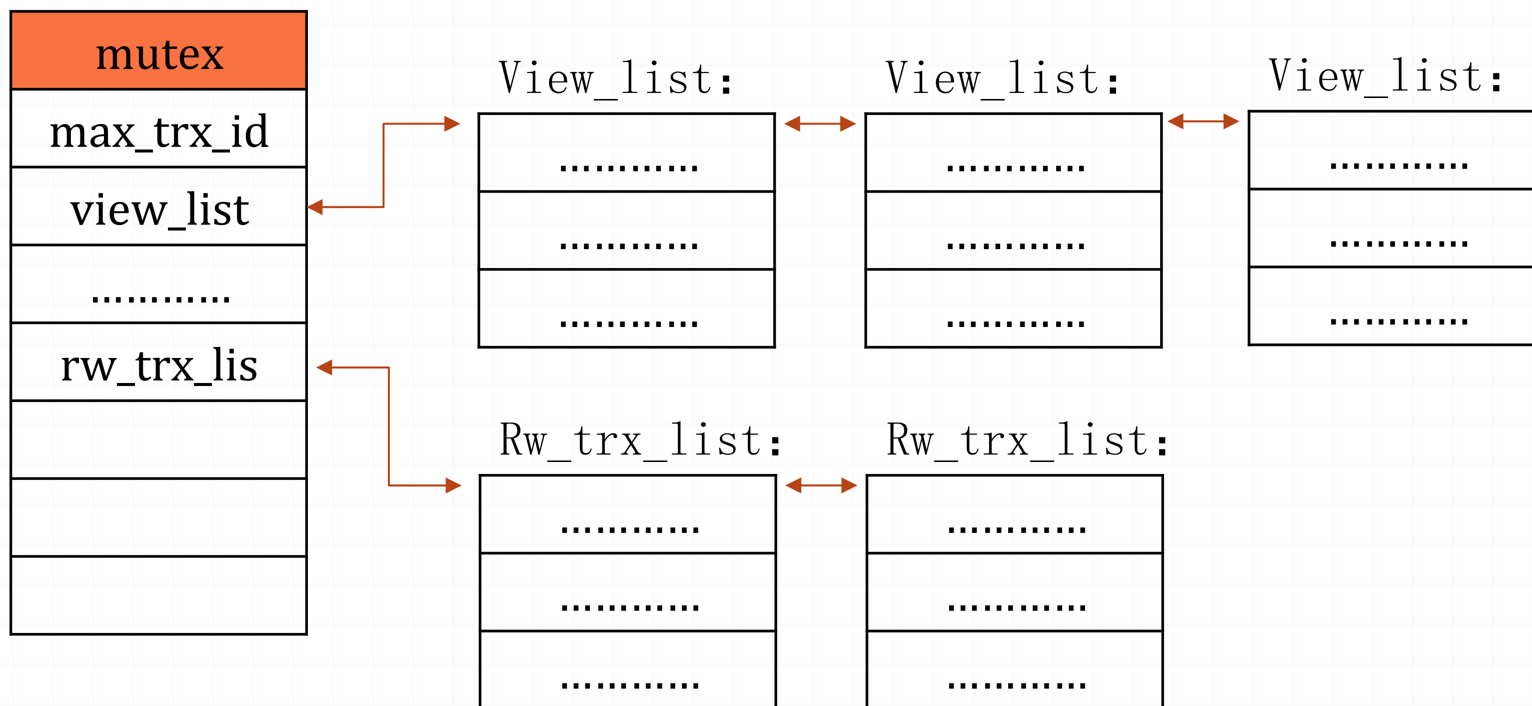
软件瓶颈：全局锁

相比之下， Oracle普通的Select和DML全局锁非常少，只在硬解析时会持有少数全局锁，软解析和软软解析时没有全局锁。

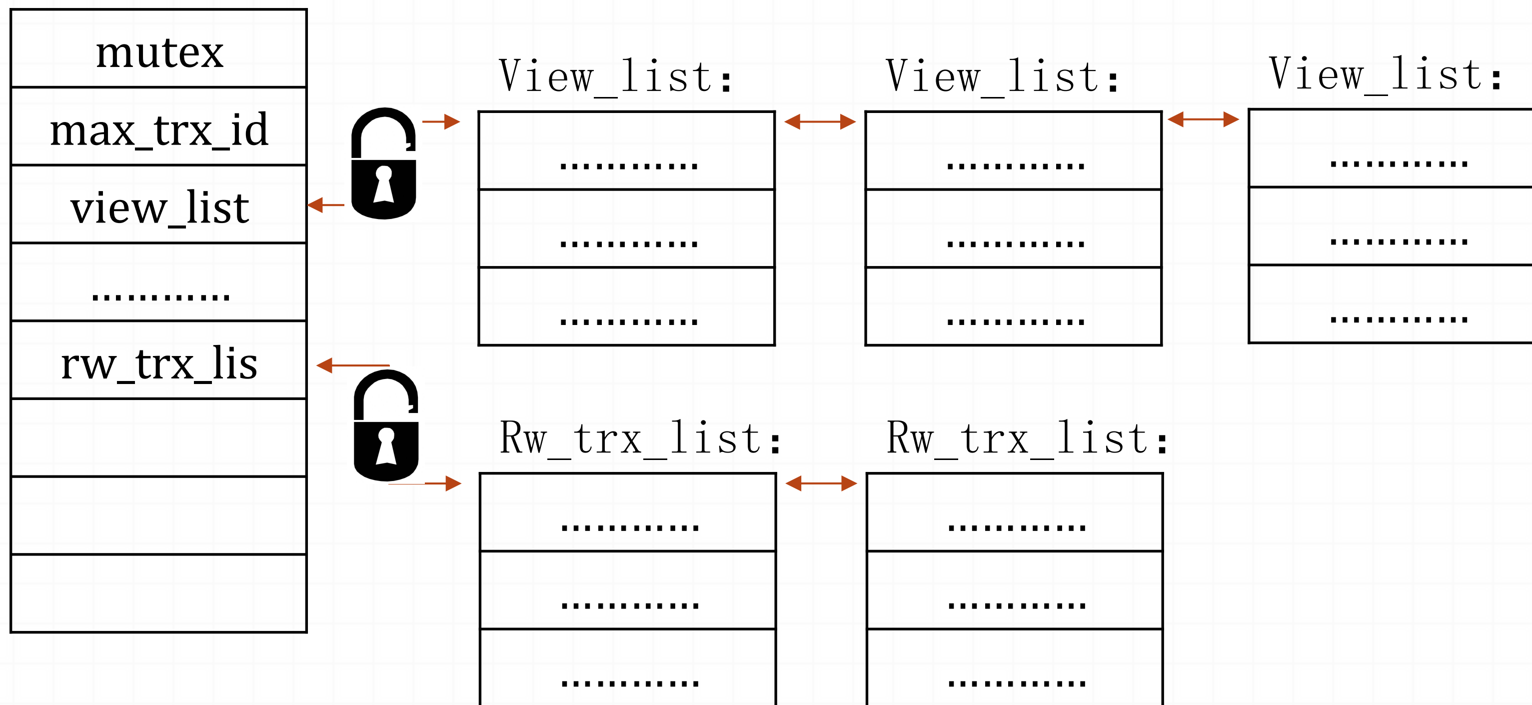
Oracle的整个事务流程中，没有一处全局锁。比如事务ID，MySQL有一个“最大事务ID”，是一个保存trx_sys中的整数。每个线程开始事务时，要在全局锁的保护下，将这个最大事务ID加1，得到的新值就是此新事务的ID。

Oracle的事务ID不是一个整数，它是根据事务所占空间计算得到的，因此“得到事务ID”的过程，不需要像MySQL哪样在全局Mutex锁的保护下进行。

软件瓶颈：全局锁



软件瓶颈：全局锁



软件瓶颈

并发机制是为了避免竞争，但无论多么复杂的并发机制，都无法避免所有的竞争。

仍以交通为例，避免竞争的最好方式，是减少车流量。如果把一个上海分成十个、一百个上海，“魔堵”情况，将大大改善。

因此，分布式是解决“软件瓶颈”的最佳选择。

分布式数据库的一般方法：分散数据

分布式数据库本质上就是将数据分散存储。简单点说，就是将一张表分散存储到多个数据库中。这种技术在数据库界有一系列专门的名词：Database Sharding、horizontal partitioning/horizontal split、拆表、分表等等。

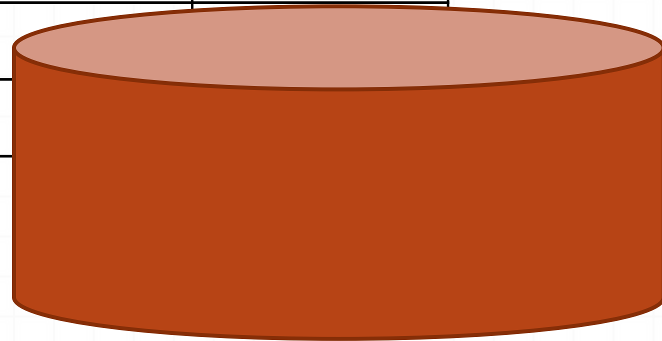
分布式数据库的一般方法：分散数据

分布式数据库的实现方法，就是根据某种规则，将数据分散存储。分散规则总结来说有以下三种：

- HASH
- RANG
- 路由库

分散数据规则：HASH

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



假设USER表被拆分为64份，分别分布到64个子库中。

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USER的数据保存到几号子库中。

分散数据规则：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。

USER_ID除以64余1

User_id	……
1	……
65	……
129	……
193	……
……	……

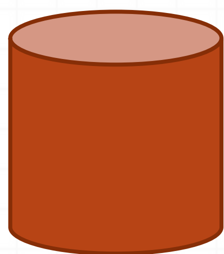
USER_ID除以64余2

User_id	……
2	……
66	……
130	……
194	……
……	……

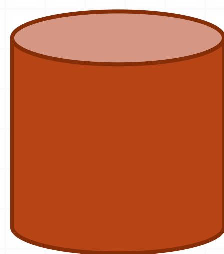
…………

USER_ID除以64余0

User_id	……
64	……
128	……
192	……
256	……
……	……

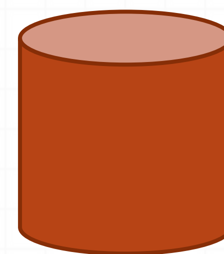


1号子库



2号子库

…………



64号子库

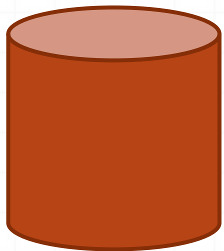
分散数据规则：HASH

SELECT username, balance FROM prod_tab WHERE user_id=130 ;

130 mod 64 = 2

USER_ID除以64余1

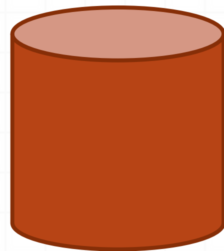
User_id	……
1	……
65	……
129	……
193	……
……	……



1号库

USER_ID除以64余2

User_id	……
2	……
66	……
130	… …
……	……

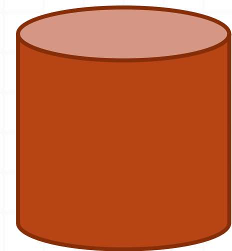


2号库

……………

USER_ID除以64余0

User_id	……
64	……
128	……
192	……
256	……
……	……



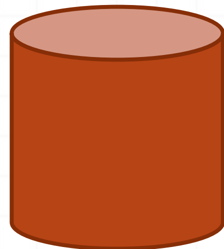
64号库

分散数据规则：RANG

按USER_ID的范围拆分：

USER_ID在1到10万

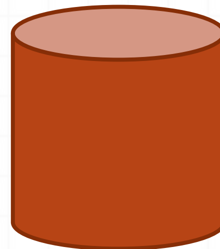
User_id	……
1	……
2	……
3	……
……	……
100,000	……



1号库

USER_ID在10万零1到20万

User_id	……
100,001	……
100,002	……
100,003	……
……	……
200,000	……



2号库

....

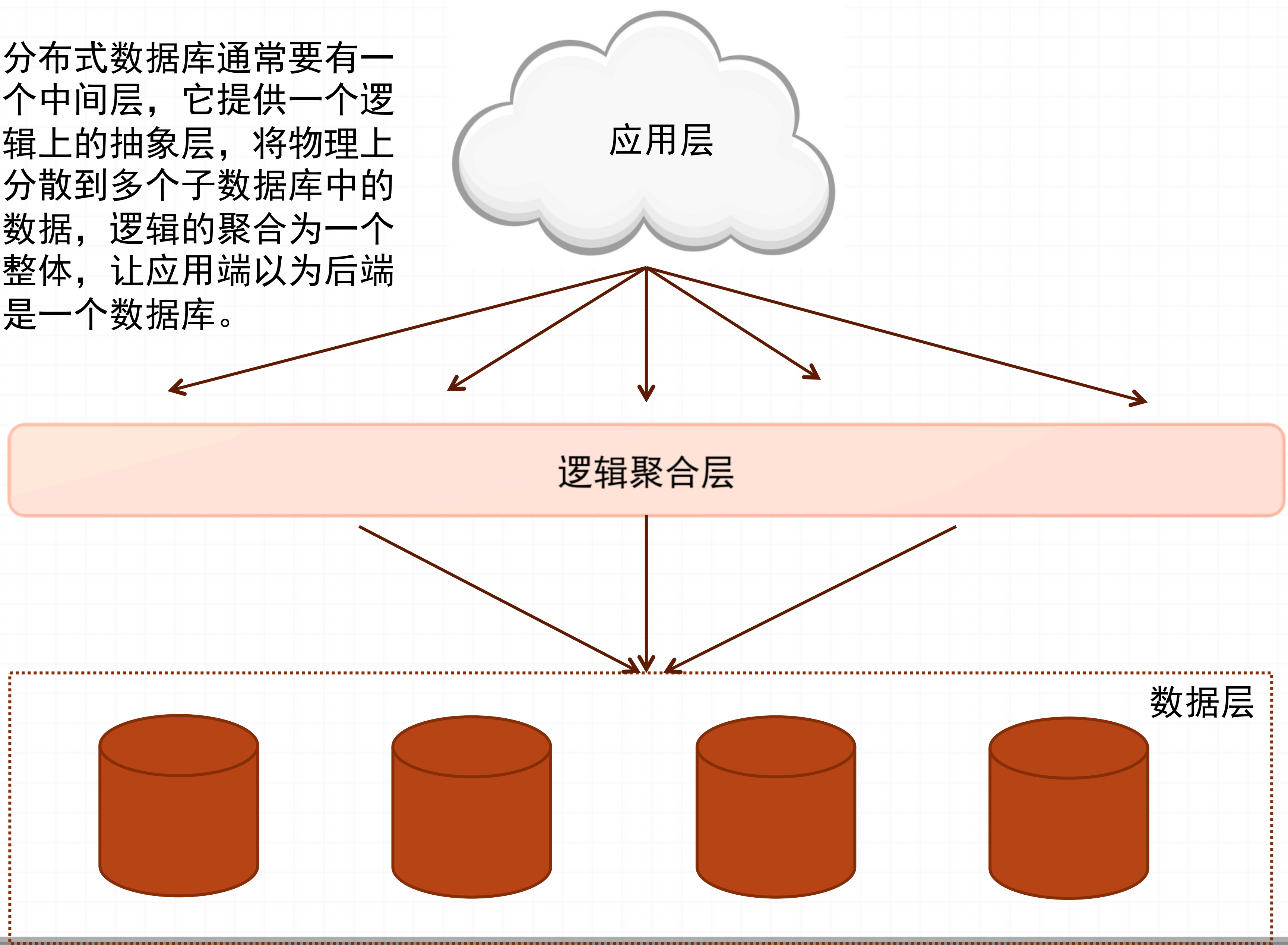
....

分散数据规则：路由

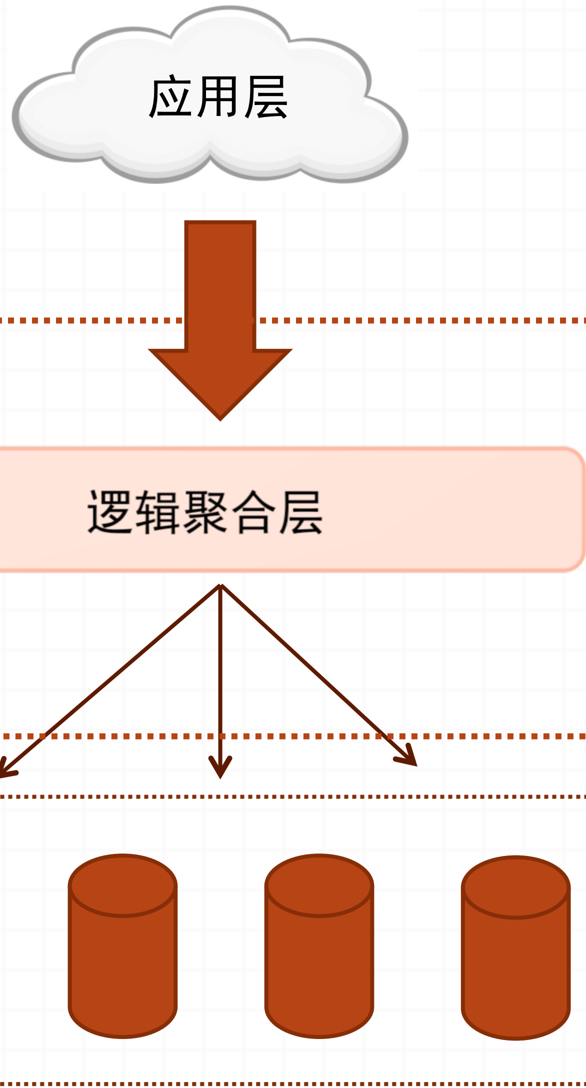
将USER_ID的分布法则写到一张表中，每次访问数据时，比如要查询USER_ID为130的行，先查询路由表，得到130存在哪个库中，再去到指定库中查询

分布式数据的关键：逻辑聚合层

分布式数据库通常要有一个中间层，它提供一个逻辑上的抽象层，将物理上分散到多个子数据库中的数据，逻辑的聚合为一个整体，让应用端以为后端是一个数据库。



分布式数据的关键：逻辑聚合层



应用连接逻辑聚合层。在应用眼中，它认为逻辑聚合层就是数据库。它不知道数据库被拆分成多个。

逻辑聚合层根据ID列，将应用传递过来的SQL发送到对应数据库。

分布式数据的关键：逻辑聚合层

SELECT username, balance FROM prod_tab WHERE user_id=130 ;

130 mod 64 = 2

USER_ID除以64余1

User_id	......
1	......
65	......
129	......
193	......
......	......

1号库

USER_ID除以64余2

User_id	......
2	......
66	......
130	......
194	......
......	......

2号库

....

USER_ID除以64余0

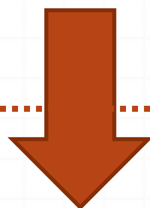
User_id	......
64	......
128	......
192	......
256	......
......	......

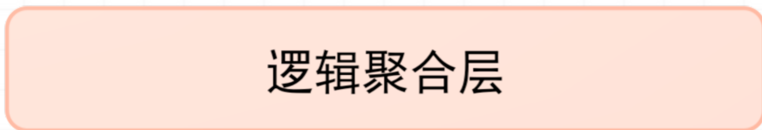
64号库

分布式数据与中间层

调用逻辑聚合层提供的接口函数，连接数据库;

SELECT name, balance FROM prod_tab WHERE user_id=130 ;

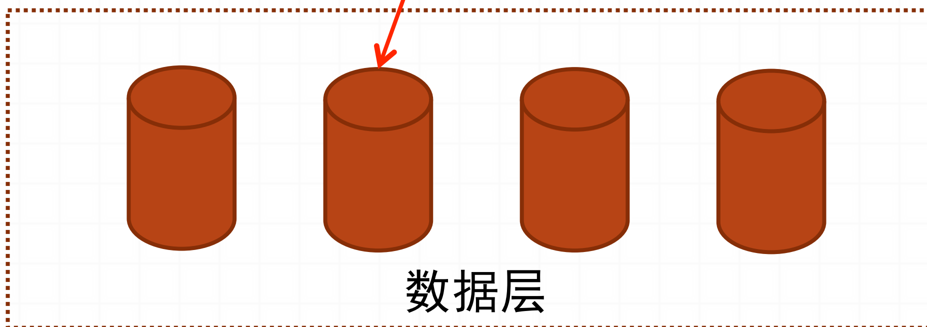




根据user_id=130判断出用户要查询的数据在2号数据库中;

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

分布式数据的关键：逻辑聚合层

逻辑聚合层的实现技术有很多种，最简单的就是JDBC外包一层，或者在数据库的监听程序外包一层。

无论哪种实现方式，在处理查询、和只包含一条DML语句的事务时，逻辑都是简单的，但处理多条DML的事务时，就比较复杂了。

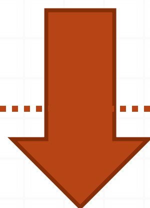
比如支付操作，A向B支付100块钱：

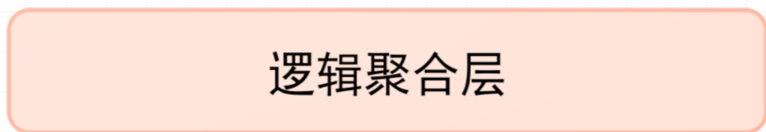
事务开始

```
Update prod_tab set bala=bala-100 where user_id=A;
Update prod_tab set bala=bala+100 where user_id=B;
Commit;
```

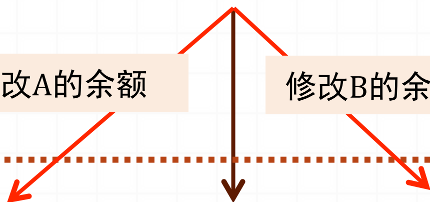
最终一致性

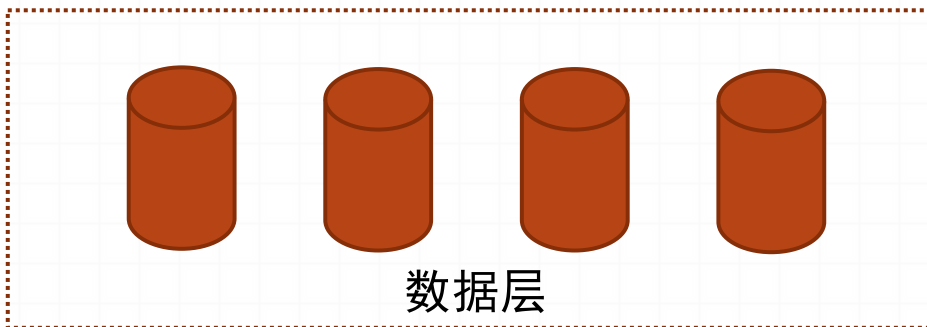
连接数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;





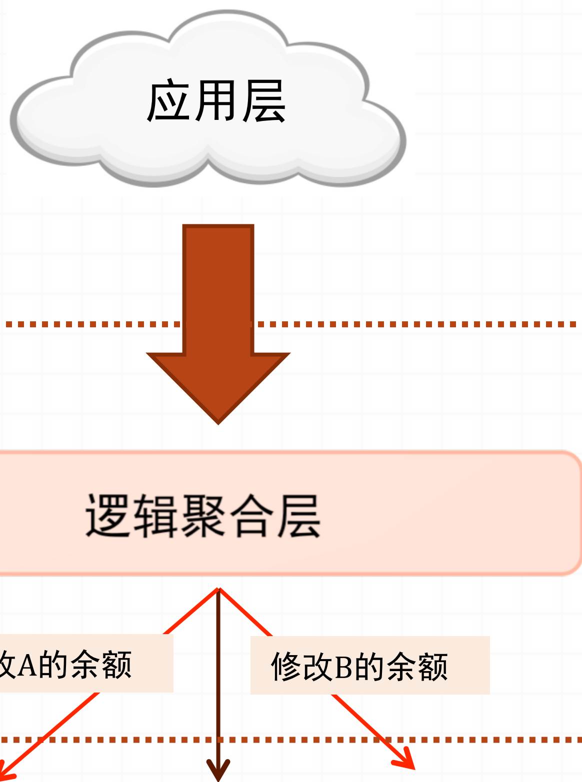
连接1号数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
结束事务 Commit;
连接4号数据库;
开始事务;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;





两个数据库分别开始自己的事务

最终一致性



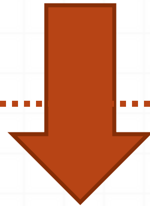
连接数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;

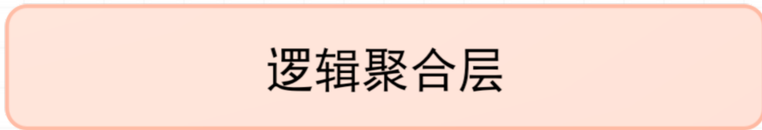
连接1号数据库;
开始事务;
Update prod_tab set bala=bala-100 where user_id=A;
结束事务 Commit;

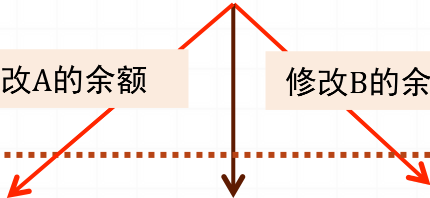
连接4号数据库;
开始事务;
Update prod_tab set bala=bala+100 where user_id=B;
结束事务 Commit;

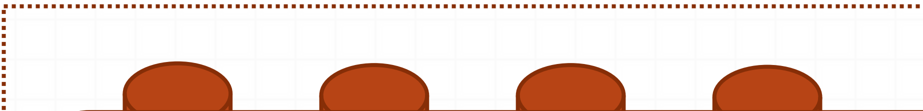
两个数据库分别开始自己的事务

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。







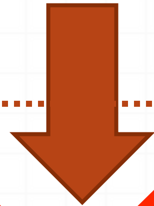


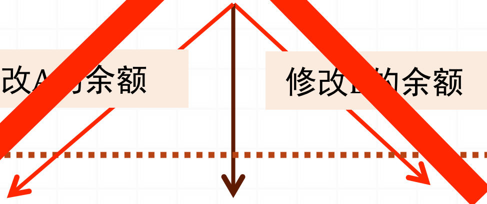
连接数据库；开始事务；
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完
Message_queue：取出消息
Update 某帐户
结束事务 Commit;
Message_queue：取出消息
Update 某帐户
结束事务 Commit;

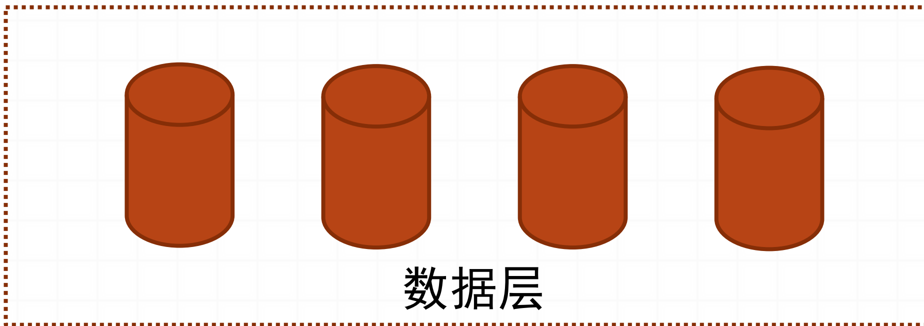
连接1号数据库;
开始事务;
…………

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。



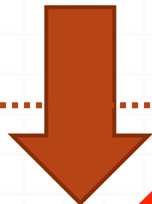


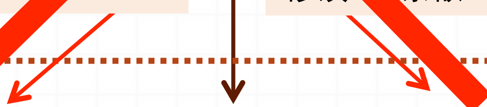


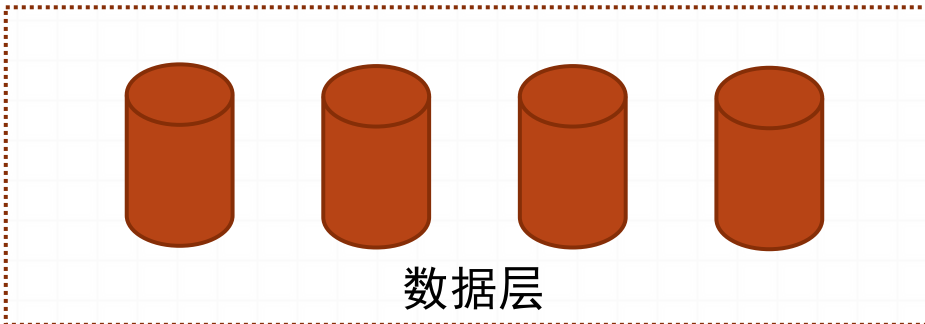
连接数据库；开始事务;
Insert into tran_log values(tran_id, A,B,100);
Message_queue入队(tran_id, A, -100);
Message_queue入队(tran_id, B, +100);
事务结束 返回信息：交易成完
Message_queue：取出消息
Update 某帐户
结束事务 Commit;
Message_queue：取出消息
Update 某帐户
结束事务 Commit;

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;
…………

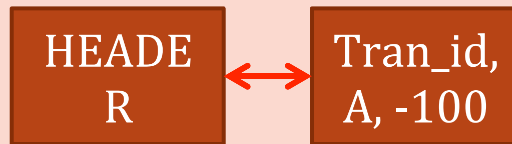
两个数据库分别开始自己的事务

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)

Tran_id,
B, +100

Message_qu… …n_id、帐户名和其他信息。
连接数据库…
查询tran_l…
查询msg_l… …在。
开始事务;
Update 某帐…



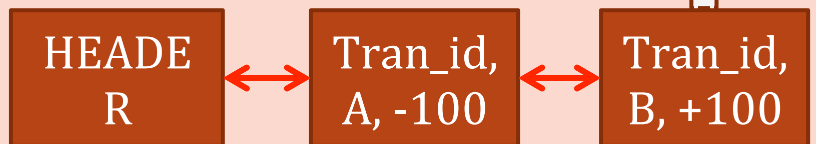
Insert into msg_log values(Tran_id, 某帐户，...);
事务结束（update insert在同一数据库）
Message_queue：去除消息(remove_message)

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)

Update PROD_TABLE set balance= balance+100 where user_id=B

M ... 信息。
连
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
Insert into m
事务结束（u
Message_que



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)

节约成本的方法：适当分布式

分布式大大增加了系统的复杂程度，过度的分布式反而有可能增加成本。

节约成本的方法：适当分布式

我们有一套700T的OLTP库，不使用分布式时，700T数据还在一个数据库中，无论容量还是性能，都会有问题。

我们选择的分布式方案，使用SUN SPACK主机，全闪存存储，加上Oracle数据库。所有数据分为近30个子Oracle数据库存储。

SPACK主机优点是单机CPU核心多，可以达到单机512核。按每核心20ms处理一个事务算的心，单机一秒可处理25600笔事务。

Oracle数据库，是并发机制做的最好的数据库，可以最大程度上减少软件瓶颈对性能的影响。

30个左右的子库，虽然也是分布式，但分布的不算多，管理成功并不高。30台SUN SPACK主机，700TB存储空间，成本也只是千万级别。Oracle license费用，当有如此数据规模时，License费用好商量的。

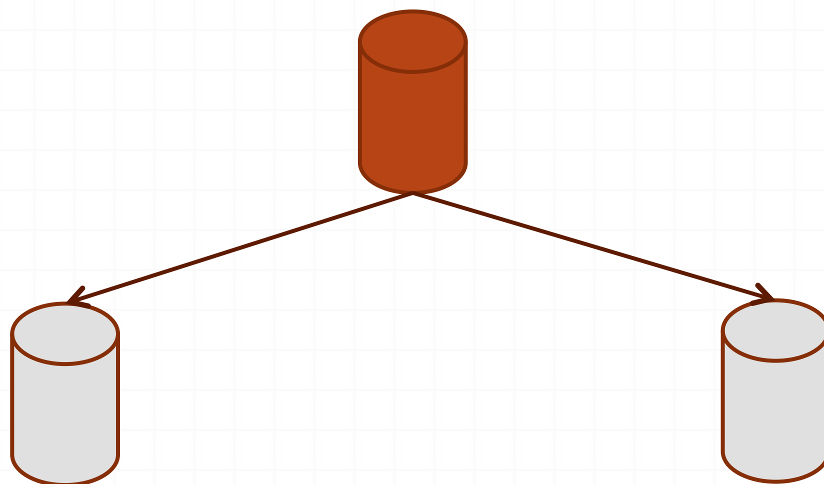
节约成本的方法：适当分布式

因为需要事务，NoSQL可以不再考虑，如果使用MySQL的分布式方案，大概需要多少成本呢。

通常MySQL为了数据安全，都是采用一拖二模式（一主两备）。按一台主机在做完RAID后有效空间5TB算，700T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PC Server，也是一笔庞大的费用。和使用Oracle的架构，其实差不多。

另外，MySQL的并发性比Oracle要差，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。



节约成本的方法：适当分布式

分布式是可以解决扩展性和性能&容量问题，但适当分布式，才能真正节省成本、减少工作量。

THANKS

吕海波（VAGE）