DTCC

第十一届中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2020

架构革新 自主可控

IT168.com ChinaUnix.net ITPUB

北京国际会议中心 | 2020/09/21-09/23

借鉴Oracle 深入修改MySQL/PostgreSQL内核

吕海波（VAGE)

内容介绍

- 借鉴Oracle增进PG/MYSQL内核实践：逻辑读的改进
- 事务ID的可改进之处
- 借鉴Oracle的计算与存储分离、可计算存储架构

个人介绍

吕海波

（VAGE）美创科

从业经历，十数年数据库经验，惯看IT江

网公司从事数据库管理与研究工作。2009
居库专家（P8)，并于2014年以特招方式
ebay全球唯一无法英文听说的技术人员），
乍。目前主要研究方向数据安全、数据库
acle内核技术揭密》，被誉为国内最深度
术书籍。

数据库最频繁的操作：逻辑读

Report Summary

Load Profile

DB Time(s):				
DB CPU(s):				
Redo size (bytes):				
Logical read (blocks):				
Block changes:				
Physical read (blocks):				
Physical write (blocks):				
Read IO requests:				
Write IO requests:				
Read IO (MB):				
Write IO (MB):				
User calls:				
Parses (SQL):				
Hard parses (SQL):				
SQL Work Area (MB):				
Logons:				
Executes (SQL):	57,858.2	93.2		
Rollbacks:	0.2	0.0		
Transactions:	620.7			

Redo Size是字节数，并不是次数。

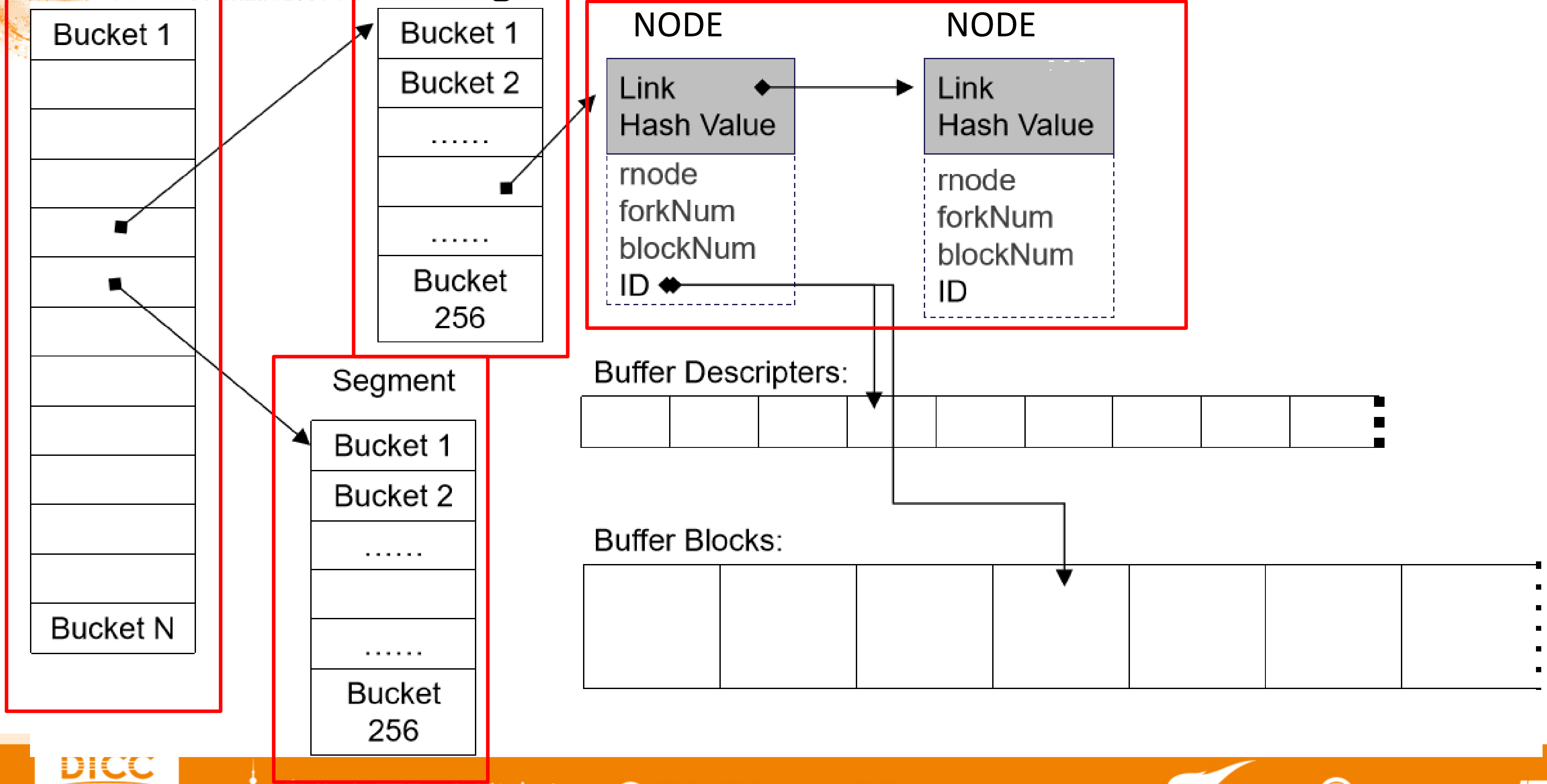
高的是logical read。

每秒超过52万次逻辑读，如果一次逻辑读可以节省1毫秒，52万次逻辑读，共可以节省520秒的CPU消耗。

性能还不是最主要的。试想，如果一个路口，每秒要经过52万辆车，万一这个路口出现事故，那怕只停一秒，就会有52万辆车拥堵。这就是竞争带来的隐患。

对于高频的操作，除性能之外，竞争，也是要关注的点。

PostgreSQL逻辑读：Shard Buffer的HASH表



PostgreSQL逻辑读：BufferAlloc()函数

```
//计算HASH值
INIT_BUFFERTAG(newTag, smgr->smgr_rnode.node, forkNum, blockNum);
newHash = BufTableHashCode(&newTag);

//根据HASH值，得到Buffer Mapping Partition Lock
newPartitionLock = BufMappingPartitionLock(newHash);

//以共享模式持有Buffer Mapping Partition Lock
LWLockAcquire(newPartitionLock, LW_SHARED);

//搜索HASH表
buf_id = BufTableLookup(&newTag, newHash);

//如果找到了，Buffer命中，进入条件。如果没找到，跳过此if语句，开始物理读
if (buf_id >= 0)
{
            …………
            return buf;
}
```

开始物理读

PostgreSQL逻辑读：BufferAlloc()函数

```
//如果找到了，进入条件。如果没找到，跳过此if语句，开始物理读
    if (buf_id >= 0)
    {
//得到Buffer Descripter结构：
        buf = GetBufferDescriptor(buf_id);
//在Buffer Descripter结构上中Pin锁：
        valid = PinBuffer(buf, strategy);
//释放Buffer Mapping Partition Lock
        LWLockRelease(newPartitionLock);
//BufferAlloc()结束，进入读取行数据模块
        return buf;
    }
//Buffer未命中，开始物理读
    LWLockRelease(newPartitionLock);
    buf = StrategyGetBuffer(strategy, &buf_state);
```

PostgreSQL逻辑读：BufferAlloc()函数

```
//Buffer未命中，开始物理读
1063      LWLockRelease(newPartitionLock);

//选择一个可覆盖页（牺牲页）：
1078      buf = StrategyGetBuffer(strategy, &buf_state);

//计算牺牲页的HASH值
1180              oldTag = buf->tag;
1181              oldHash = BufTableHashCode(&oldTag);

//计算牺牲页对应的Buffer Mapping Partition Lock
1182              oldPartitionLock = BufMappingPartitionLock(oldHash);

//加Buffer Mapping Partition Lock，独占模式。
1190                  LWLockAcquire(oldPartitionLock, LW_EXCLUSIVE);
1191                  LWLockAcquire(newPartitionLock, LW_EXCLUSIVE);

//在HASH表中，新的位置，插入一个Bucket
1221      buf_id = BufTableInsert(&newTag, newHash, buf->buf_id);
```

PostgreSQL逻辑读：BufferAlloc()函数

```
//加Buffer Mapping Partition Lock，独占模式。
                     LWLockAcquire(oldPartitionLock, LW_EXCLUSIVE);
                     LWLockAcquire(newPartitionLock, LW_EXCLUSIVE);

//在HASH表中，新的位置，插入一个Bucket
         buf_id = BufTableInsert(&newTag, newHash, buf->buf_id);

//删除牺牲页在HASH表中的Bucket
             BufTableDelete(&oldTag, oldHash);

//释放Buffer Mapping Partition Lock
         LWLockRelease(oldPartitionLock);
         LWLockRelease(newPartitionLock);
```

内容介PostgreSQL逻辑读：HASH锁

Buffer Mapping Partition Lock计算规则：

```
127 #define BufTableHashPartition(hashcode) \
128      (
```

```
129 #defi
130      (
131
```

```
112 /* Nu
113 #define NUM_BUFFER_PARTITIONS 128
```

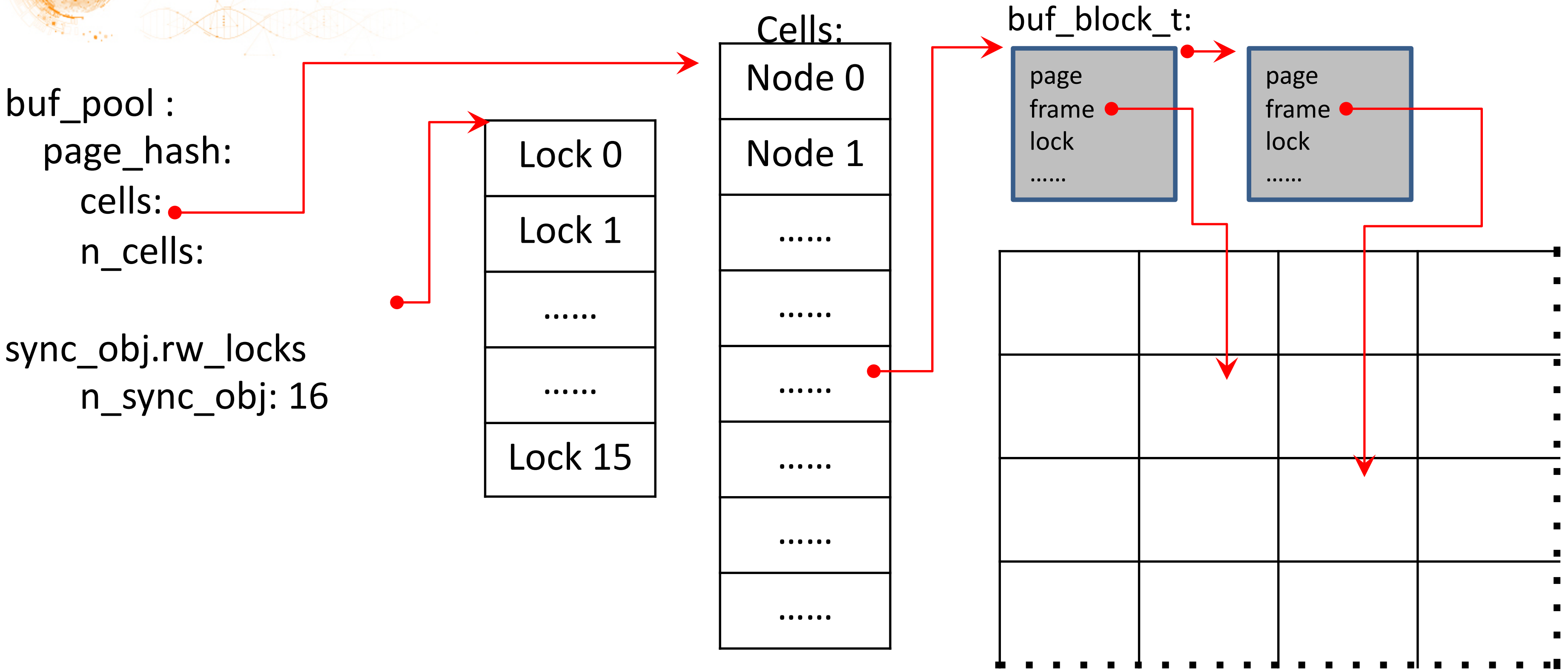
即：

MainLWLockArray[45 + HASH值 % 128].lock

HASH值 % 128代表了什么：

Buffer Mapping Partition Lock数量有128个。

MySQL逻辑读：HASH表



MySQL逻辑读简要流程

从buf_page_get_gen()开始:

```
4157     Buf_fetch_normal fetch(page_id, page_size);
4167     return (fetch.single_page());
```

进入Buf_fetch_normal::single_page()：

```
3968 template <typename T>
3969 buf_block_t *Buf_fetch<T>::single_page() {
```

```
3975     if (static_cast<T *>(this)->get(block) == DB_NOT_FOUND) {
3976       return (nullptr);
3977     }
```

MySQL逻辑读简要流程

```
dberr_t Buf_fetch_normal::get(buf_block_t *&block) {
  for (;;) {

// 计算HASH值  得到HASH表锁 搜索HASH链
    block = lookup();

// 如果HASH表中找到不相应Buffer，开始物理读
    if (block != nullptr) {
      buf_block_fix(block);    // 相当于在Buffer上加Pin

      rw_lock_s_unlock(m_hash_lock);   // 释放HASH表锁，狭义上的逻辑读到此结束
      break;
    }

// 如果逻辑读没有命中，开始物理读
    /* Page not in buf_pool: needs to be read from file */
    read_page();
  }
```

MySQL逻辑读简要流程

```
template <typename T>
buf_block_t *Buf_fetch<T>::lookup() {
```

```
// 计算HASH值，并根据HASH值得到Hash表锁
  m_hash_lock = buf_page_hash_lock_get(m_buf_pool, m_page_id);
```

```
// 加HASH表锁
  rw_lock_s_lock(m_hash_lock);
```

```
// 搜索HASH表
  if (block == nullptr) {
    block = reinterpret_cast<buf_block_t *>(
        buf_page_hash_get_low(m_buf_pool, m_page_id));
  }
```

MySQL逻辑读简要流程

// 计算HASH值，并根据HASH值得到

```
3491   m_hash_lock = buf_page_hash_lock_get(m_buf_pool, m_page_id);
```

```
2003 /** Get appropriate page_hash_lock. */
2004 #define buf_page_hash_lock_get(buf_pool, page_id) \
2005   hash_get_lock((buf_pool)->page_hash, (page_id).fold())
```

hash_get_lock(**(buf_pool)->page_hash**, (page_id).fold())

计算Fload值：

page_id_t::flod()：

```
179         m_fold = (m_space << 20) + m_space + m_page_no;
```

相当于：

m_space * 1024 * 1024 + m_space + m_page_no

MySQL逻辑读简要流程

```
// 计算HASH值，并根据HASH值得到HASH Lock：
    m_hash_lock = buf_page_hash_lock_get(m_buf_pool, m_page_id);
```

```
hash_get_lock((buf_pool)->page_hash, (page_id).fold())
```

```
hash_get_sync_obj_index(table, fold);
```

```
ut_2pow_remainder(hash_calc_hash(fold, table), table->n_sync_obj)
```

```
ut_hash_ulint(fold, table->n_cells)  // 最终的计算HASH值的函数：
```

```
        key = key ^ UT_HASH_RANDOM_MASK2;

        return (key % table_size);
```

key: 参数，就是flod。
UT_HASH_RANDOM_MASK2：宏定义，值1653893711。
table_size ：buf_pool->page_hash->n_cells，17393。

MySQL逻辑读简要流程

// 计算HASH值，并根据HASH值得到

```
3491    m_hash_lock = buf_page_hash_lock_get(m_buf_pool, m_page_id);
```

```
hash_get_lock((buf_pool)->page_hash, (page_id).fold())

hash_ge

ut_2pow

ut_2pow
  188 #
```

MySQL HASH Lock数量：

16个。

n : hash_calc_hash(fold, table)的返回值，刚计算出的HASH值

m : table->n_sync_obj，即buf_pool->page_hash->n_sync_obj，值为16。

ut_2pow_remainder的作用：Hash值 % m 。

最终，HASH锁的定位：

buf_pool->sync_obj.rw_locks + 宏ut_2pow_remainder的结果

MySQL逻辑读简要流程

物理读时的HASH表锁：

在buf_read_page_low()中：

```
bpage = buf_page_init_for_read(err, mode, page_id, page_size, unzip);
```

在buf_page_init_for_read()中：

```
mutex_enter(&buf_pool->LRU_list_mutex);

hash_lock = buf_page_hash_lock_get(buf_pool, page_id);

rw_lock_x_lock(hash_lock);

buf_page_t *watch_page;
```

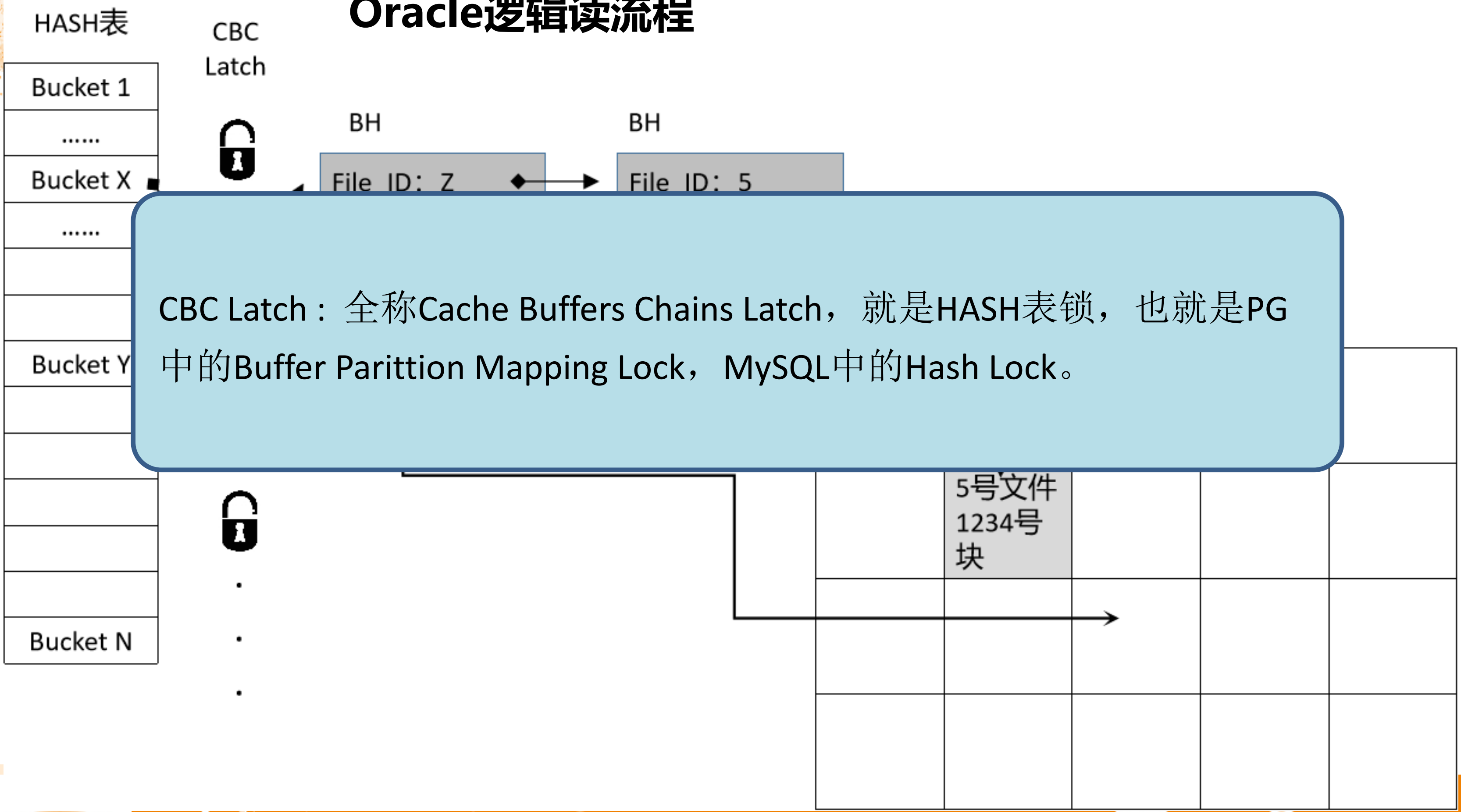
逻辑读简要流程总结

逻辑读流程总结：

① 计算HASH值
② 根据HASH值，计算并得到HASH表锁
③ 共享方式申请HASH表锁
④ 搜索HASH表
⑤ 如果找到目标Buffer
⑥ Pin住Buffer
⑦ 释放HASH表锁

如果HASH表中没有找到目标Buffer
换独占HASH表锁
物理读

Oracle逻辑读流程



CBC Latch：全称Cache Buffers Chains Latch，就是HASH表锁，也就是PG中的Buffer Parittion Mapping Lock，MySQL中的Hash Lock。

Oracle的HASH函数代码

```
int sp1=0x11;
int sp2=0x7f;
int sp3=0x7;
int hash_header=0x394953140;

hash(int p1,int p2)
{
        int uk=p1;
        int rdba=p2;

        int v1,v2;
        int target_addr;

        uk=uk<<0x11+rdba;
        uk=(0x9e370001*uk)&0x00000000FFFFFFFF;
        uk=uk>>sp1;
        v1=uk&sp2;
        v1=v1<<0x4;
        v2=uk>>0xsp3;
        v2=v2<<0x4;
        target_addr=hash_header+v2+v1;
}
```

Oracle的HASH函数代码

➢ Oracle中HASH表中Bucket的数量：计算规则，buffer数量的两倍。

➢ CBC Latc

假设有64GB的 H锁的
数量：

PostgreSQL ： 128个
MySQL ： 16*Instance个

页数量：8,3

Hash Bucket数量：16,777,216

HASH 锁数量：4,194,304

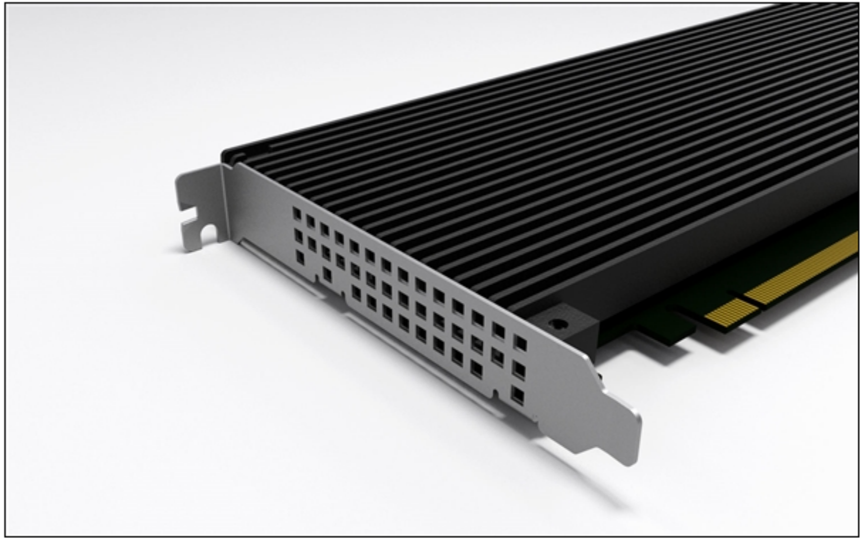
HASH锁的影响

逻
逻
物

影响的关键点：物理读。
因为物理读会造成竞争。
以前文例子中每秒52万次逻辑读为例，假设Hash Lock的数量是128，520000/128，等于4062。也就是说，每秒中会有4062次逻辑读要申请同一个Hash Lock。
相当于每秒52万辆车，不是过一个路口，是过128个路口，这当然分散了只有一个路口的竞争，但是128个路口也有点不够啊。每个路口每秒会有4062辆车。

HASH锁的影响

这款SSD配备了**四颗群联PS5016-E16主控，搭配定制固件**，才达成PCIe 4.0 x16，内部具体结构不详，估计是一个RAID。



最多可搭载32TB 3D TLC闪存，实际可用容量提供30.72TB、25.6TB、15.36TB、12.8TB、7.68TB、6.4TB等多种选择。

它支持NVMe 1.2.1，**持续读写速度都达到了恐怖的24GB/s，随机读写则是400万IOPS，稳定随机写入速度为60万IOPS，读写延迟则分别大约为80微秒、20微秒**。

HASH锁数量少，在马达驱动的机械硬盘时代，不是问题。

机械硬盘20毫秒的I/O已经可以认为是还可以的速度。

如今的硬件设备，I/O响应时间已经可以降至20微秒（提升1000倍）。

HASH锁导致的高竞争频率，在先进的硬件设备下，已经有可能成为竞争的焦点。

修改内核 减少竞争 增进性能

PostgreSQL：

```
#define BufTableHashPartition(hashcode) \
    ((hashcode) % NUM_BUFFER_PARTITIONS)

#define BufMappingPartitionLock(hashcode) \
    (&MainLWLockArray[BUFFER_MAPPING_LWLOCK_OFFSET + \
        BufTableHashPartition(hashcode)].lock)

/* Number of partitions of the shared buffer mapping hashtable*/
#define NUM_BUFFER_PARTITIONS 128
```

即：

```
MainLWLockArray[ 45 + HASH值 % 128 ].lock
```

它在src/include/storage/lwlock.h文件中

修改内核 减少竞争 增进性能

112 /* Number of ... hashtable*/

113 #define NUM_...

即：

MainLWLockArray[...

它在src/include/...

修改代码十分简单，... Lock增加了100倍。

相比对外围功能动辄... 算是修改源码。

```
[postgres@pg03 vage]$ ./vage500.sh &
[1] 13862
[postgres@pg03 vage]$
[postgres@pg03 vage]$ ./vage500.sh &
[2] 13866
[postgres@pg03 vage]$
[postgres@pg03 vage]$ ./vage500.sh &
[3] 13869
[postgres@pg03 vage]$
[postgres@pg03 vage]$ ./vage500.sh &
[4] 13883
[postgres@pg03 vage]$
[postgres@pg03 vage]$ ./vage500.sh &
[5] 13887
[postgres@pg03 vage]$
[postgres@pg03 vage]$ ERROR:  shared buffer hash table corrupted
CONTEXT:  SQL statement "EXECUTE cmtest(14988750)"
PL/pgSQL function inline_code_block line 13 at EXECUTE
ERROR:  shared buffer hash table corrupted
CONTEXT:  SQL statement "EXECUTE cmtest(1226120)"
PL/pgSQL function inline_code_block line 13 at EXECUTE

[1]   Done                    ./vage500.sh
[2]   Done                    ./vage500.sh
[postgres@pg03 vage]$
[postgres@pg03 vage]$ ERROR:  shared buffer hash table corrupted
CONTEXT:  SQL statement "EXECUTE cmtest(2594130)"
PL/pgSQL function inline_code_block line 13 at EXECUTE
ERROR:  index "vage500_pkey" contains corrupted page at block 7305
HINT:  Please REINDEX it.
CONTEXT:  SQL statement "EXECUTE cmtest(3057390)"
PL/pgSQL function inline_code_block line 13 at EXECUTE
```

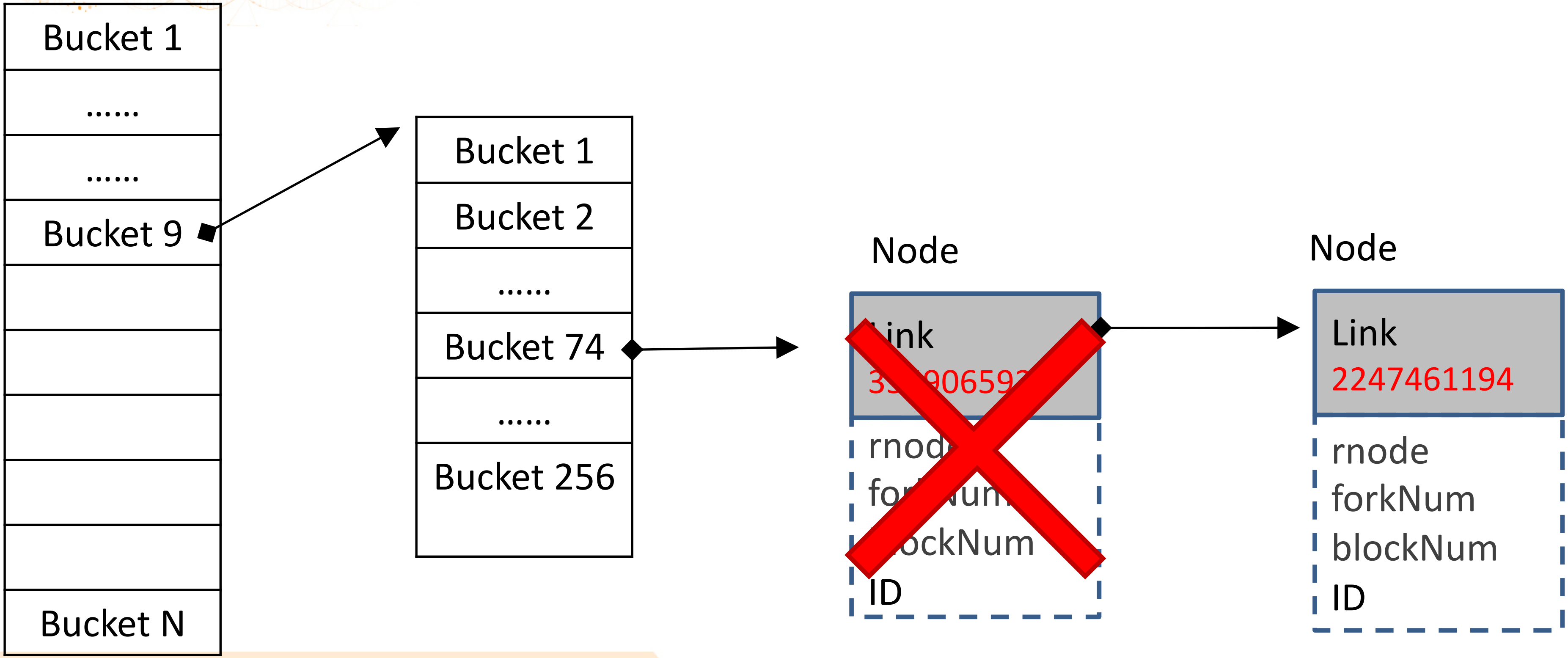
修改内核 减少竞争 增进性能

- 错误：shared buffer hash table corrupted

```
[postgres@pg03 vage]$ ERROR:  shared buffer hash table corrupted
CONTEXT:  SQL statement "EXECUTE cmtest(14988750)"
PL/pgSQL function inline_code_block line 13 at EXECUTE
ERROR:  shared buffer hash table corrupted
```

- 像宪法一样，字少，但核心，哪怕动一个字，都有可能产生难以预料的后果

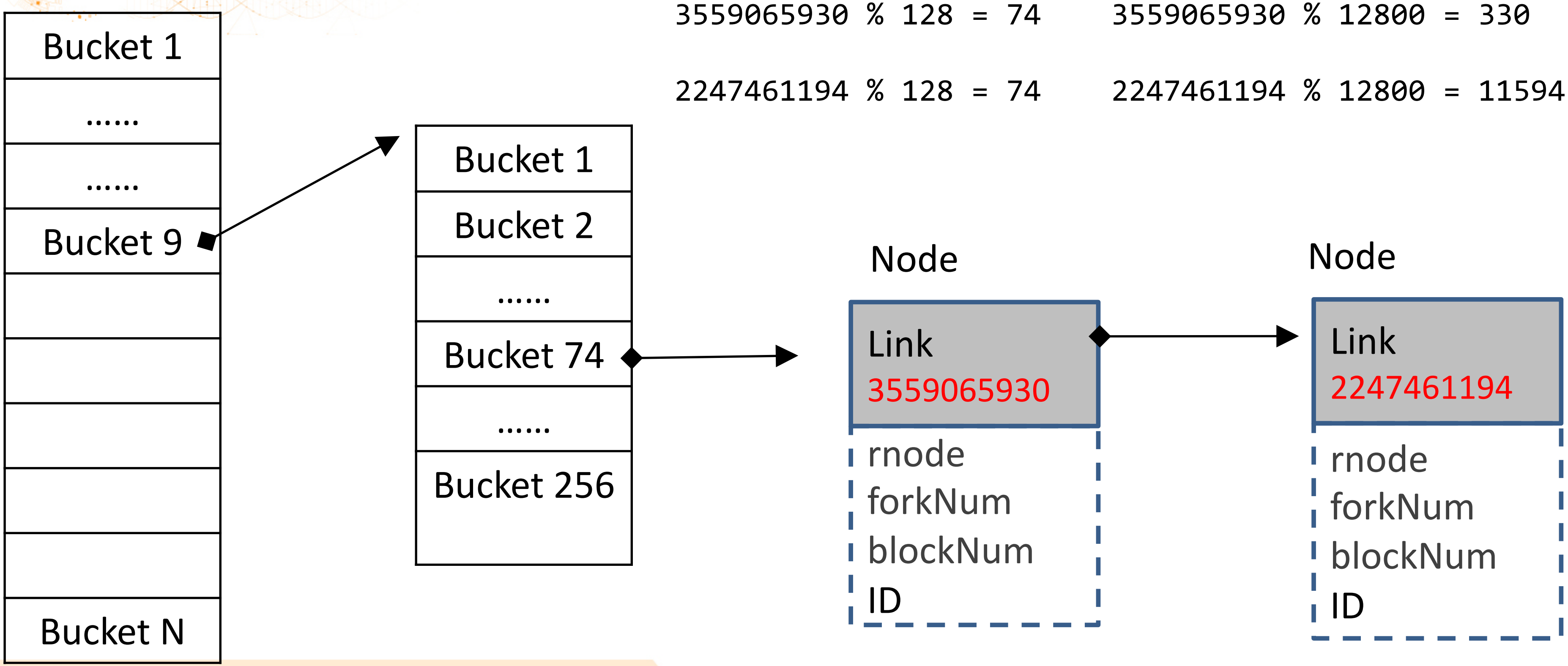
修改内核 减少竞争 增进性能



修改内核 减少竞争 增进性能

3559065930 % 128 = 74   3559065930 % 12800 = 330

2247461194 % 128 = 74   2247461194 % 12800 = 11594



修改内核 减少竞争 增进性能

```
mysql> show variables like 'innodb_page_hash_locks';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| innodb_page_hash_locks | 16    |
+-------------------------+-------+
1 row in set (0.02 sec)
```

两点限制：

- 只能增大到1024
- 只在Debug模式下，才有此参数

修改内核 减少竞争 增进性能

```
#!/bin/bash
psql lhbdb <<EOF
DO LANGUAGE plpgsql \$\$
DECLARE
    v_id1 int;
    v_id2 int;
    v_c1 varchar;
    v_c2 varchar;
    v_tmp int;
BEGIN
    execute 'PREPARE cmtest as SELECT * FROM vage500 WHERE id1 = \$1';
    for i in 1..500000 loop
        v_tmp = trunc(random()*1000000);
        v_tmp = mod(v_tmp, 224694) * 70;
        execute 'EXECUTE cmtest('||v_tmp||')' INTO v_id1, v_id2, v_c1, v_c2;
    end loop;
    deallocate prepare cmtest;
end;
\$\$;
\q
EOF
```

修改内核 减少竞争 增进性能

```
mysql> delimiter $$
mysql> CREATE PROCEDURE tmp2(in nums int, in maxid int)
    -> BEGIN
    ->   declare done int default 0;
    ->   declare i int;
    ->   declare v_id2 int;
    ->
    ->   declare cur_test CURSOR for select id2 from vage where id1=@v_id1;
    ->   declare continue handler for not found set done = 1;
    ->
    ->   set i=1;
    ->   while i<=nums DO
    ->     open cur_test;
    ->     set @v_id1 = (mod(floor(RAND() * 100000), maxid)) * 100;
    ->     fetch cur_test into v_id2;
    ->     close cur_test;
    ->     set i=i+1;
    ->   end while;
    ->   commit;
    -> END$$
delimiter ;
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;
mysql>
```

修改内核 减少竞争 增进性能

使用的动态内核跟踪语言：systemtap，统计脚本comp2.stp，局部代码：

统计逻辑读次数：

```
probe process("postgres").function("FileRead") {
    if ($amount == 8192)
        pread_num <<< 1
}
```

统计逻辑读时间：

```
probe process("postgres").function("BufferAlloc") {
    lgr[pid(), "BufferAlloc"] = 1
    tm1[pid()] = gettimeofday_us()
}

probe process("postgres").function("ReadBuffer_common").return {
    lgr[pid(), "BufferAlloc"] = 0
    lgr[pid(), "LWLockAcquire"] = 0
    lgr[pid(), "StrategyGetBuffer"] = 0
    lnum <<< 1
    tm2 <<< gettimeofday_us() - tm1[pid()]
}
```

修改内核 减少竞争 增进性能

succ	fail	succ_p	fail_p	count	avg	max	pysical	
105290	3	38674	2	66616	23	6167	12891	0
105325	0	38946	0	66379	23	13040	12982	0
105336	2	38790	0	66544	24	3306	12930	0
105860	1	39385	0	66474	23	8809	13128	0
105016	3	38764	0	66252	22	9239	12922	0
105484	0	39017	0	66465	23	4410	13006	0
104915	3	38705	1	66211	22	5659	12901	0

succ : 成功获得Buffer Mapping Partition Lock次数

fail : 申请Buffer Mapping Partition Lock失败，遇到阻塞次数

succ_p和fail_p，是物理读时成功获得锁的次数，和遇到阻塞次数

count: 逻辑读次数

avg : 逻辑读平均响应时间

max : 逻辑读最长响应时间

pysical : 物理读次数

无标题的列：Buffer Mapping Partition Lock申请失败的次数和物理读的比值。如果此列值为20，说明100次物理读，会引发20次锁竞争。

修改内核 减少竞争 增进性能

修改前：

succ	fail	succ_p	fail_p	count	avg	max	pysical	
484169	18397	276414	7726	207760	629	119145	92384	19
162388	5670	92730	2384	69658	613	107959	30982	18
162444	5425	92787	2286	69643	620	104457	31009	17
162062	5659	92473	2496	69597	618	132470	30896	18
162616	5718	92849	2447	69766	619	85938	31020	18
161458	5843	92100	2443	69352	615	82432	30780	18
161608	6322	92208	2735	69403	635	108372	30806	20
161364	6465	92124	2649	69243	630	76704	30781	21
323604	12413	185039	5096	138562	630	101626	61840	20
325058	11022	185739	4567	139315	609	102844	62084	17
323945	11678	184980	4884	138967	625	106728	61808	18
161377	6330	91983	2596	69397	629	75366	30737	20

修改后：

succ	fail	succ_p	fail_p	count	avg	max	pysical	
497119	65	284382	24	212738	211	40077	94792	0
166327	30	95035	8	71291	207	88741	31676	0
166300	16	95103	12	71200	212	36303	31703	0
165999	16	94717	10	71282	214	37906	31571	0
165239	24	94428	18	70808	207	79506	31476	0
166112	27	95005	17	71108	217	40028	31667	0
165989	19	94909	7	71079	215	59308	31640	0
333154	67	190408	33	142745	212	82777	63467	0
498201	92	284629	39	213560	216	114525	94868	0
165333	34	94357	19	70980	218	45581	31456	0
164900	21	94095	13	70805	234	47079	31364	0
164448	27	93693	11	70752	250	45029	31231	0
491537	128	280454	60	211079	273	107906	93482	0

修改内核 减少竞争 增进性能

succ	fail	count	avg	max	pysical	
8195	3	2723	33	5933	446	0
9183	1	3060	31	5824	524	0
10050	1	3350	31	6197	568	0
9661	2	3216	32	6136	547	0
9829	1	3269	45	27854	551	0
9078	0	3026	66	112548	501	0
9823	4	3270	33	6111	542	0
18741	1	6248	103	66216	1051	0
10333	0	3437	33	5834	579	0
9333	0	3112	29	207	511	0
10039	0	3341	35	6062	557	0
9877	5	3286	32	5822	534	0
8841	2	2947	28	168	489	0
10456	0	3481	35	6142	580	0
9766	0	3249	35	10559	540	0

慢速存储设备上的测试，每秒物理读次数较低，HASH锁数量多与少，基本不影响竞争。

修改内核 减少竞争 增进性能

修改前：

procs		-----------memory----------				---swap--		-----io----		-system--		------cpu-----				
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st
52	0	0	414204	2324	14890232	0	0	0	121	5039	8209	88	11	1	0	0
18	0	0	414576	2324	14890240	0	0	0	20	4873	7589	88	12	1	0	0
49	0	0	414576	2324	14890244	0	0	0	0	5000	7955	87	11	2	0	0
19	0	0	414444	2324	14890252	0	0	0	12	4954	8426	88	11	1	0	0
20	0	0	414656	2324	14890260	0	0	0	20	4993	7365	88	11	2	0	0
30	0	0	414708	2324	14890268	0	0	0	0	4902	7429	89	11	0	0	0
38	0	0	414568	2324	14890276	0	0	0	4	5017	8563	89	11	1	0	0
26	0	0	413792	2324	14890284	0	0	0	20	5030	7085	90	10	1	0	0
33	0	0	413792	2324	14890284	0	0	0	80	4962	7599	88	11	1	0	0
17	0	0	413992	2324	14890292	0	0	0	4	4937	7645	88	11	1	0	0

修改后：

procs		-----------memory----------				---swap--		-----io----		-system--		------cpu-----				
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st
45	0	0	723688	2324	14616552	0	0	0	0	4758	3283	89	11	0	0	0
42	0	0	723580	2324	14616560	0	0	0	16	4954	3714	90	10	0	0	0
49	0	0	723440	2324	14616564	0	0	0	144	4886	3448	89	11	0	0	0
50	0	0	723440	2324	14616572	0	0	0	0	4942	3648	89	11	0	0	0
43	0	0	722456	2324	14616580	0	0	0	16	5228	4214	89	11	0	0	0
50	0	0	722448	2324	14616584	0	0	0	0	4967	3582	89	11	0	0	0
49	0	0	722448	2324	14616592	0	0	0	0	4917	3562	90	11	0	0	0
44	0	0	722448	2324	14616600	0	0	0	40	4981	3584	89	11	0	0	0
49	0	0	722400	2324	14616604	0	0	0	0	4559	2837	89	11	0	0	0
47	0	0	722556	2324	14616608	0	0	0	212	4927	3508	90	10	0	0	0

修改内核 减少竞争 增进性能

修改前：

```
[postgres@pg03 vage]$ DO
Time: 250371.404 ms (04:10.371)
DO
Time: 271858.590 ms (04:31.859)
DO
Time: 285658.549 ms (04:45.659)
DO
Time: 286958.456 ms (04:46.958)
DO
Time: 300658.486 ms (05:00.658)
DO
Time: 306394.310 ms (05:06.394)
DO
Time: 309908.208 ms (05:09.908)
DO
Time: 312063.334 ms (05:12.063)
DO
Time: 316887.343 ms (05:16.887)
DO
Time: 318885.939 ms (05:18.886)
DO
Time: 321996.946 ms (05:21.997)
DO
Time: 322731.673 ms (05:22.732)
DO
Time: 324172.719 ms (05:24.173)
DO
Time: 325137.246 ms (05:25.137)
DO
Time: 324711.256 ms (05:24.711)
```

修改后：

```
DO
Time: 296692.228 ms (04:56.692)
DO
Time: 293774.225 ms (04:53.774)
DO
Time: 296882.952 ms (04:56.883)
DO
Time: 272215.597 ms (04:32.216)
DO
Time: 273588.100 ms (04:33.588)
DO
Time: 276389.954 ms (04:36.390)
DO
Time: 271610.350 ms (04:31.610)
DO
Time: 280288.620 ms (04:40.289)
DO
Time: 274780.265 ms (04:34.780)
DO
Time: 277446.487 ms (04:37.446)
DO
Time: 275582.488 ms (04:35.582)
DO
Time: 270736.363 ms (04:30.736)
DO
Time: 276453.020 ms (04:36.453)
DO
Time: 266146.205 ms (04:26.146)
DO
```

修改内核 减少竞争 增进性能

```
lhbdb=# select
lhbdb-# 294226.330+299789.626+301783.643+302925.730+300864.651+302550.192+304912.751+302782.096+
lhbdb-# 303246.718+304649.969+304008.474+303388.819+300223.468+301685.621+303560.112+302668.818+
lhbdb-# 299978.168+300873.507+299946.340+297984.390+299439.174+298763.509+297242.642+294560.966+
lhbdb-# 296555.417+297964.053+294479.271+296692.228+293774.225+296882.952+272215.597+273588.100+
lhbdb-# 276389.954+271610.350+280288.620+274780.265+277446.487+275582.488+270736.363+276453.020+
lhbdb-# 266146.205+269513.771+268284.053+278576.954+267644.473+270341.267+267041.403+265716.714+
lhbdb-# 269129.796+274090.979;
   ?column?

 14443980.719
(1 row)

lhbdb=# select
lhbdb-# 250371.404+271858.590+285658.549+286958.456+300658.486+306394.310+309908.208+312063.334+
lhbdb-# 316887.343+318885.939+321996.946+322731.673+324172.719+325137.246+324711.256+327661.980+
lhbdb-# 327801.976+328286.583+328976.795+328715.136+329140.937+331137.627+331094.706+330125.975+
lhbdb-# 330816.635+330182.452+330960.899+329600.964+328297.804+330468.041+331182.692+329351.808+
lhbdb-# 327880.709+328042.939+326860.727+327593.289+330357.912+327085.323+325298.627+326812.465+
lhbdb-# 326140.221+320933.769+323817.697+324875.784+323378.696+320673.814+322758.362+322111.217+
lhbdb-# 320064.312+319419.960;
   ?column?
--------------
 16026303.292
(1 row)
```

修改前总耗时：16,026,303.292毫秒
修改后总耗时：14,443,980.719毫秒

MySQL的事务ID获取机制

```
UNIV_INLINE
trx_id_t trx_sys_get_new_trx_id() {
  ut_ad(trx_sys_mutex_own());

  /* VERY important: after the database is started, max_trx_id value is
  divisible by TRX_SYS_TRX_ID_WRITE_MARGIN, and the following if
  will evaluate to TRUE when this function is first time called,
  and the value for trx id will be written to disk-based header!
  Thus trx id values will not overlap when the database is
  repeatedly started! */

  if (!(trx_sys->max_trx_id % TRX_SYS_TRX_ID_WRITE_MARGIN)) {
    trx_sys_flush_max_trx_id();
  }

  return (trx_sys->max_trx_id++);
}
```

MySQL的事务ID获取机制

Select操作，在trx_start_low()中调用trx_sys_get_new_trx_id（ ）：

```
trx_sys_mutex_enter();

 trx->id = trx_sys_get_new_trx_id();
```

DML操作，在trx_set_rw_mode()中调用trx_sys_get_new_trx_id()：

```
mutex_enter(&trx_sys->mutex);

ut_ad(trx->id == 0);
trx->id = trx_sys_get_new_trx_id();
```

PostgreSQL事务ID的获取

只读事务（Select）不会增加事务ID

读写事务增加事务ID的方式：

GetNewTransactionId()：

```
LWLockAcquire(XidGenLock, LW_EXCLUSIVE);   //以独占方式，得到全局的XidGenLock锁

full_xid = ShmemVariableCache->nextFullXid;  //从ShmemVariableCache->nextFullXid中，得到XID（即事务ID）
xid = XidFromFullTransactionId(full_xid);
```

```
#define XidFromFullTransactionId(x)      ((uint32) (x).value)
```

PostgreSQL事务ID的获取

增加ShmemVariableCache->nextFullXid：

```
FullTransactionIdAdvance(&ShmemVariableCache->nextFullXid);
```

```
static inline void
FullTransactionIdAdvance(FullTransactionId *dest)
{
    dest->value++;
    while (XidFromFullTransactionId(*dest) < FirstNormalTransactionId)
        dest->value++;
}
```

释放XidGenLock锁

```
LWLockRelease(XidGenLock);
```

PostgreSQL事务ID竞争测试

insert1.sh

```
psql lhbdb <<EOF
\timing

DO LANGUAGE plpgsql \$\$
DECLARE
    v_id1 int;
    v_tmp int;
    v_c1 varchar;
    v_sql varchar;
BEGIN
    execute 'PREPARE cmtest as insert into t9 values(\$1, \$2, \$3, \$4)';
    for i in 1..123456 loop
        v_c1 = 'aaaa'||i;
        v_sql = 'EXECUTE cmtest('||i||', '||i||', ''aaaa'||v_c1||''', '''||v_c1||''')';
        execute  v_sql;
        commit;
    end loop;
    deallocate prepare cmtest;
end;
\$\$;
\q
EOF
```

PostgreSQL事务ID竞争测试

```
lhbdb=# select pid, wait_event_type, wait_event, state, backend_type from pg_stat_activity ;
  pid  | wait_event_type |      wait_event        | state  |        backend_type
-------+-----------------+------------------------+--------+------------------------------
 15648 | Activity        | AutoVacuumMain         |        | autovacuum launcher
 15650 | Activity        | LogicalLauncherMain    |        | logical replication launcher
  9307 |                 |                        | active | client backend
 15470 |                 |                        | active | client backend
 15473 | LWLock          | XidGenLock             | active | client backend
 15476 | LWLock          | XidGenLock             | active | client backend
 15479 | LWLock          | XidGenLock             | active | client backend
 15482 | LWLock          | XidGenLock             | active | client backend
 15485 |                 |                        | active | client backend
 15488 | IPC             | ProcArrayGroupUpdate   | active | client backend
 15491 | LWLock          | XidGenLock             | active | client backend
 15494 | LWLock          | XidGenLock             | active | client backend
 15497 | LWLock          | XidGenLock             | active | client backend
 15501 | LWLock          | XidGenLock             | active | client backend
 15504 | LWLock          | XidGenLock             | active | client backend
 15510 | LWLock          | XidGenLock             | active | client backend
 15509 | LWLock          | XidGenLock             | active | client backend
 15513 | LWLock          | XidGenLock             | active | client backend
 15515 | LWLock          | XidGenLock             | active | client backend
 15523 | LWLock          | XidGenLock             | active | client backend
 15522 | LWLock          | XidGenLock             | active | client backend
 15524 | LWLock          | XidGenLock             | active | client backend
 15527 | LWLock          | XidGenLock             | active | client backend
 15533 | LWLock          | XidGenLock             | active | client backend
 15536 | LWLock          | XidGenLock             | active | client backend
 15537 | IPC             | ProcArrayGroupUpdate   | active | client backend
 15545 | LWLock          | XidGenLock             | active | client backend
 15547 | LWLock          | XidGenLock             | active | client backend
 15548 | LWLock          | XidGenLock             | active | client backend
 15554 | LWLock          | XidGenLock             | active | client backend
 15552 | LWLock          | XidGenLock             | active | client backend
```

Oracle的事务ID（XID）策略

- MySQL/PostgreSQL：

事务ID或XID即是事务的唯一标志，又代表事务的先后顺序

- Oracle：

事务ID（即Oracle中的XID）仅仅是事务的唯一标志，并不代表先后顺序

SCN，代表先后事务顺序。

Oracle的事务ID策略

- Oracle的XID：

类似UUID，不连续的全局标识，由事务的回滚段编号、事务槽号、和序列号构成。

获得过程无需任何锁。

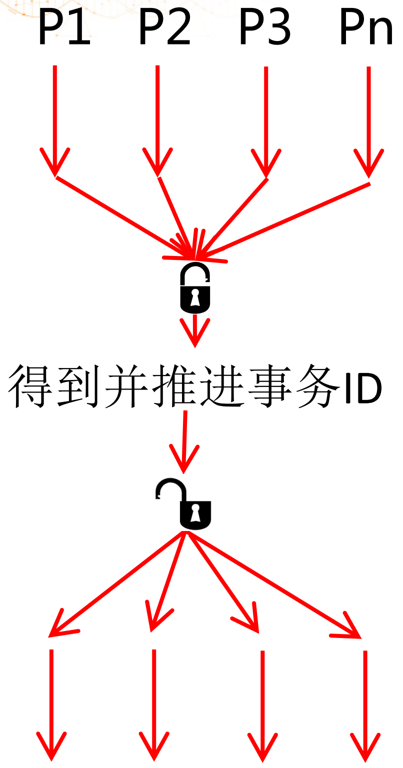
- SCN：

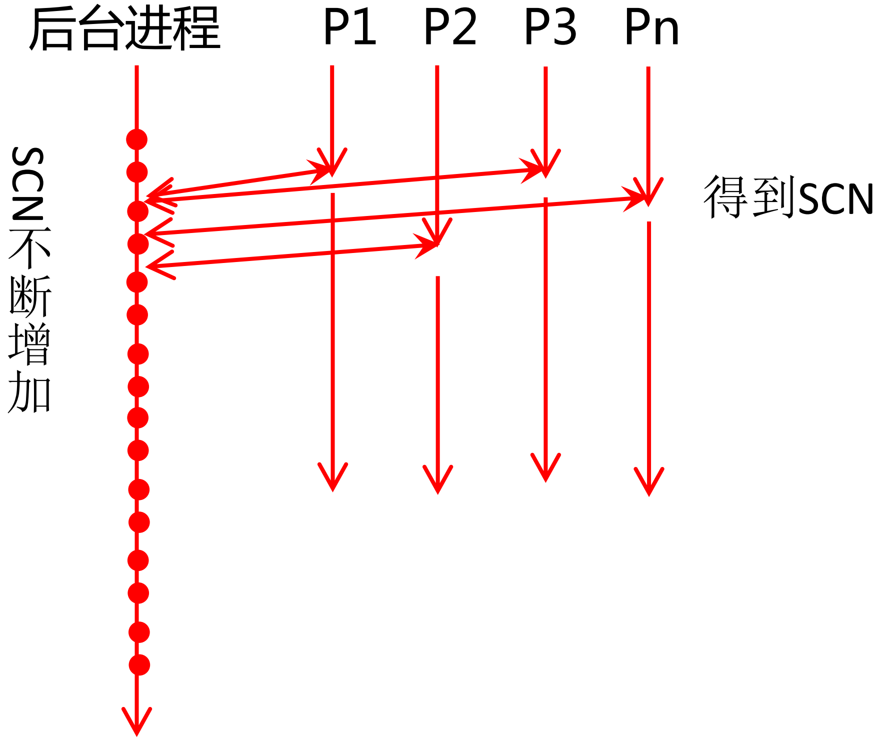
由核心后台进程（LGWR、CKPT、PMON、SMON、DBWR）维持自增的全局变量，在自增时需持有独占的SCN锁。

用户进程的Select和DML操作，只读取SCN，不持有SCN锁（只在读取失败时才会持有独占锁）

Commit操作，尝试进行用原子操作推进SCN，如果失败，自动放弃修改。不持有SCN锁。

事务ID策略对比





Oracle的事务ID策略

```
> $r
%rax = 0x0000000000000000        %r8  = 0x000000000cee53e0
%rbx = 0x0000000000000000        %r9  = 0xfffffd7fffdfc7b0
%rcx = 0x0000000000000082        %r10 = 0x0000000000000000
%rdx = 0xfffffd7fffdfc7e8        %r11 = 0xfffffd7ffcd0473a
%rsi = 0xfffffd7fffdfb628        %r12 = 0xfffffd7fffdfb730
%rdi = 0x0000000380017d70        %r13 = 0xfffffd7fffdfb730
                                 %r14 = 0x0000000000000000
                                 %r15 = 0x0000000000000020
```

Shared Pool

```
> 0x0000000380017d70,20::dump -e
380017d70:  00692705 00000000 00000000 00000000
380017d80:  000000b7 00000000 00000000 00000000
>
> 0x0000000380017d70,20::dump -e
380017d70:  00692707 00000000 00000000 00000000
380017d80:  000000b8 00000000 00000000 00000000
```

```
SQL> select CURRENT_SCN, to_char(CURRENT_SCN, 'xxxxxxx') from v$database;

 CURRENT_SCN TO_CHAR(
------------ --------
     6891270   692706
```

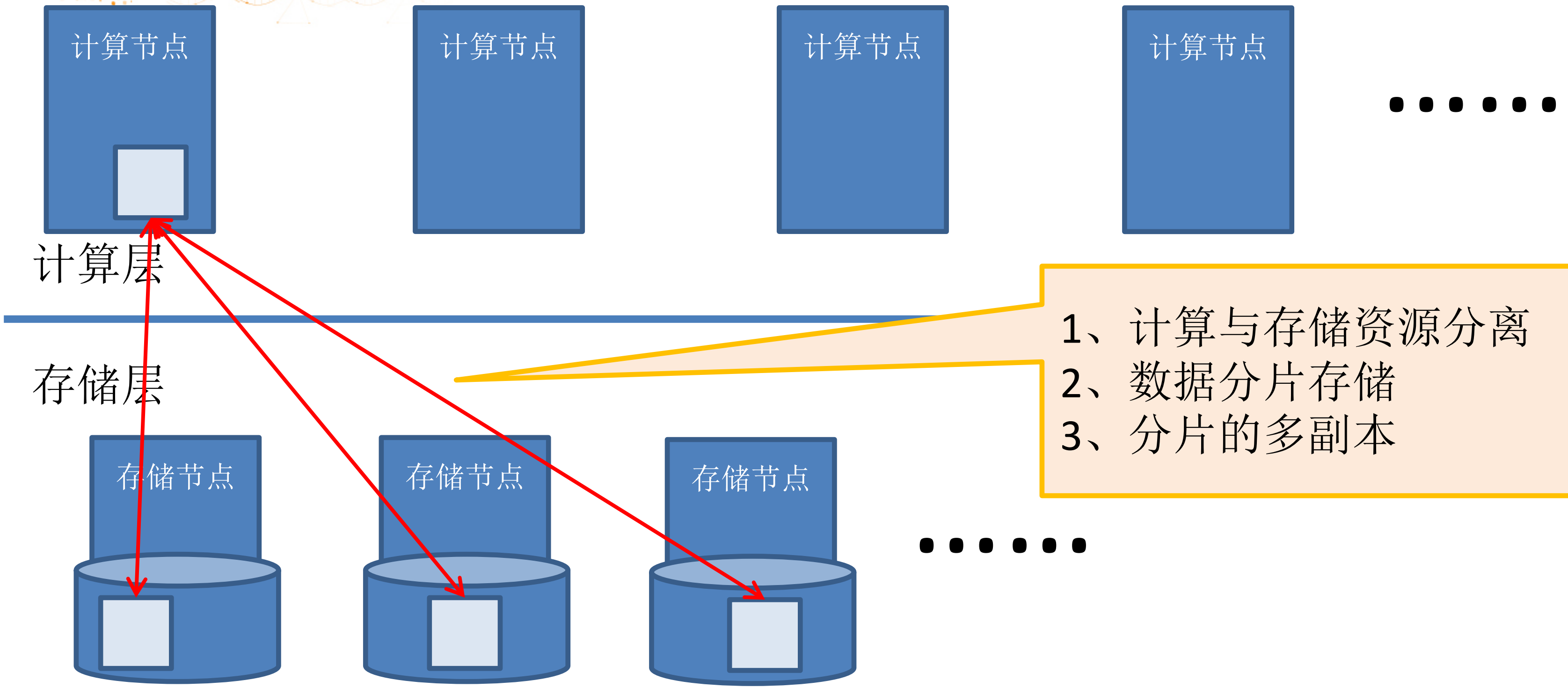
读取计时器的值：
0x00b7，到局部变量

读取SCN值：0x00692705

再次读取计时器，和之前读到的计时器作比较，如相等，以刚才读取得到SCN值0x00692705为结果

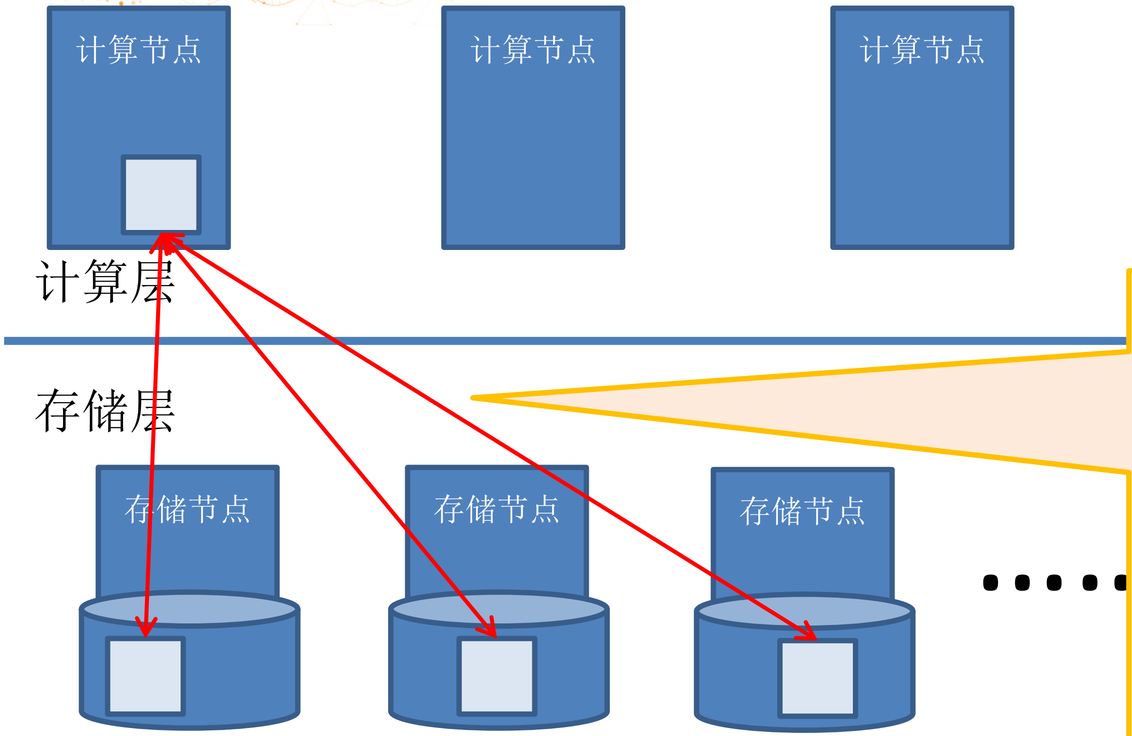
```
pushq   %rbp
movq    %rsp,%rbp
pushq   %r12
pushq   %r13
pushq   %r14
subq    $0x18,%rsp
movq    %rdi,%r14
movq    %rsi,%r13
movl    0x10(%r14),%eax
movl    %eax,-0x24(%rbp)
movq    $0x1,%rax
testl   %eax,%eax
je      +0xce       <kcscur3+0xf2>
movl    0x14(%r14),%eax
testl   %eax,%eax
jne     +0x41       <kcscur3+0x71>
movq    (%r14),%r12
movq    $0x1,%rax
testl   %eax,%eax
je      +0xbe       <kcscur3+0xfc>
movl    -0x24(%rbp),%eax
movl    0x10(%r14),%ecx
cmpl    %ecx,%eax
jne     +0x63       <kcscur3+0xb0>
movl    $-0x1,%r8d
andq    %r12,%r8
movl    %r8d,0x0(%r13)
shrq    $0x20,%r12
movw    %r12w,0x4(%r13)
addq    $0x18,%rsp
popq    %r14
popq    %r13
popq    %r12
leave
ret
```

计算与存储分离、可计算存储架构



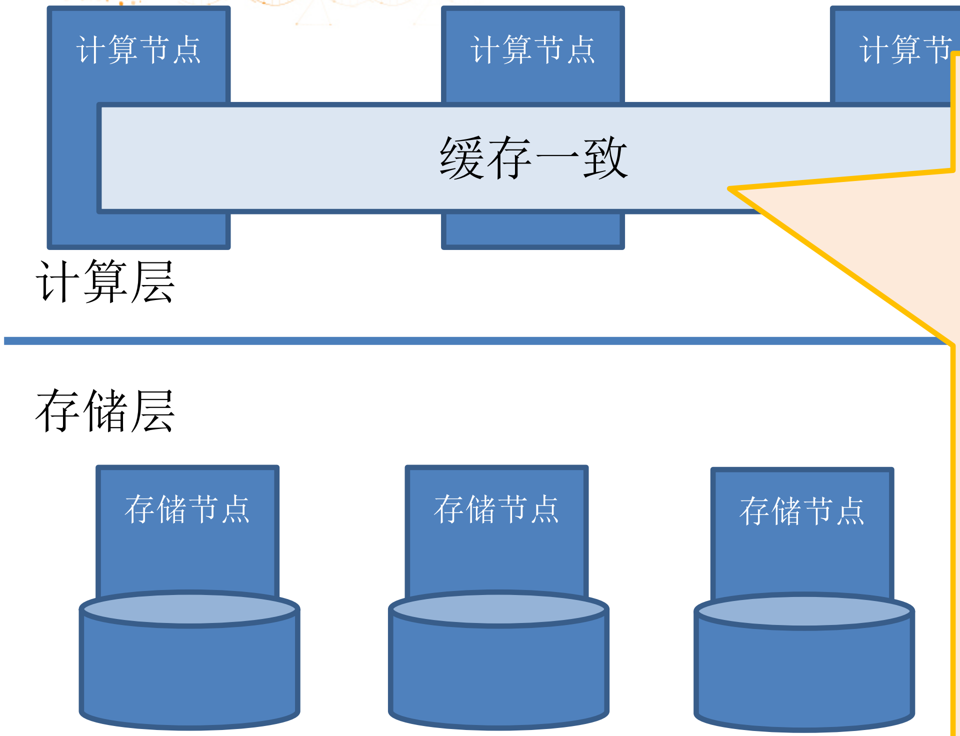
1、计算与存储资源分离
2、数据分片存储
3、分片的多副本

计算与存储分离、可计算存储架构



1、RDMA低延迟技术
2、可计算存储（存储节点可参于where条件运算，过滤无关数据，具体参见Exadata Storage Index和Smart Scan）
3、存储层的透明压缩

Oracle的计算与存储分离、可计算存储架构



目前最大的云计算厂商Amazon，Aurora，还没有实现不同计算节点缓存的统一。当一个计算节点宕机，另一个计算节点要接管数据资源时，必然有较大的延时。

多主可写与更高的可用性，未来是云数据库发展方向之一。

Oracle在20年前，已经实现了跨节点的缓存一致，在此基础上实现了多主可写与更高的可用性，至今已经发展20年，累积了无数宝贵的经验。

谢谢大家　欢迎讨论

VAGE微信

THANKS