DTCC 2010-2022

第十三届中国数据库技术大会

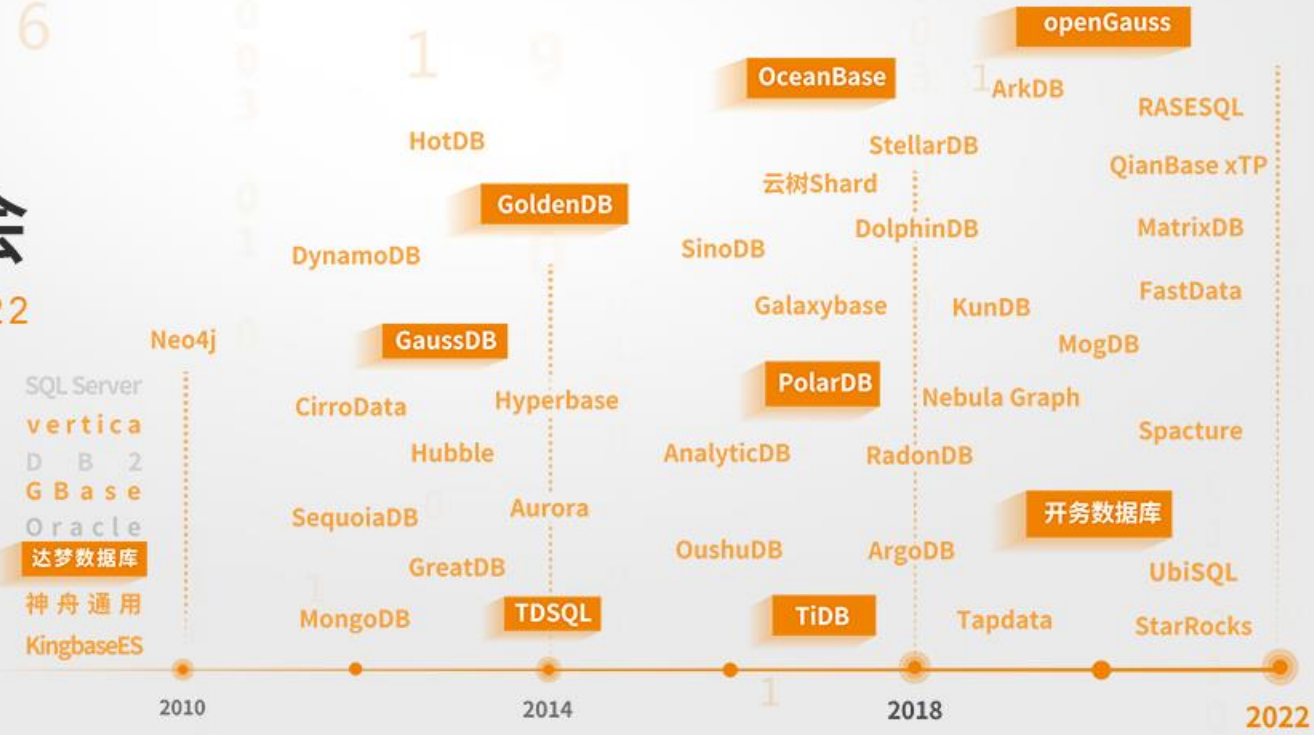
DATABASE TECHNOLOGY CONFERENCE CHINA 2022

数据智能 价值创新

IT168.com ChinaUnix.net ITPUB

线上直播 | 2022/12/14-16

数据来源：数据库产品上市商用时间



用最少的代码量 取得最大程度的性能提升

---- PostgreSQL 内核深度优化

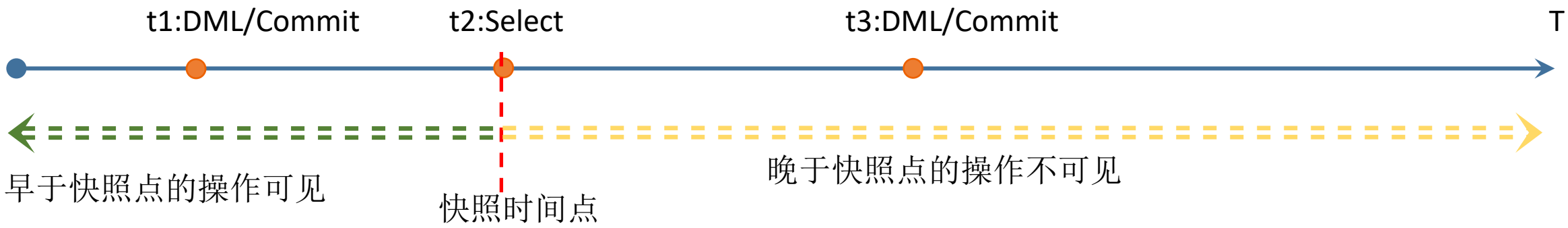
吕海波　杭州美创科技 技术研究院数据库内核专家

内容介绍

- PostgreSQL内核优化之一：创建快照的改进
- Oracle一致性的引发的考量
- PostgreSQL内核优化之二：LWLockAcquire自璇的潜在问题

现代数据库基础概念 -- 快照

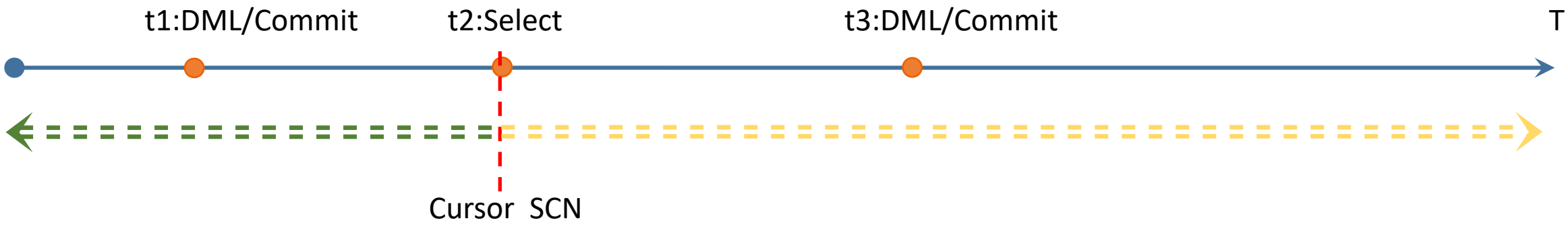
- 什么是快照
  - 字面上，快照是对数据库拍个照。
  - 快照本身，类似一个时间点。
  - 快照的作用：主要用于“可见性判断”。



现代数据库基础概念 -- 快照：Oracle中的快照

- Oracle的快照

t2时刻开始的Select，可以看到t1 DML的结果，但看不到t3 DML的数据

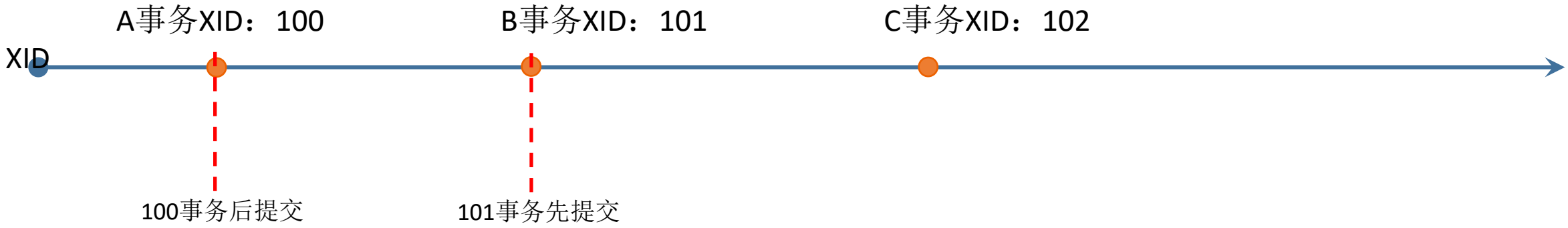


Oracle使用SCN做为”快照“的时间。

PG/MySQL使用XID。

现代数据库基础概念 -- 快照：PG中的快照（一）

- PG快照的本质：XID，事务开始序号

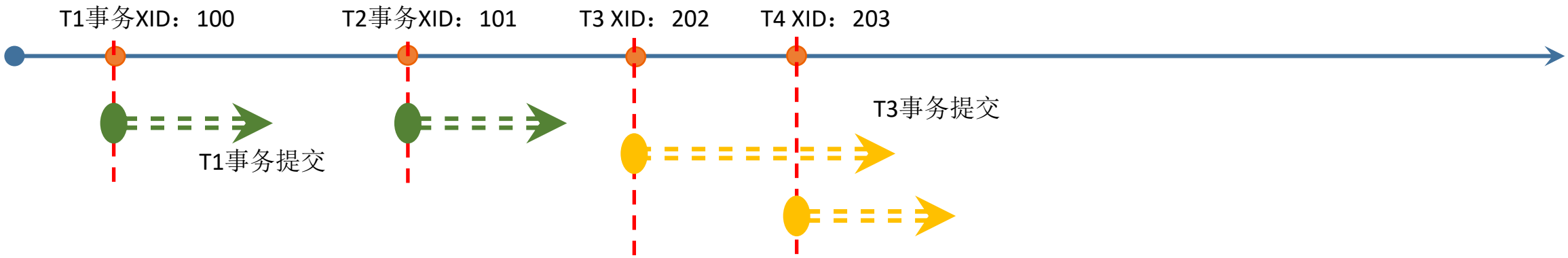


现代数据库基础概念 -- 快照：PG中的快照（二）

- PG快照的本质：XID，事务开始序号

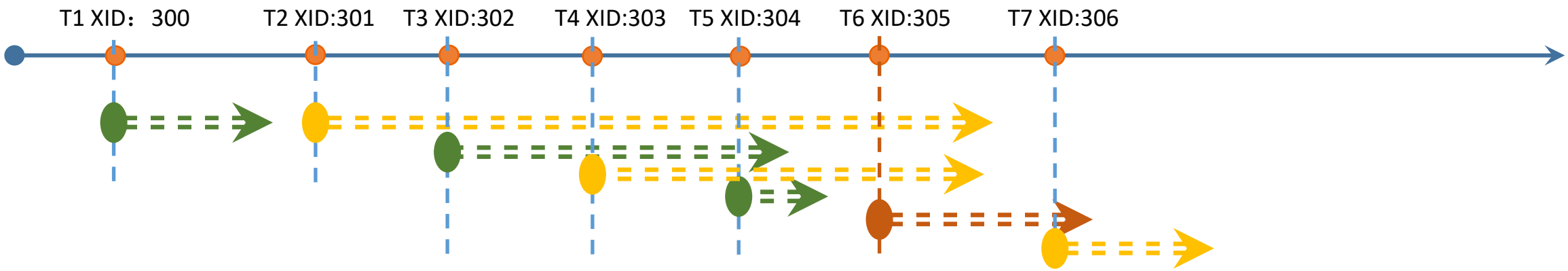
现代数据库基础概念 -- 快照：PG中的快照（三）

- PG快照的本质：XID，事务开始序号



现代数据库基础概念 -- 快照：PG中的快照（四）

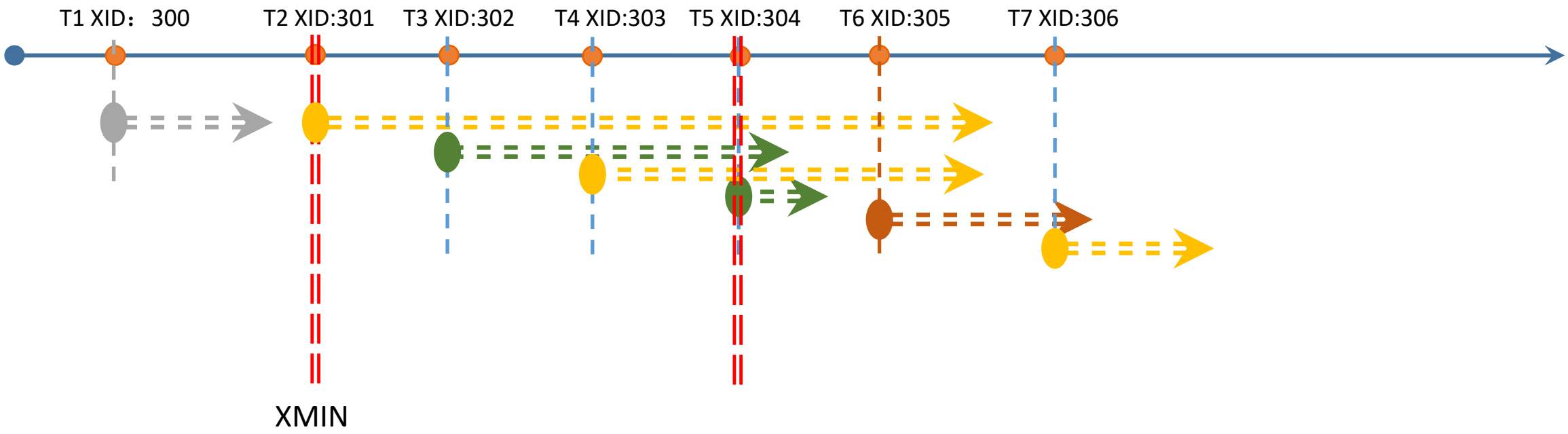
- PG快照的本质：XID，事务开始序号



T6开始时，记录下所有的活动事务到一个列表中: 301, 303, 306，称为活动事务列表

现代数据库基础概念 -- 快照：PG中的快照（五）

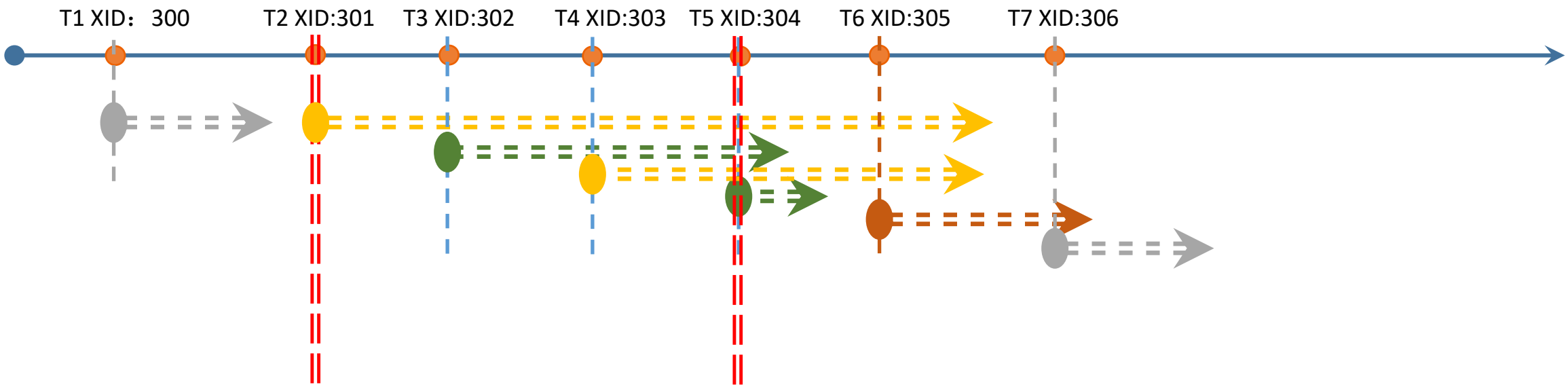
- PG快照的本质：XID，事务开始序号



T6开始时，记录下所有的活动事务: 301, 303, 306

现代数据库基础概念 -- 快照：PG中的快照（五）

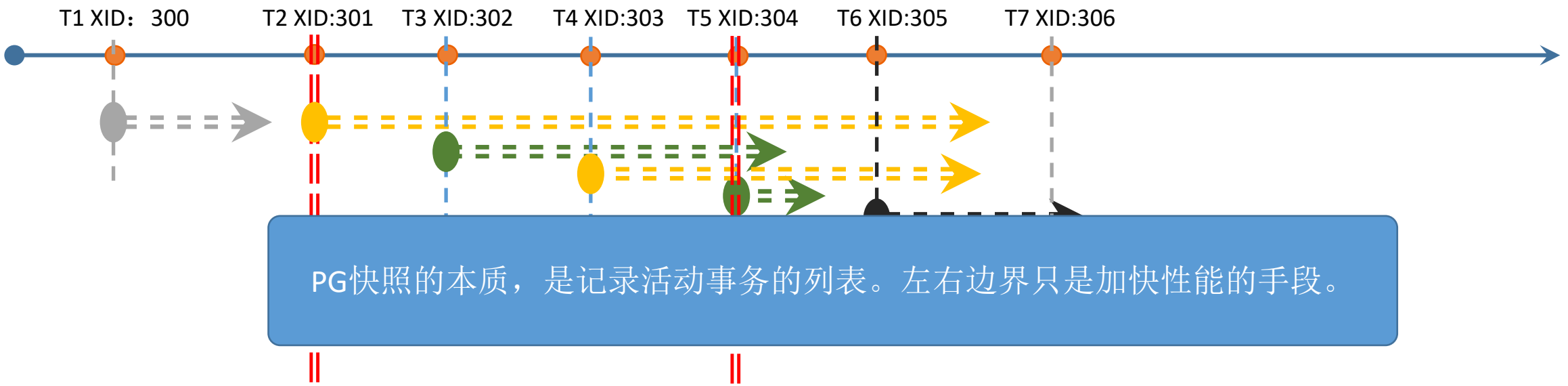
- PG快照的本质：XID，事务开始序号



T6开始时，记录下所有的活动事务: 301, 303

现代数据库基础概念 -- 快照：PG中的快照（六）

- PG快照的本质：XID，事务开始序号

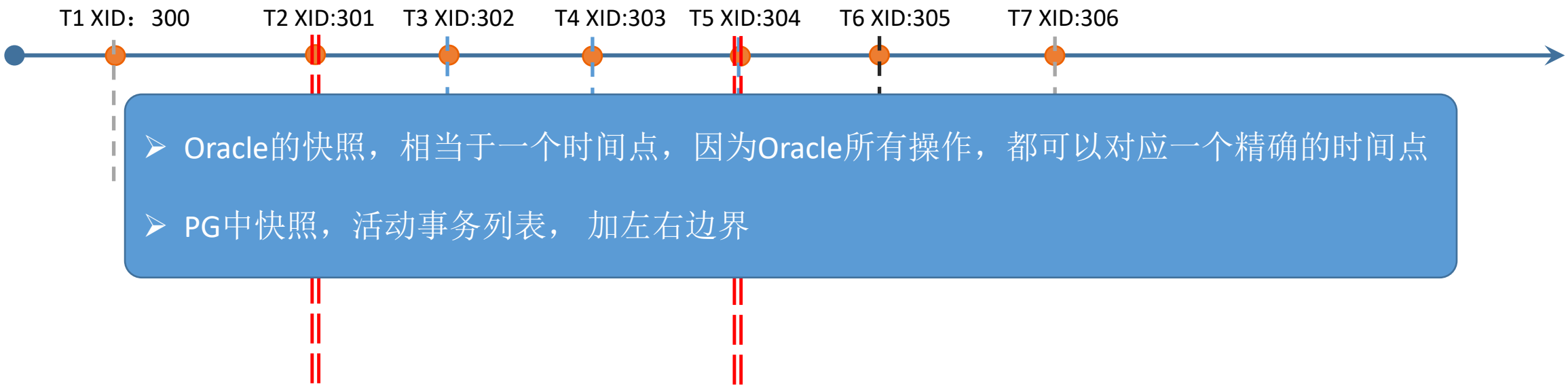


左边界xmin: 301
右边界xmax: 304
活动事务列表: 301, 303

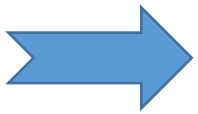
xmin：xmax：xip[0],xip[1] ,……, xip[xcnt]

现代数据库基础概念 -- 快照：PG中的快照（六）

- PG快照的本质：XID，事务开始序号



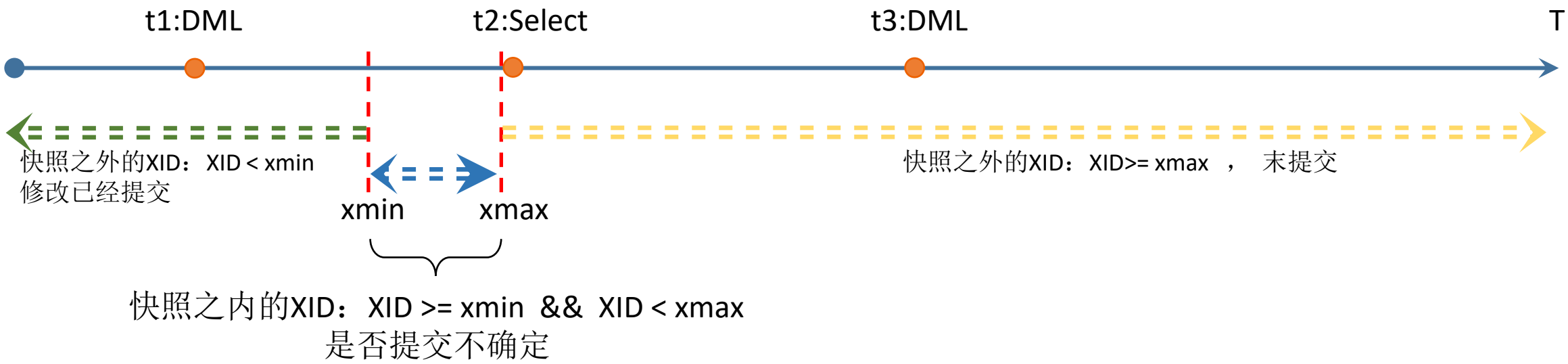
左边界xmin: 301
右边界xmax: 304
活动事务列表: 301, 303



xmin：xmax：xip[0],xip[1] ,……, xip[xcnt]

现代数据库基础概念 -- 快照：PG中的快照（总结）

- PG中的快照：时间点变时间段

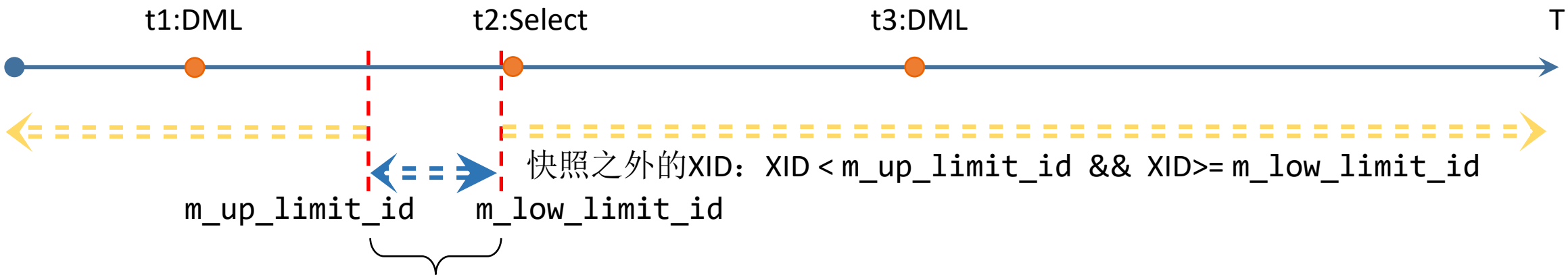


事务XID在xmin和xmax之间：搜索活动事务列表

（减少搜索活动事务列表的次数）

现代数据库基础概念 -- 快照：MySQL中的快照

- MySQL的实现方法



快照之内的XID：XID >= m_up_limit_id && XID < m_low_limit_id

m_creator_trx_id ：当前事务ID，Select为0
m_low_limit_id ：当前的最大事务ID（等同于PG中的snapshot->xmax）
m_low_limit_no ：同上
m_up_limit_id ：最早的活动事务ID（等同于PG中的snapshot->xmin）

具体规同PG：在m_up_limit_id 和 m_low_limit_id 之间，属于快照之内，需查询活动事务列表。在快照范围之外无需查询活动事务列表

PostgreSQL快照的创建

- 创建快照的函数：GetSnapshotData

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取关键的共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
for(遍历所有Session)
{
　　对比所有Session中的pgxact->xmin，得到最小的pgxact->xmin
　　对比所有Session中的pgxact->xid，得到最小的pgxact->xid
　　如pgxact->xid 在 ShmemVariableCache->latestCompletedXid之下（小于），将pgxact->xid加入活动事务列表
}
LWLockRelease(ProcArrayLock)；释放共享锁
将得到的数据写入快照snapshot

PostgreSQL快照的创建

- GetSnapshotData流程总结

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取关键的共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
for(遍历所有Session)
{
　　对比所有Session中的pgxact->xmin，得到最小的pgxact->xmin
　　对比所有Session中的pgxact->xid，得到最小的pgxact->xid
　　如pgxact->xid 在 ShmemVariableCache->latestCompletedXid之下（小于），将pgxact->xid加入活动事务列表
}
LWLockRelease(ProcArrayLock)；释放共享锁
将得到的数据写入快照snapshot

ProcArrayLock锁，保护与Session相关的数据。它只有一把，共享模式的ProcArrayLock锁将阻塞：
　　会话连接
　　事务提交

主要是事务提交，对于一秒中上千次、上万次事务的OLTP系统（Oracle中每秒几万次提交的系统很多），ProcArrayLock共享锁将极大的影响性能，只能靠CPU单核芯频率、极度优化的代码，减少锁的持有时间。

PostgreSQL内核优化：为提交确定顺序（一）

- 快照如何建立：

会话：	Session1	Session2	Sess 3	Sess 4	Sess 5	Sess 6			
pgxact->xid：	835	836	837	838	839	840			

commit_id：每次提交时加1

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取关键的共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
for(遍历所有Session)
{
　　对比pgxact->xid、pgxact->xmin，找到最小的pgxact->xid
　　pgxact->xid 在 ShmemVariableCache->latestCompletedXid之下的（小于），
　　将pgxact->xid加入活动事务列表
}
LWLockRelease(ProcArrayLock)；释放共享锁
将得到的数据写入快照snapshot

PostgreSQL内核优化：为提交确定顺序（一）

- 快照如何建立：

会话：	Session1	Session2	Sess 3	Sess 4	Sess 5	Sess 6			
pgxact->xid：	835	836	837	838	839	840			
事务数组下标：	835	836	837	838	839	840			
提交ID：	98	101	0	99	100	0			

commit_id：每次提交时加1

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取关键的共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
for(遍历所有Session)
{
　　对比pgxact->xid、pgxact->xmin，找到最小的pgxact->xid
　　pgxact->xid 在 ShmemVariableCache->latestCompletedXid之下的（小于），
　　将pgxact->xid加入活动事务列表
}
LWLockRelease(ProcArrayLock)；释放共享锁
将得到的数据写入快照snapshot

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
读取共享变量：commit_id
LWLockRelease(ProcArrayLock)；释放共享锁
for(txid = min_pgxact_xid; txid<=latestCompletedXid; txid++)
{ // 从min_pgxact_xid到latestCompletedXid，扫描事务数组
　　寻找活动事务的最小txid
　　将活动事务加入列表
}
将得到的数据写入快照snapshot

PostgreSQL内核优化：为提交确定顺序（二）

- 快照如何建立：

会话：	Session1	Session2	Sess 3	Sess 4	Sess 5	Sess 6			
pgxact->xid：	835	836	837	838	839	840			
	commit_id： 99								
事务数组下标：	835	836	837	838	839	840			
提交ID：	98	101	0	99	100	0			

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取关键的共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
for(遍历所有Session)
{
　　对比pg
　　pgxact
　　将pgxa
}
LWLockRelease(ProcArrayLock)；释放共享锁
将得到的数据写入快照snapshot

只要提交有序可查，你只要记录下当前时间点，哪些提交在你之前，哪些提交在你之后，不就一目了然了吗。

LWLockAcquire(ProcArrayLock, LW_SHARED)；得到共享锁
读取共享变量：ShmemVariableCache->latestCompletedXid　// 得到xmax
读取共享变量：commit_id
　　将活动事务加入列表
}
将得到的数据写入快照snapshot

PostgreSQL内核优化：事务数组

- 快照如何建立：

会话：	Session1	Session2	Sess 3	Sess 4	Sess 5	Sess 6			
pgxact->xid：	835	836	837	838	839	840			
	commit_id： 99								
事务数组下标：	835	836	837	838	839	840			
提交ID：	98	101	0	99	100	0			

事务数组如何设计?

链表式的结构，效率太低，因为下一个链表Node的地址不固定，CPU的硬件预取很难发挥作用，极大的影响性能。（硬件预取对性能的影响非常大）

如果仍采用数组方式，以XID为下标，那么数组大小将是21亿，就算每个数组元素都是一个4字节整型值（commit_id），事务数组共将占用近8GB内存。关键其中大部分内存都将是空闲的。

解决方法也非常简单，取XID的低n位即可，比如低16位（XID & 0xffff），可以容纳65535个事务，占256KB内存。

PostgreSQL内核优化：对提交的影响

- 快照如何建立：

会话：	Session1	Session2	Sess 3	Sess 4	Sess 5	Sess 6			
pgxact->xid：	835	836	837	838	839	840			
	commit_id： 99								
事务数组下标：	835	836	837	838	839	840			
提交ID：	98	101	0	99	100	0			

;

提交时增加的工作：

- 读取commit_id到事务数组
- 推进commit_id

不需要额外的锁操作，提交本身就有独占的ProcArrayLock。

PostgreSQL内核优化：总结

- GetSnapshotData改进逻辑 : ShmemVariableCache的定义 与 初始化

在transam.h中定义事务数组、commit_id等共享变量
的varsup.c进行基本的初始化
在ipci.c、shmem.c中分配内存，初始化
在xlog.c的StartupXLOG ()函数中，增加对commit_id等变量的初始化

在ProcArrayEndTransactionInternal()中增加commit_id自增等操作
修改GetSnapshotData()函数，替换遍历Session的循环

PostgreSQL内核优化：测试对比

- 性能影响测试：创建一千个Session

测试脚本1，创建1000个连接：

```
[postgres@localhost scripts]$ cat session.sh
psql -p6014 <<EOF
select pg_sleep($1);
EOF

[postgres@localhost scripts]$ cat ts.sh | head -n 10
./session.sh $1 &
./session.sh $1 &
./session.sh $1 &
……

[postgres@localhost scripts]$ cat ts.sh | wc -l
1001
```

PostgreSQL内核优化：测试对比

- 性能影响测试：创建100个事务

测试脚本2，创建100个活动事务（发起事务不提交）：

```
[postgres@localhost scripts]$ cat trx.sh
psql -p6014 <<EOF
begin;
insert into t3 values(1, 'AAAAAA', 2, 'BBBBBB');
select pg_sleep($1);
commit;
EOF
```

```
[postgres@localhost scripts]$ cat mtrx.sh  | head -n 10
./trx.sh $1 &
./trx.sh $1 &
./trx.sh $1 &
……

[postgres@localhost scripts]$ cat mtrx.sh  | wc -l
101
```

PostgreSQL内核优化：测试对比

- 性能影响测试：使用sysbench读写混合测试

	TPS per sec.
正常	1700 - 1800
1000 Session 100事务	1500 - 1600
1000 Session	

虽然增加了Commit_id，但并没有使用commit_id控制可见性，快照还是只使用XID。

```
[  54s] threads: 28, tps: 1779.50, reads/s: 24913.45, writes/s: 7125.49, response time: 26.22ms (95%)
[  56s] threads: 28, tps: 1772.50, reads/s: 24846.54, writes/s: 7095.01, response time: 25.93ms (95%)
[  58s] threads: 28, tps: 1780.50, reads/s: 24911.00, writes/s: 7106.50, response time: 26.68ms (95%)
[  60s] threads: 28, tps: 1777.00, reads/s: 24885.97, writes/s: 7116.49, response time: 26.62ms (95%)
OLTP test statistics:
    queries performed:
        read:                            1490888
        write:                           425968
        other:                           212984
        total:                           2129840
    transactions:                        106492 (1774.68 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 1916856 (31944.20 per sec.)
    other operations:                    212984 (3549.36 per sec.)
```

```
[  52s] threads: 28, tps: 1588.50, reads/s: 22239.50, writes/s: 6350.50, response time: 29.00ms (95%)
[  54s] threads: 28, tps: 1591.00, reads/s: 22280.49, writes/s: 6368.50, response time: 28.72ms (95%)
[  56s] threads: 28, tps: 1595.50, reads/s: 22311.02, writes/s: 6381.01, response time: 30.19ms (95%)
                                                                          ime: 29.26ms (95%)
                                                                          me: 28.97ms (95%)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 1713258 (28550.50 per sec.)
    other operations:                    190362 (3172.28 per sec.)
```

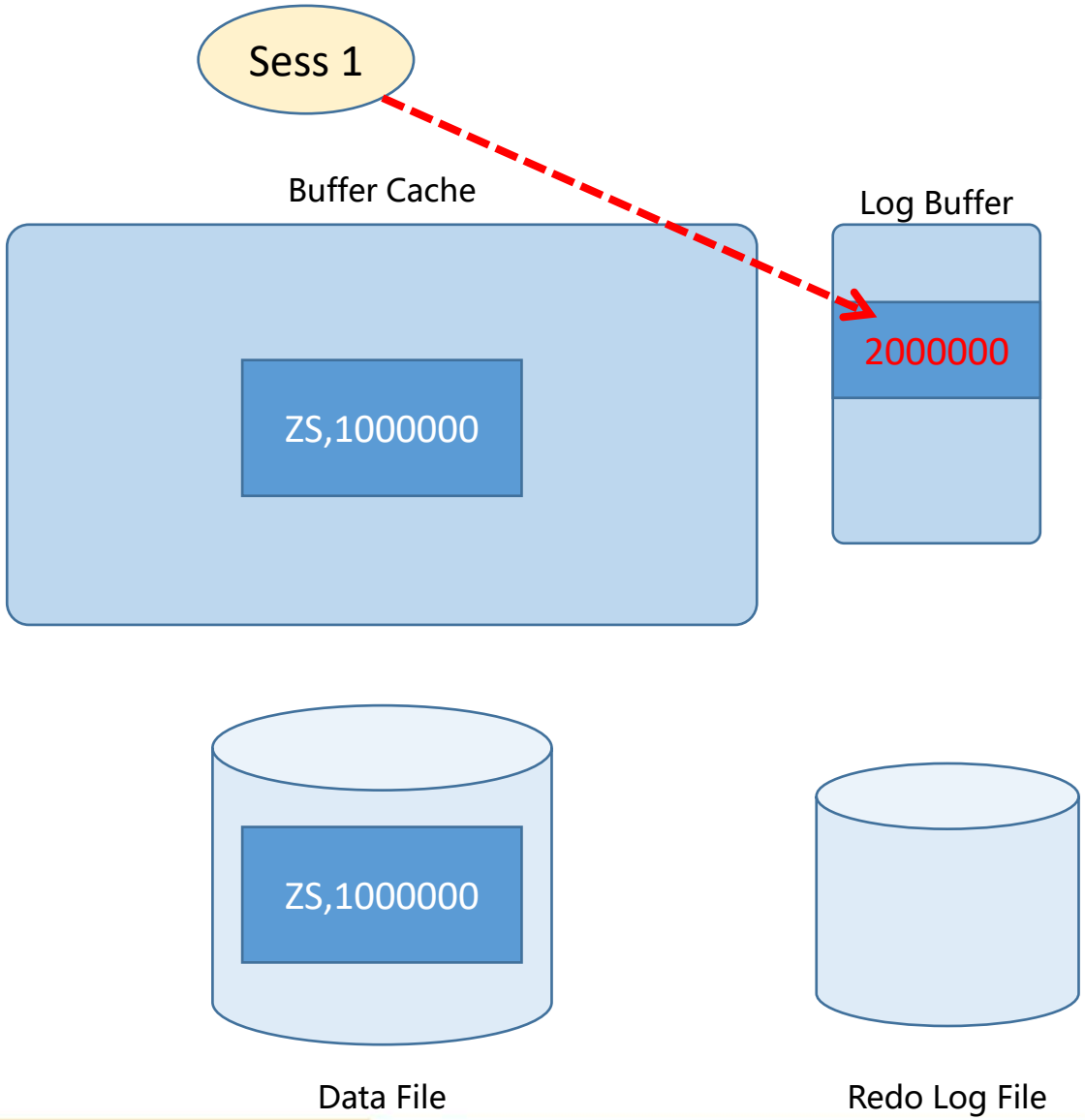
```
[  56s] threads: 28, tps: 1766.50, reads/s: 24701.05, writes/s: 7062.01, response time: 26.58ms (95%)
[  58s] threads: 28, tps: 1773.51, reads/s: 24869.09, writes/s: 7113.03, response time: 27.31ms (95%)
[  60s] threads: 28, tps: 1776.00, reads/s: 24861.51, writes/s: 7097.50, response time: 27.19ms (95%)
OLTP test statistics:
    queries performed:
        read:                            1485960
        write:                           424560
        other:                           212280
        total:                           2122800
    transactions:                        106140 (1768.82 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 1910520 (31838.81 per sec.)
    other operations:                    212280 (3537.65 per sec.)
```

Oracle的一致性问题 -- 消失的一百万


```
SQL> select * from account;
        ID NAME              SUMM
---------- ---------- ----------
         1 ZS             1000000

SQL> update account set summ=2000000 where name='ZS';

已更新 1 行。

SQL> commit;
commit
     *
第 1 行出现错误:
ORA-03113: 通信通道的文件结尾 进程 ID:32730
会话 ID: 197 序列号: 6241
```

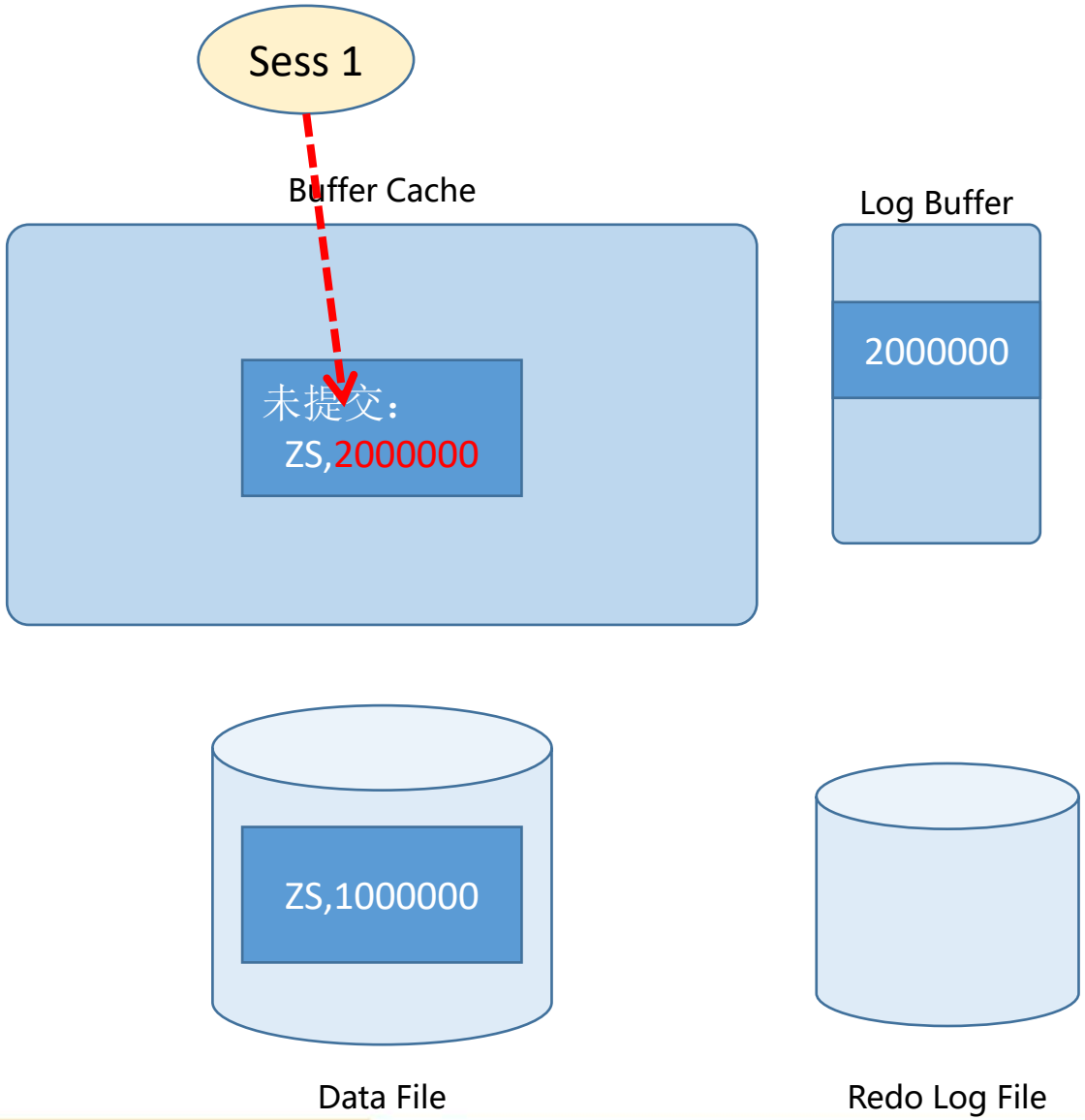
```
SQL> select * from account;
        ID NAME              SUMM
---------- ---------- ----------
         1 ZS             2000000

SQL> conn u1/a
已连接。

SQL> select * from account;
        ID NAME              SUMM
---------- ---------- ----------
         1 ZS             1000000
```

Oracle的一致性问题 -- 消失的一百万： 复盘

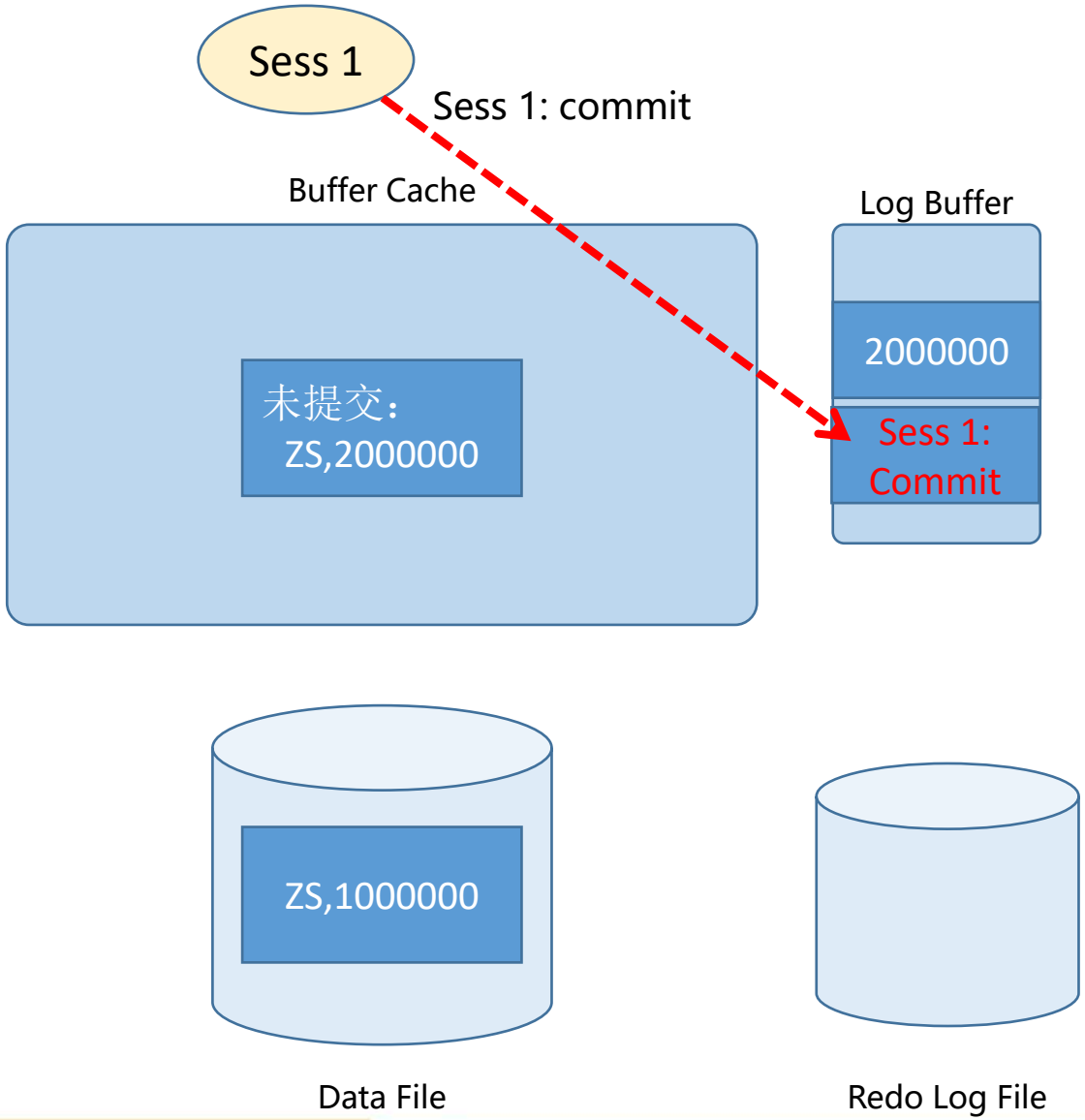


Oracle的一致性问题 -- 消失的一百万： 复盘

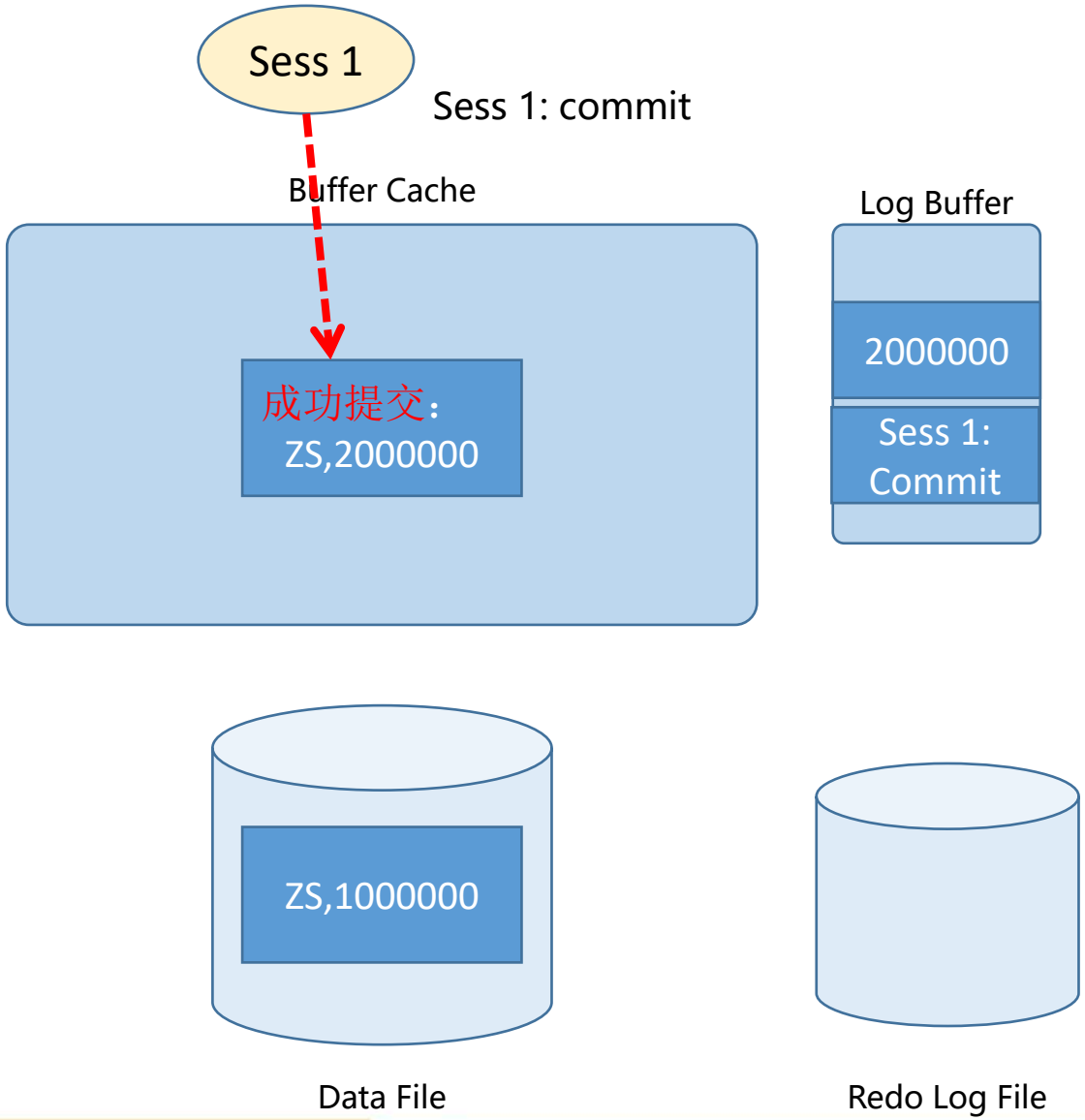
WAL：Write-Ahead Logging



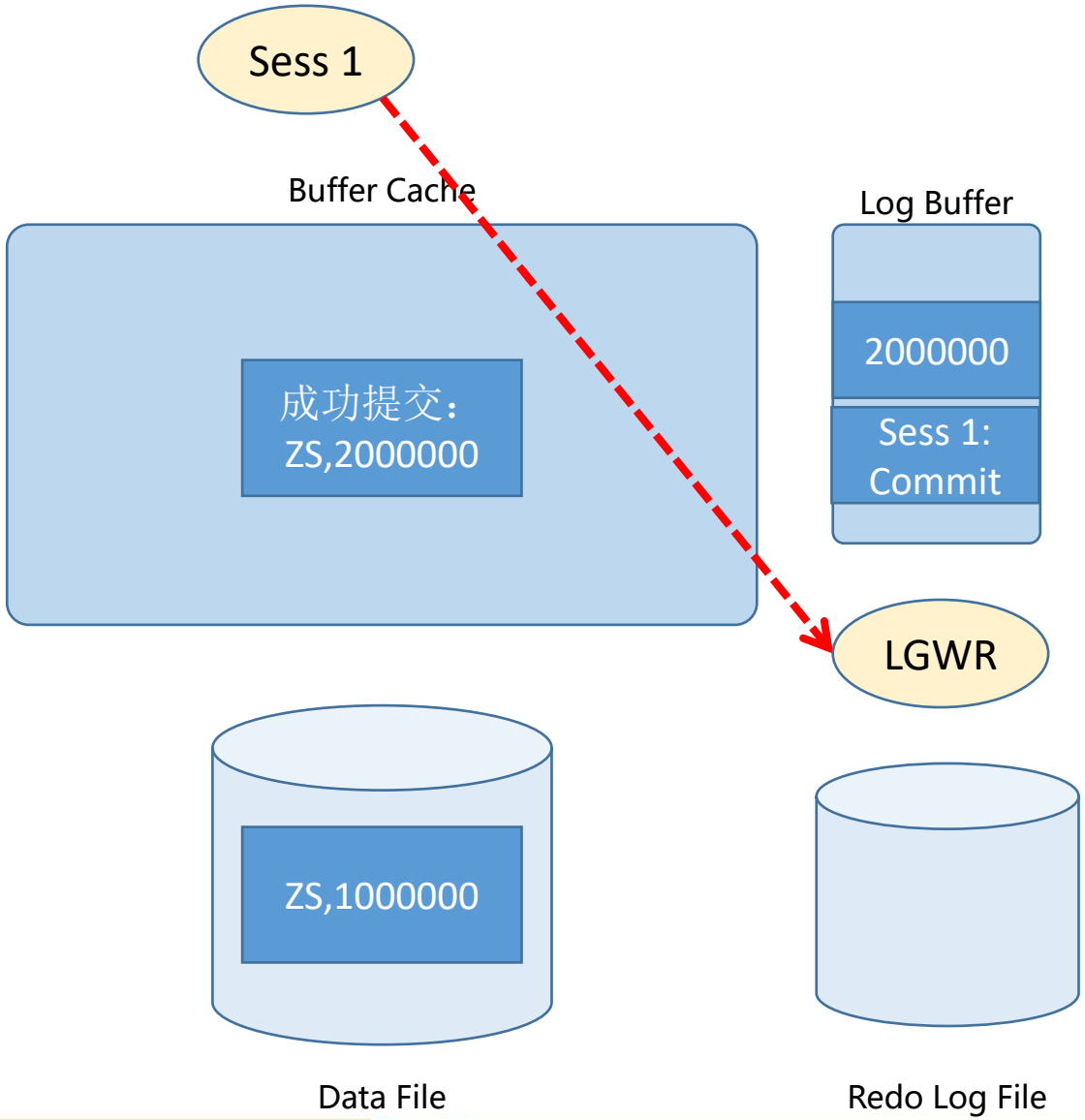
Oracle的一致性问题 -- 消失的一百万： 复盘



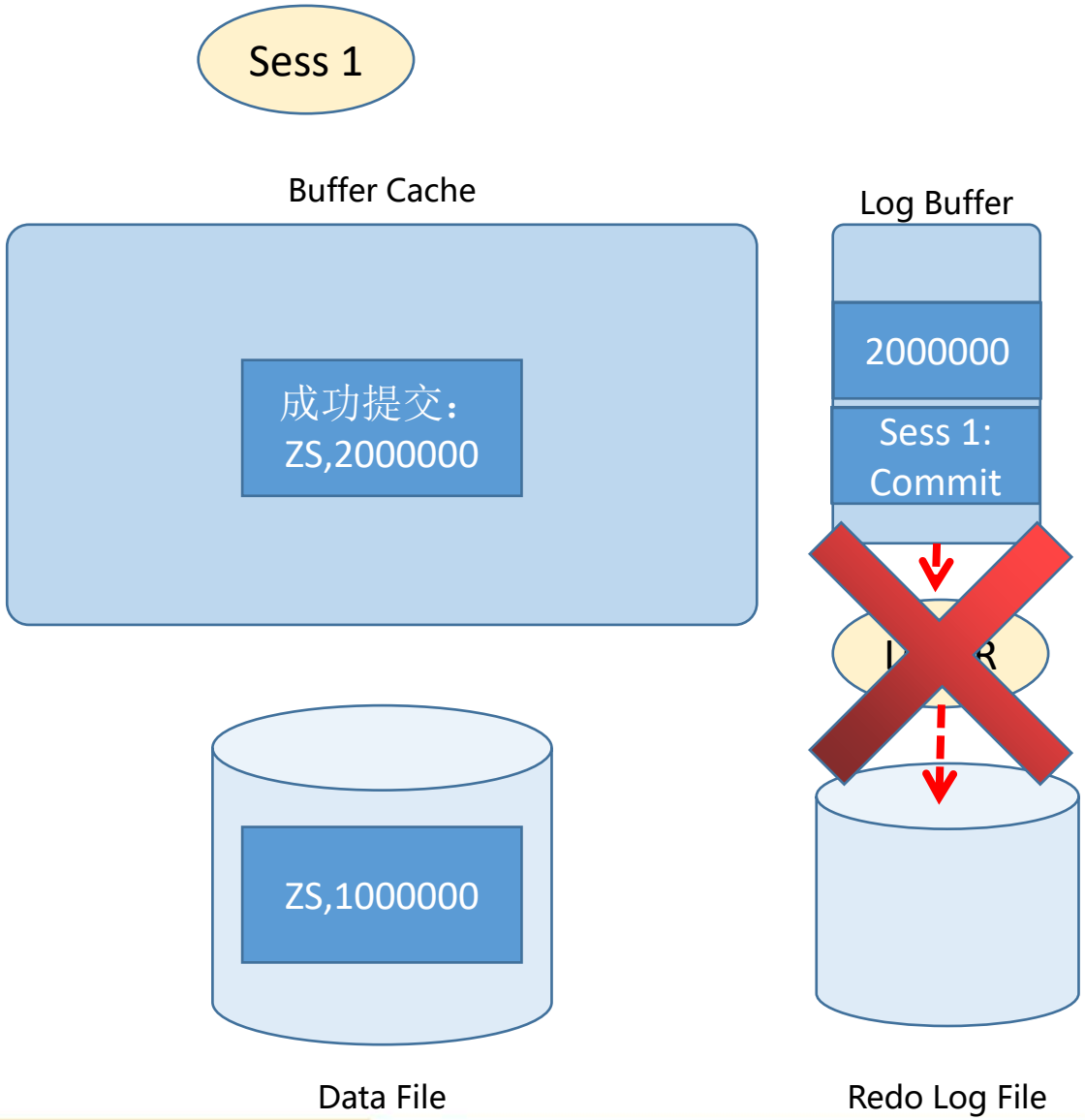
Oracle的一致性问题 -- 消失的一百万： 复盘



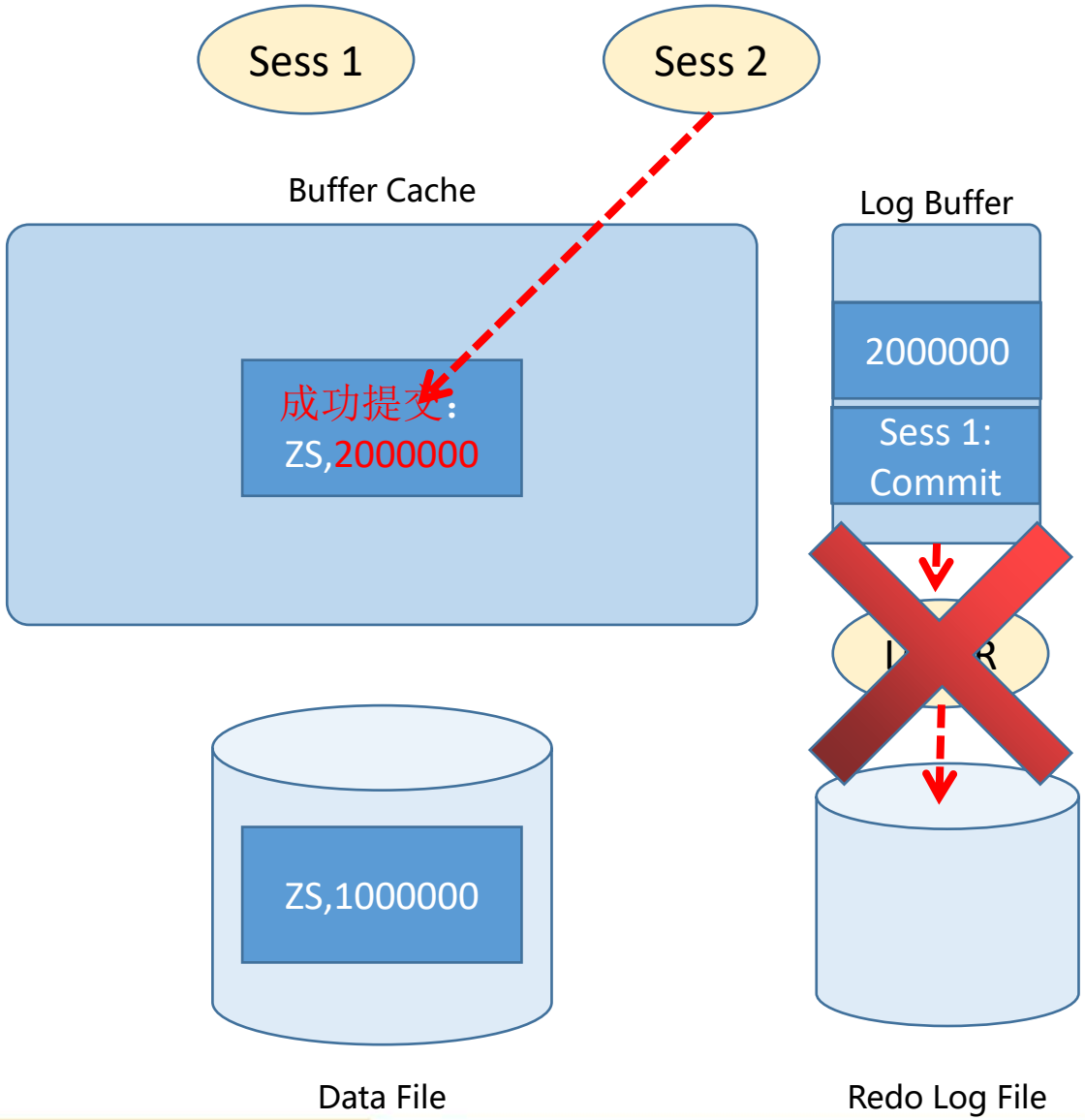
Oracle的一致性问题 -- 消失的一百万： 复盘



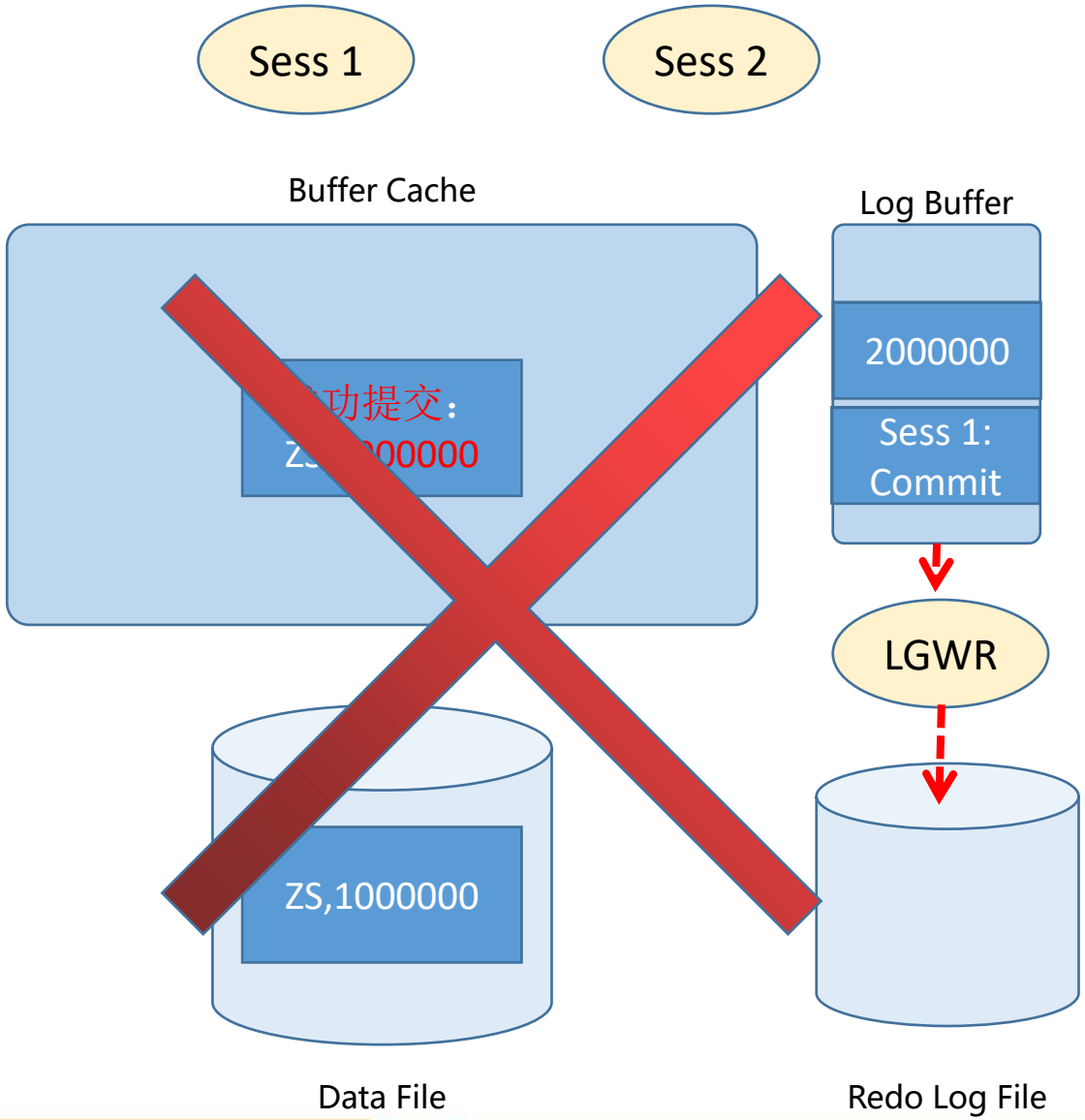
Oracle的一致性问题 -- 消失的一百万： 复盘



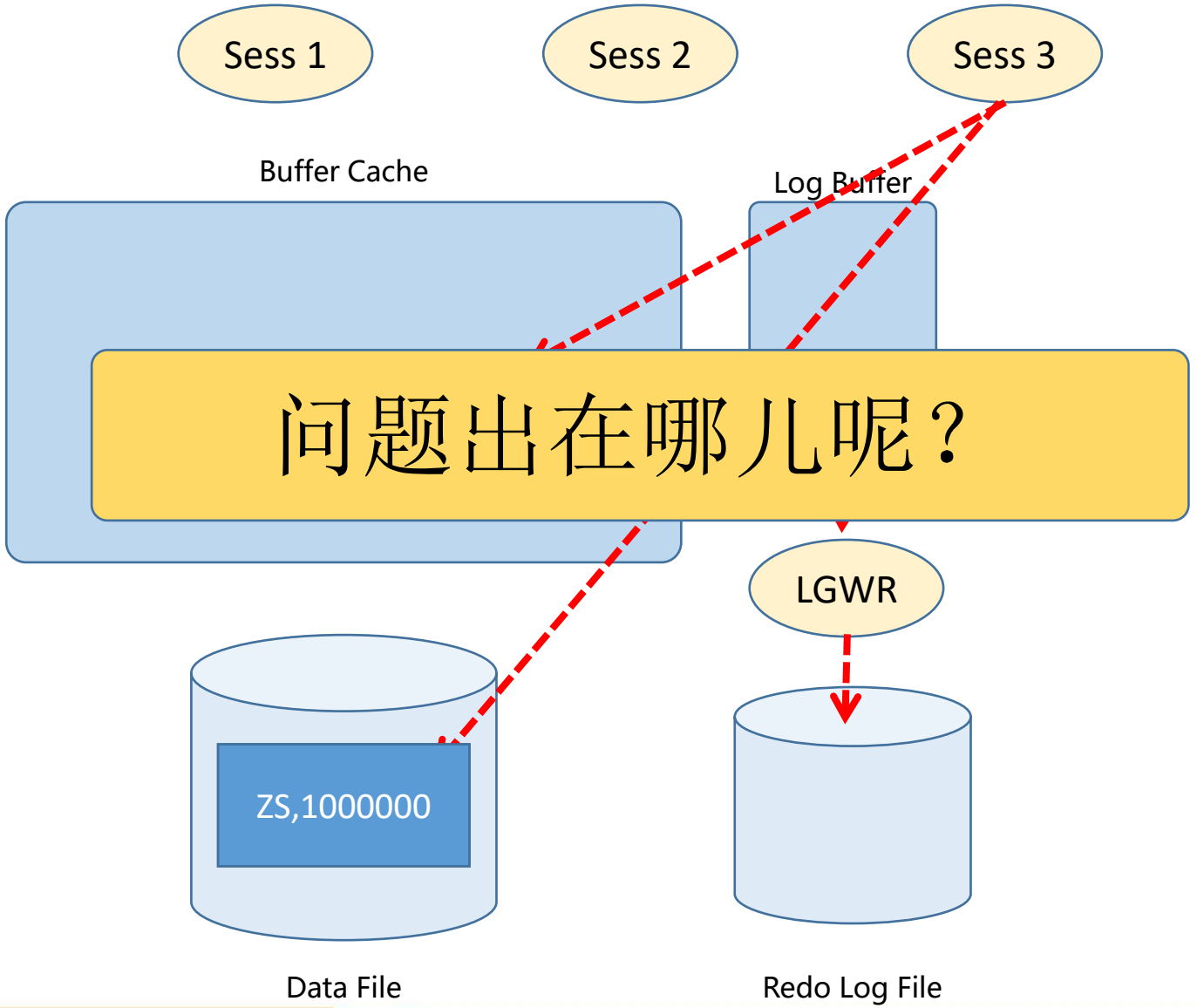
Oracle的一致性问题 -- 消失的一百万： 复盘



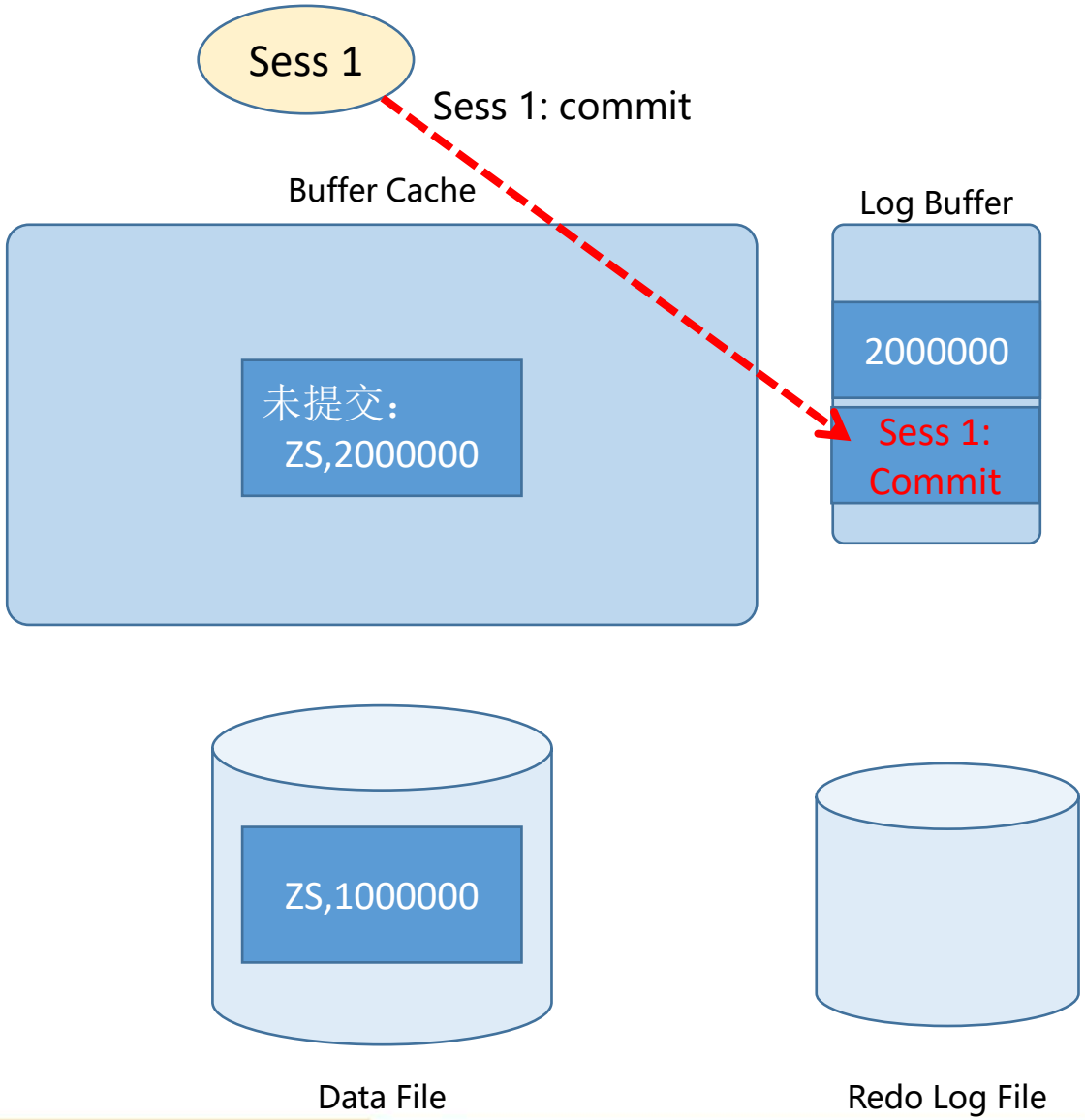
Oracle的一致性问题 -- 消失的一百万： 复盘



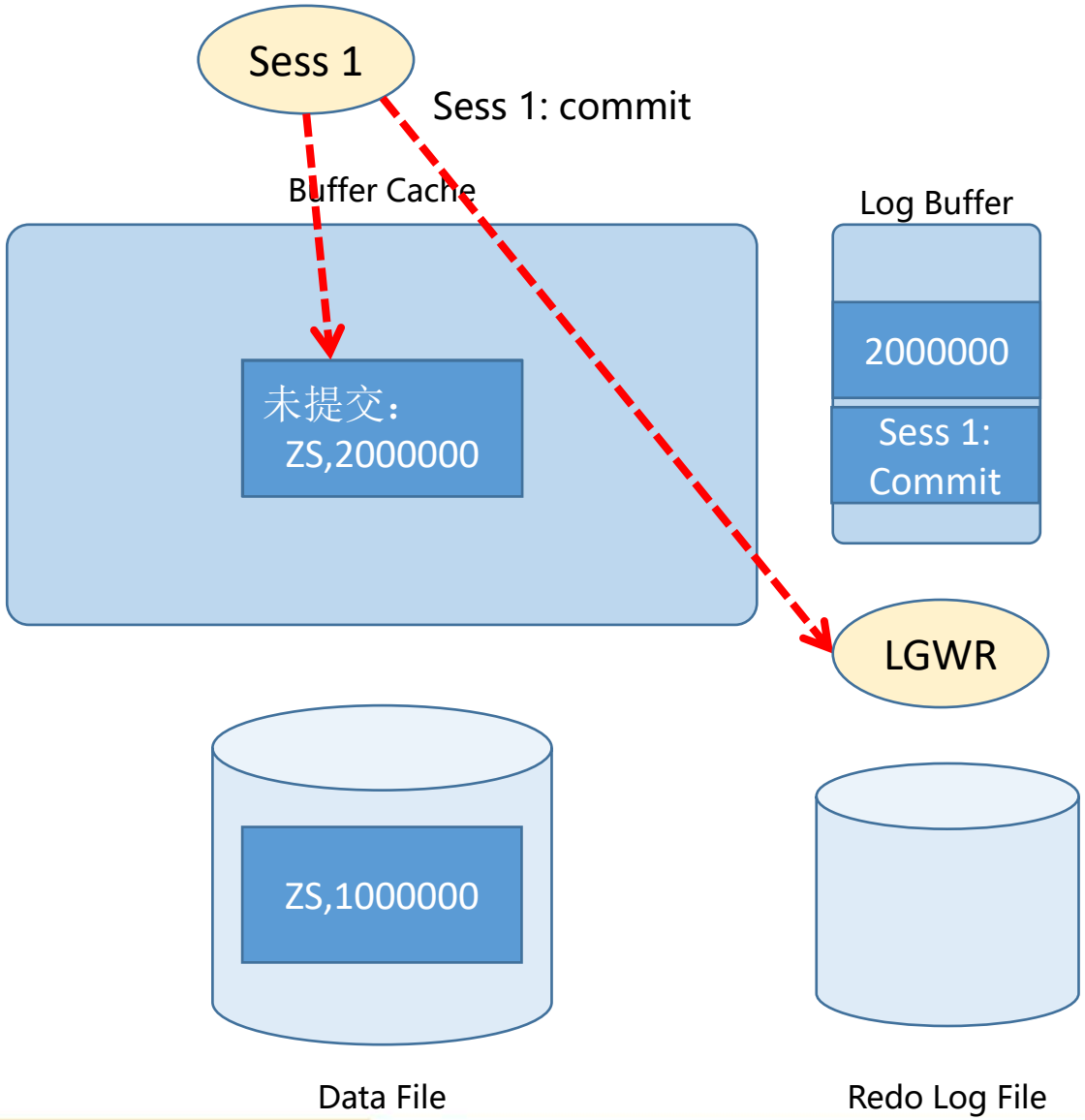
Oracle的一致性问题 -- 消失的一百万： 复盘



Oracle的一致性问题 -- 消失的一百万： 复盘



Oracle的一致性问题 -- 消失的一百万： 复盘



Oracle的一致性问题 -- 消失的一百万


```
SQL> select * from account;
        ID NAME              SUMM
---------- ---------- ----------
         1 ZS             1000000

SQL> update account set summ=2000000 where name='ZS';

已更新 1 行。

SQL> commit;
commit
     *
第 1 行出现错误:
ORA-03113: 通信通道的文件结尾 进程 ID:32730
会话 ID: 197 序列号: 6241
```

```
SQL> select * from account;
        ID NAME              SUMM
---------- ---------- ----------
         1 ZS             2000000

SQL> conn u1/a
已连接。

SQL> select * from account;
        ID NAME              SUMM
---------- ---------- ----------
         1 ZS             1000000
```

```
[root@localhost ~]# ps -ef|grep lgwr
oracle    4766    1  0 Oct26 ?        00:01:17 ora_lgwr_orcl
[root@localhost ~]# gdb -p 4766
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
```

Oracle的一致性问题 -- 消失的一百万

- BUG的种类
  - 代码BUG
  - 系统架构BUG

数据库内核技术新领域探索

数据库内核技术新领域探索：LWLockAcquire与自旋锁

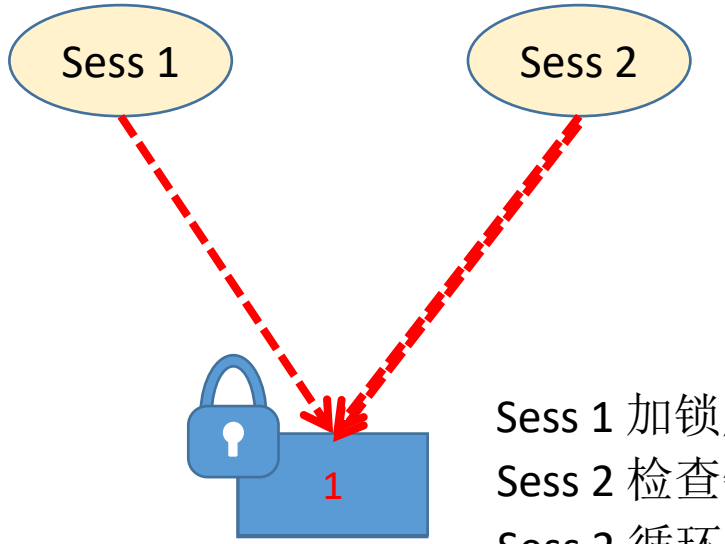
- 自旋锁

  从Lock-Free编程，到Latch/Mutex，自旋锁无处不在。

- LWLockAcquire：PG轻量级锁
  - 一次最简单的Select，最少调用它10到20次
  - 一次最简单的DML，最少调用它40次左右。
  - 在锁无法得到时，要将自己加入等待队列，此时，有可能进入自旋。

- 自旋的目的

  不自旋，无法得到锁，就要被设为Sleep状态，让出CPU。自旋可以霸占CPU，避免换出CPU后，Cache被其他进程污染。



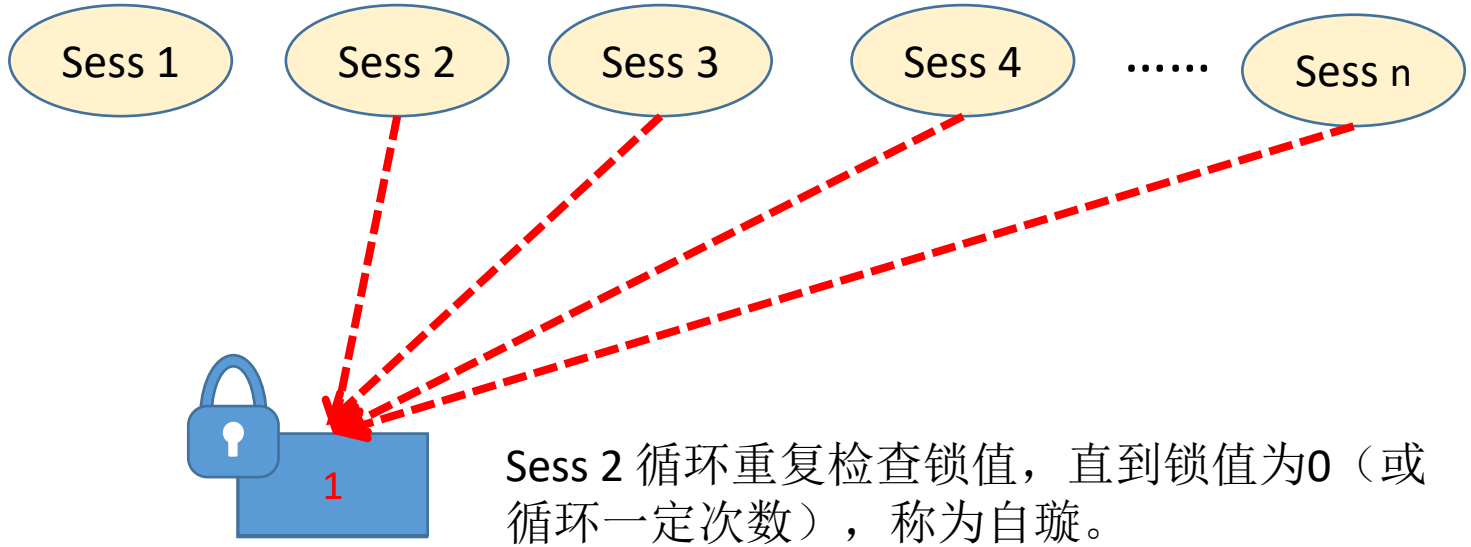
Sess 1 加锁成功
Sess 2 检查锁值，非零，锁已被其他进程持有
Sess 2 循环重复检查锁值，直到锁值为0（或循环一定次数），称为自旋。

数据库内核技术新领域探索：LWLockAcquire自璇的潜在问题

- LWLockAcquire：PG轻量级锁

  当某一锁遭遇竞争，多个进程同时自璇时，会导致一个严重问题。

  下面将视角切换到CPU。



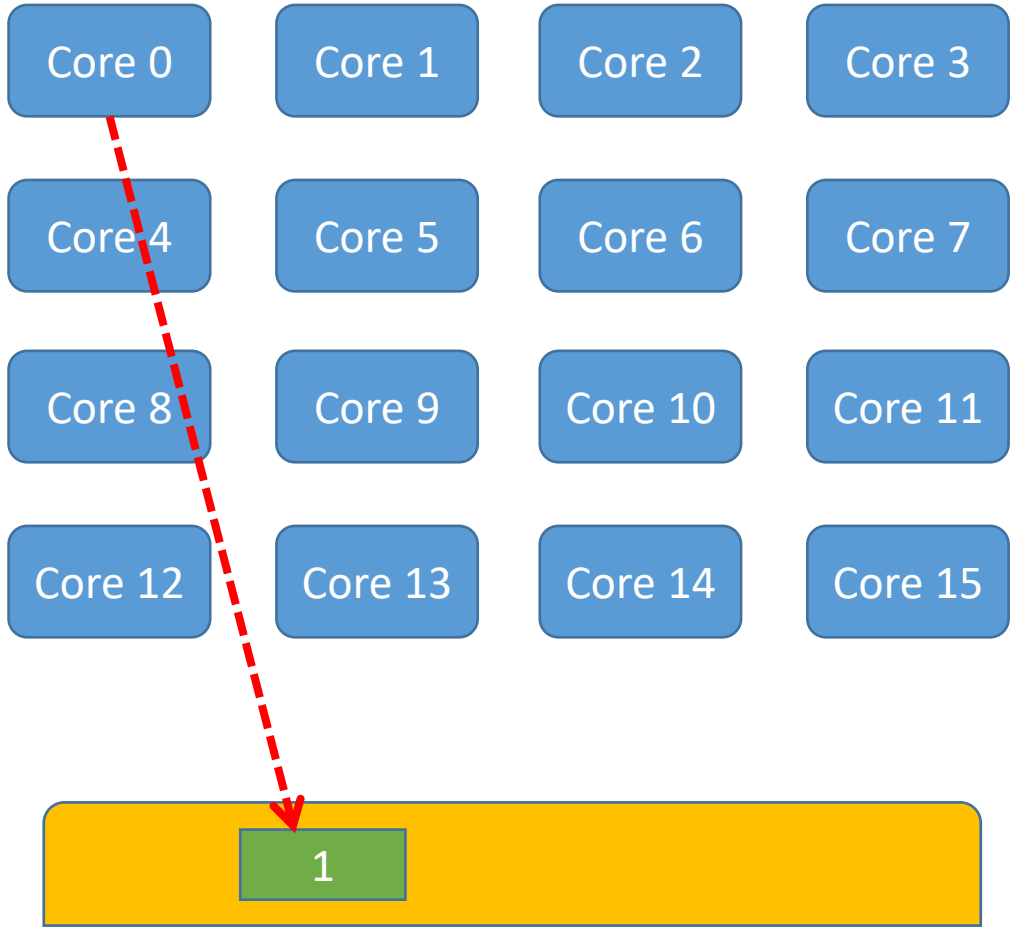
Sess 2 循环重复检查锁值，直到锁值为0（或循环一定次数），称为自璇。

数据库内核技术新领域探索：LWLockAcquire自璇的潜在问题

- 从CPU角度观察

  Core 0 持有Lock

  当Core 0 释放锁时，修改锁变量为0



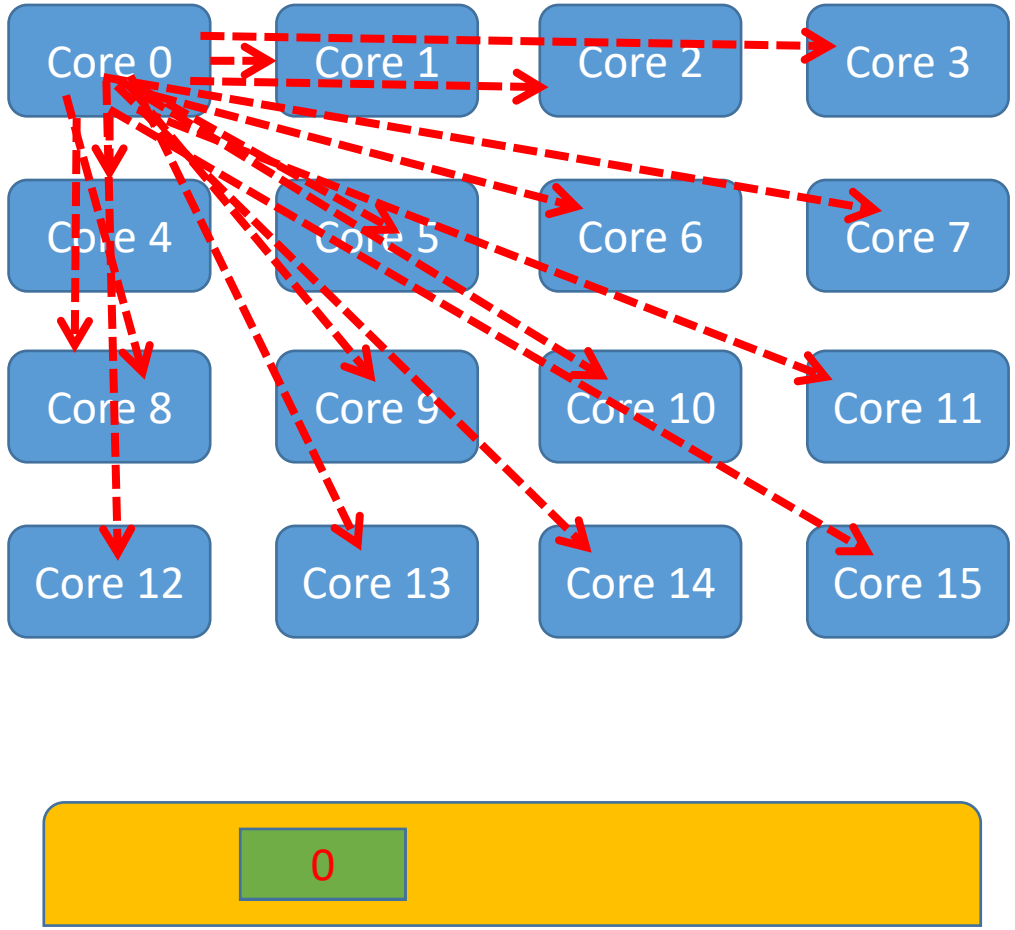
数据库内核技术新领域探索：LWLockAcquire自璇的潜在问题

- 从CPU角度观察

  Core 0 持有Lock

  当Core 0 释放锁时，修改锁变量为0

  但是另外15个Core不断在读此变量的值，Core 0要广播一个Invalite 消息给另外15个Core。之后，才能修改锁变量为0



数据库内核技术新领域探索：LWLockAcquire自璇的潜在问题

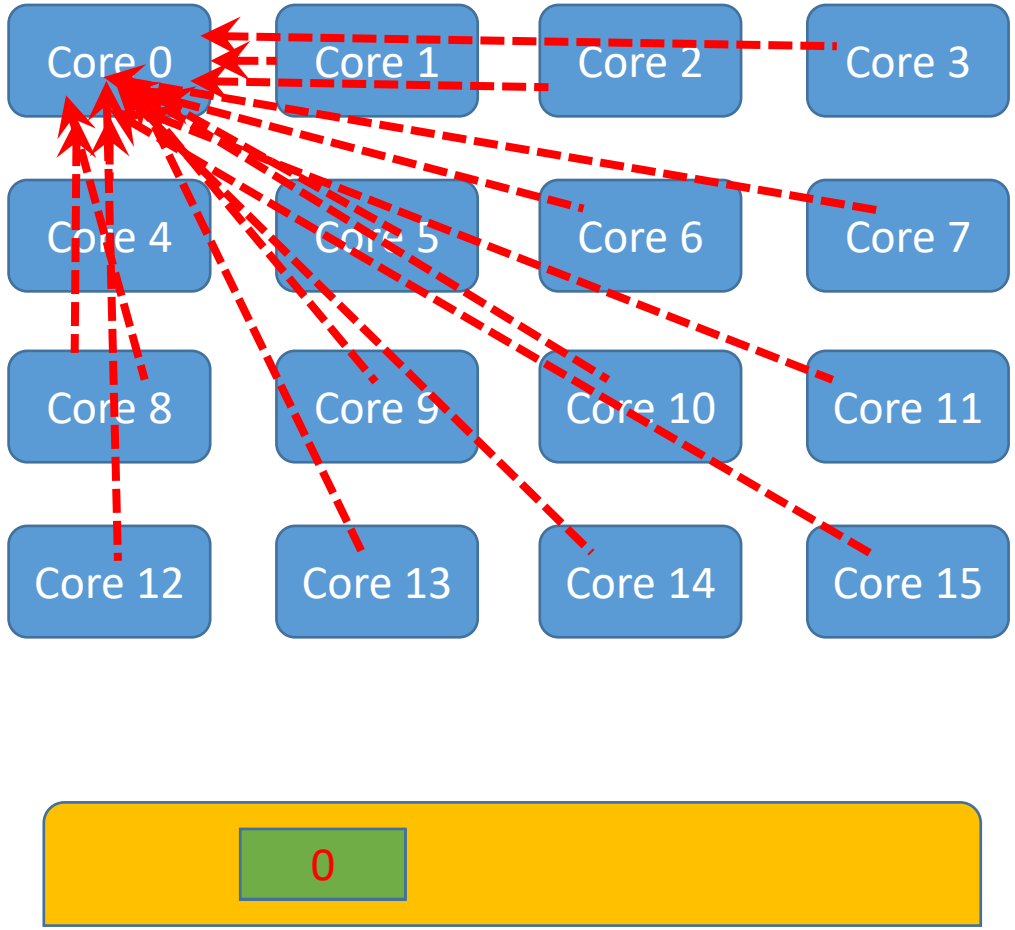
- 从CPU角度观察

Core 0 持有Lock

当Core 0 释放锁时，修改锁变量为0

但是另外15个Core不断在读此变量的值，Core 0要广播一个Invalite 消息给另外15个Core。之后，才能修改锁变量为0

另外15个核会抢着向Core 0发一个Write Update消息（即，把当前值给我，我要修改它）



数据库内核技术新领域探索：LWLockAcquire自旋的潜在问题

- 从CPU角度观察

Core 0 持有Lock

当Core 0 释放锁时，修改锁变量为0

但是另外15个Core不断在读此变量的值，Core 0要广播一个

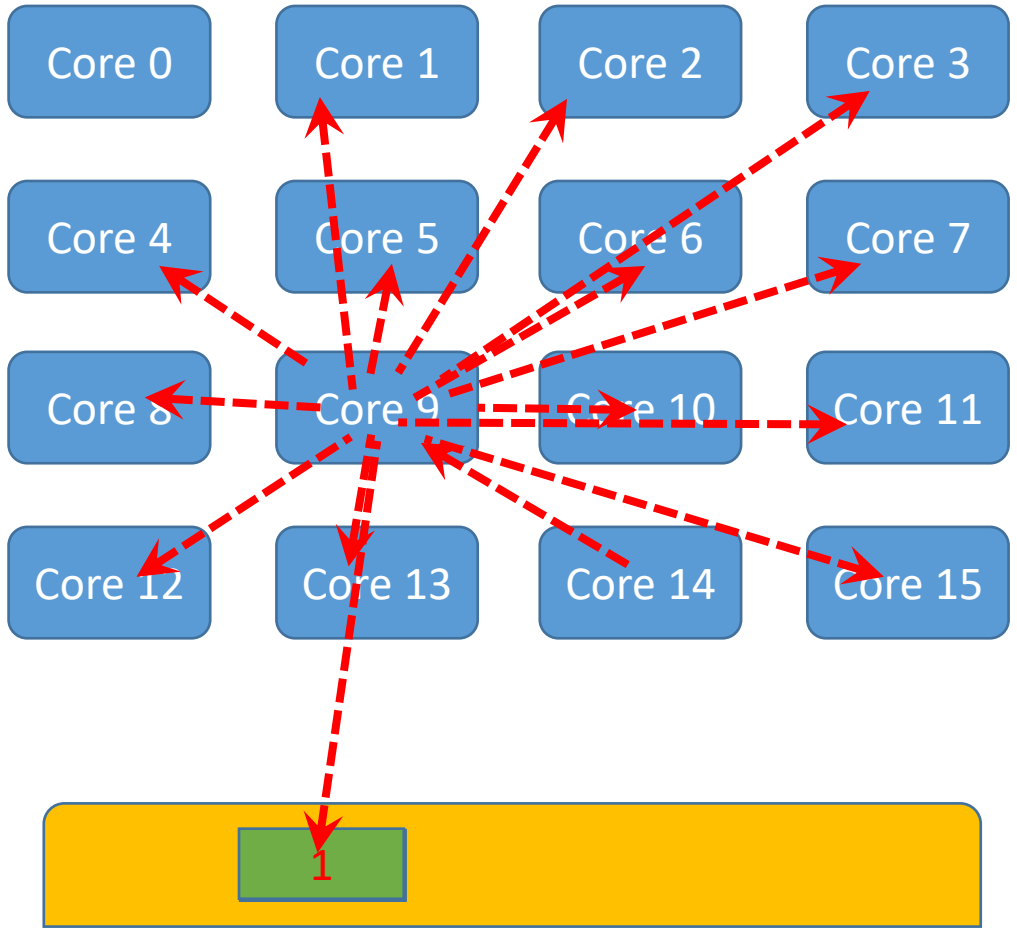
In 锁变量为0

另 e消息（即，把

当前值给我，我要修改它）

经过CPU内部的仲裁，Core 9抢到锁变量的所有权，它要向其他Core发送Write Invalidate消息。然后修改锁变量为1。

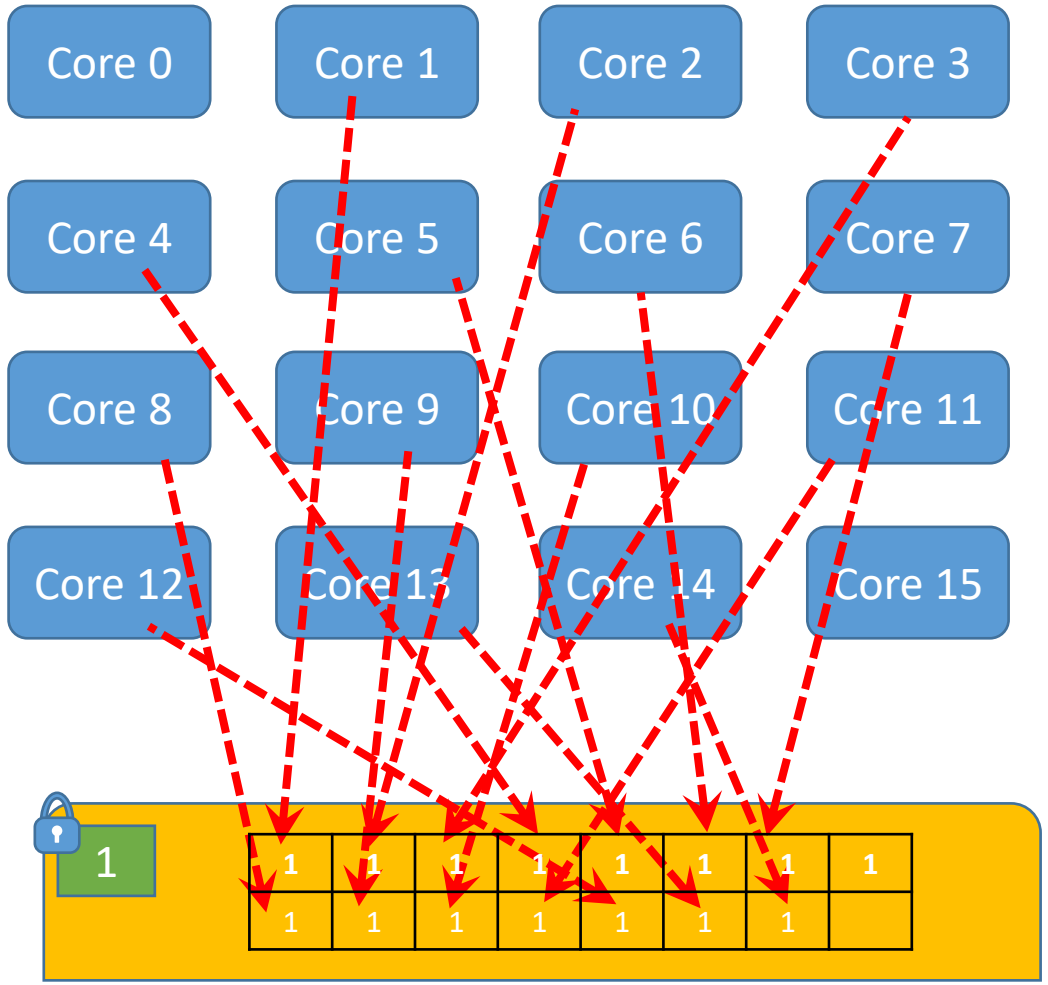
一轮轮消息同步，能将i9直接变回386。使热点竞争造成的阻塞进一步加剧。



数据库内核技术新领域探索：LWLockAcquire改进逻辑

- 改进逻辑

每个Core 自璇自个的变量，互相之间不需要同步消息



数据库内核技术新领域探索：LWLockAcquire改进逻辑

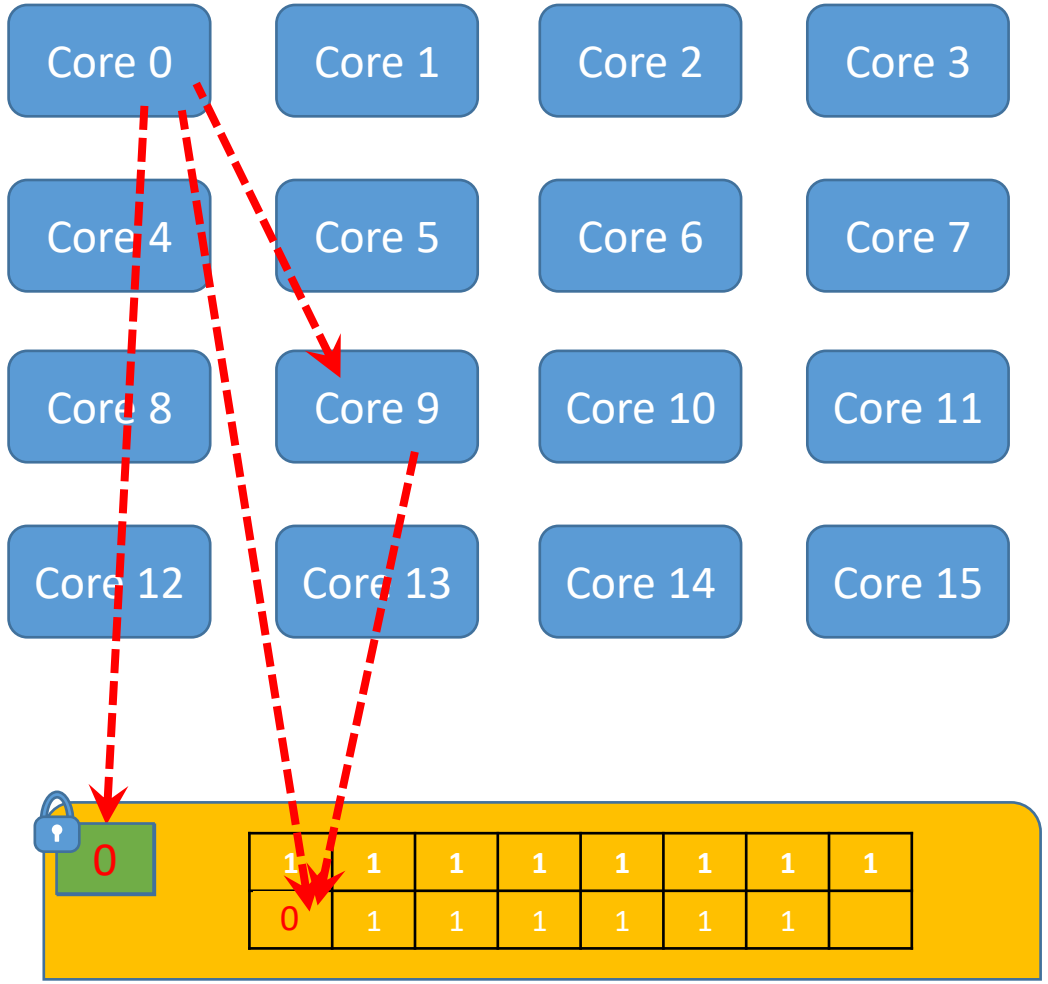
- 改进逻辑

  每个Core 自璇自个的变量，互相之间不需要同步消息

  Core 0释放锁，Core 9持有锁。Core 0只需与Core 9同步

- 相关的研究

  搜索论文《Non-scalable locks are dangerous》



数据库内核技术新领域探索：两种内核改进方式的对比

➢ Commit_id与创建快照逻辑的改进

在transam.h中定义事务数组、commit_id等共享变量
的varsup.c进行基本的初始化
在ipci.c、shmem.c中分配内存、初始化
在 的初始化
在 等操作
修

commit_id 相关patch， 在github有好些。

- 共需修改7个文件，百行级别代码
- 代码量不大，相关关系多，出意外可能性大
- 架构层面修改，BUG难以发现

➢ LWLockAcquire改进逻辑

- 只需修改 lwlock.c 一个文件
- 关联度低，意外可能性小
- 代码型BUG，测试可发现

计算机体系结构 + PostgreSQL，几乎是空白

数据库内核技术新领域探索

更多计算机体系结构 + PostgreSQL源码的应用，下会分解


openGauss
OceanBase
ArkDB
RASESQL
HotDB
StellarDB
QianBase xTP
云树Shard
GoldenDB
DolphinDB
MatrixDB
DynamoDB
SinoDB
FastData
Galaxybase
KunDB
SQL Server
vertica
D B 2
GBase
Oracle
达梦数据库
Spacture
开务数据库
UbiSQL
神舟通用
TDSQL
TiDB
Tapdata
StarRocks
KingbaseES
2010
2014
2018

THANKS