Oracle RAC资源管理算法与Cache-Fusion实现浅析

作者：何登成
微博：@何_登成
个人主页：点我

Alibaba.com

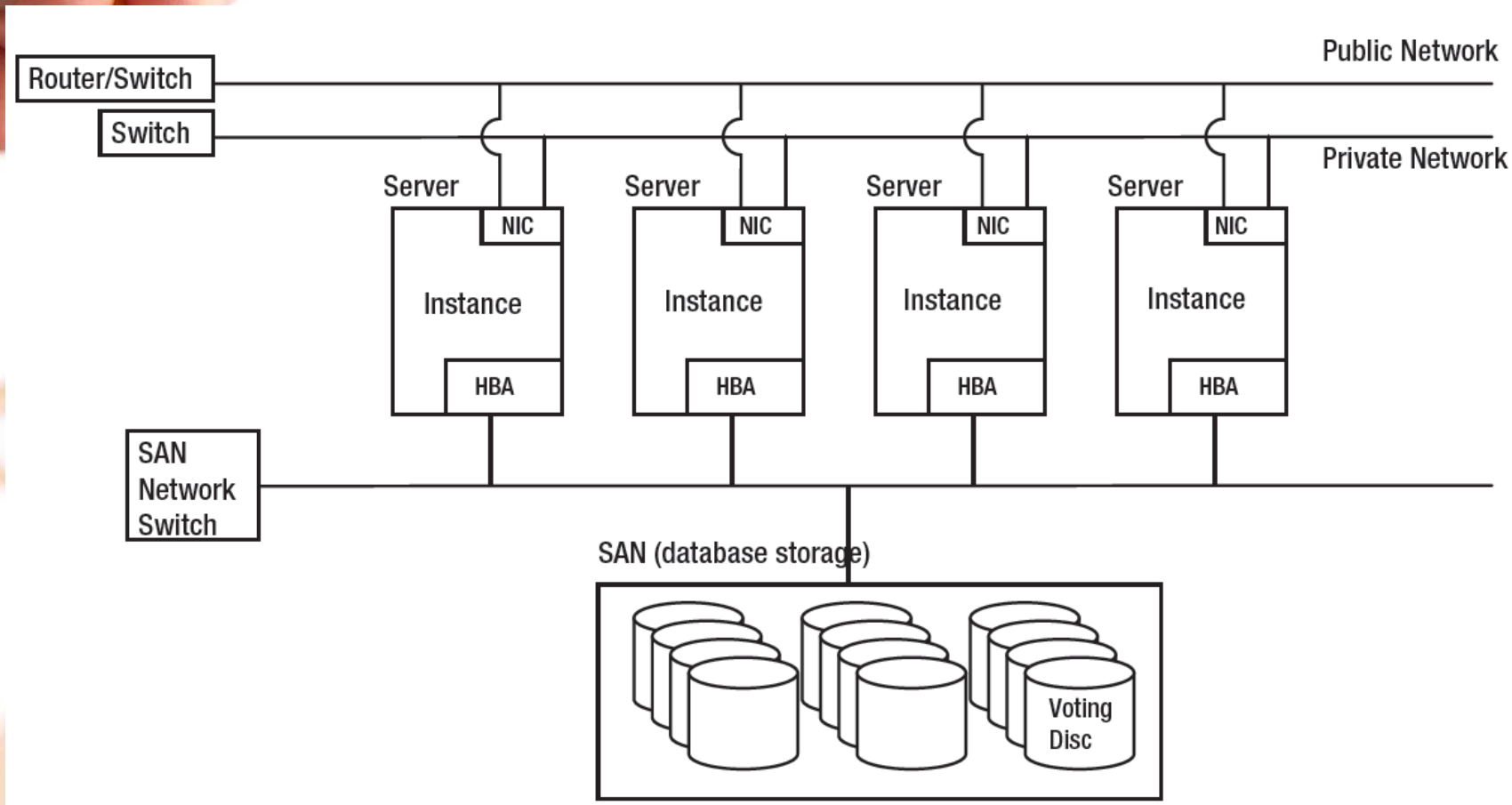
Outline

- 背景知识
  - RAC架构
  - RAC的功能
  - RAC脑裂检测
- RAC通讯算法
- RAC资源分配算法
- Cache Fusion
  - 名称简介
  - R/R
  - W/R
  - W/W
- Cache Fusion恢复

背景知识——名词解释

- **RAC**
  - Oracle Real Application Cluster
  - 由一台或两台以上计算机加共享存储设备构成的一个数据库集群
  - RAC的前身是Oracle 8i OPS
    - Oracle Parallel Server(Oracle并行服务器)
- **PCM**
  - Parallel Cache Management
  - 并行缓存管理(Data Cache)

背景知识——RAC架构

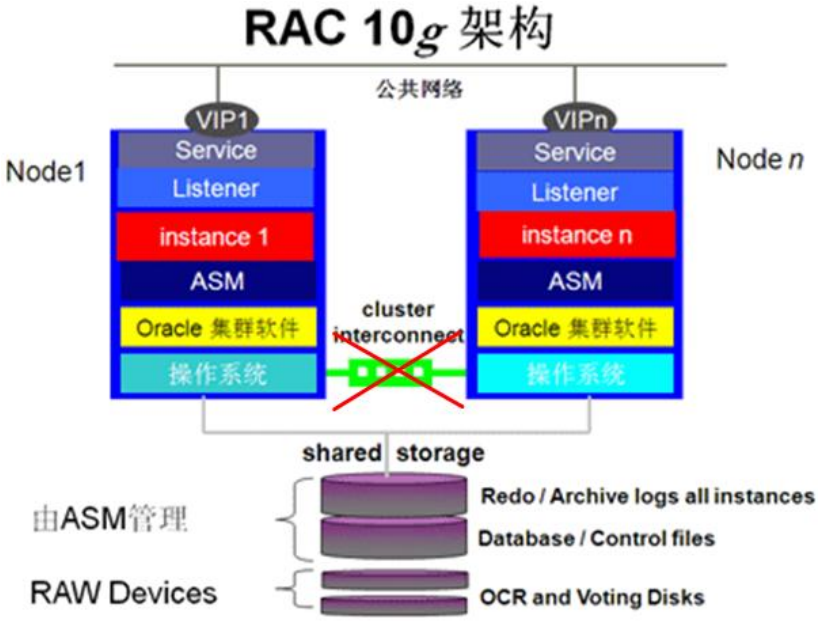

Router/Switch
Switch
Public Network
Private Network
Server
NIC
Instance
HBA
Server
NIC
Instance
HBA
Server
NIC
Instance
HBA
Server
NIC
Instance
HBA
SAN
Network
Switch
SAN (database storage)
Voting
Disc

背景知识——RAC功能

- RAC设计目标
  - **高可用(High Availability)**
    - 任何一个Oracle Instance失败，RAC中的其他Instances会在一定时间内自动对失败节点进行恢复，对前端应用透明。
  - **可扩展(Scalability)**
    - 提升响应时间
      - 增加Oracle Instance
      - 每个Instance有自己的LGWR进程与日志文件
    - 提升吞吐率
      - 增加并行度

背景知识——脑裂检测

- 何谓脑裂？
  - RAC集群中的Instances相互之间无法通讯。每一个Instances都认为自己是活着的，其余的Instances已经死亡，此时产生RAC集群的脑裂。



- 脑裂危害
  - 此时如果各Instances仍旧尝试操作共享存储，就会导致共享存储数据损坏。

背景知识——脑裂检测(续)

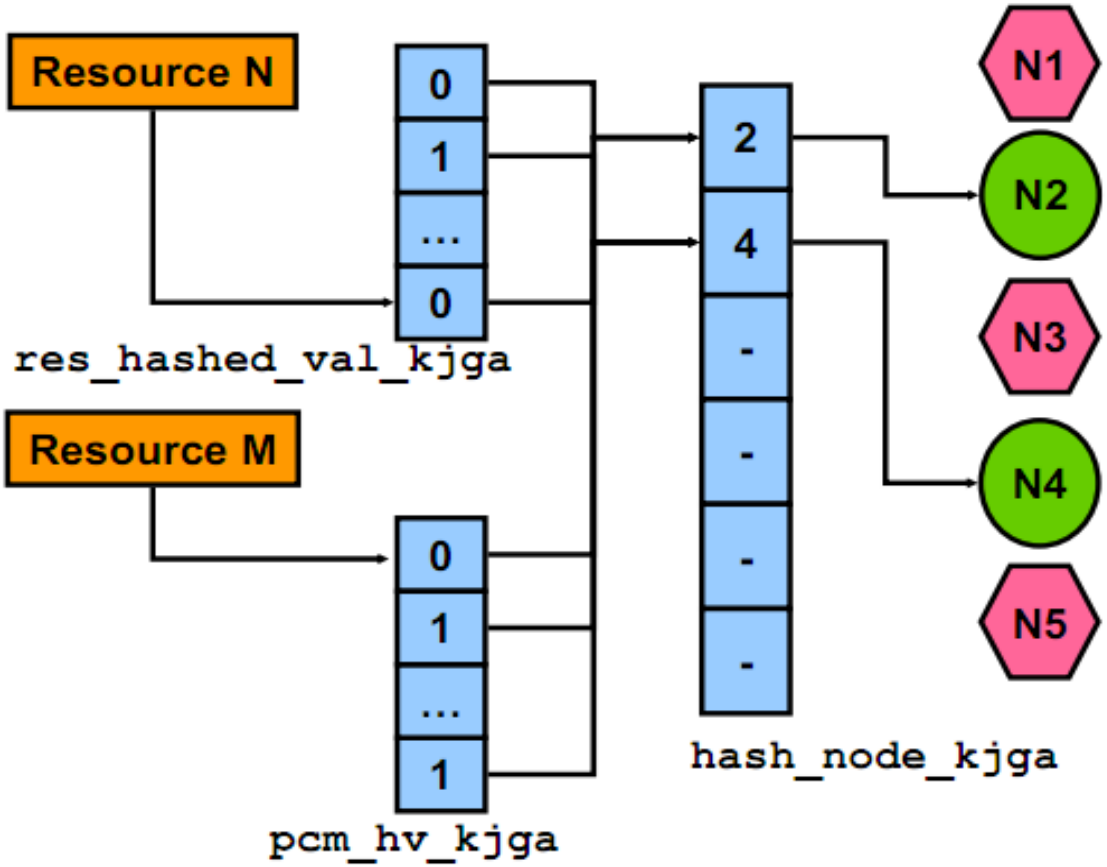
- 脑裂检测
  - **操作系统层面**
    - 所有Oracle Instances每秒写Voting Disk(一个Count计数)
    - 同时检测其他Instances的Voting Disk区域，若Count延迟，则提出对应的节点
    - 若Voting Disk更新均未延迟，但是Instances之间的网络通讯失败，则抢夺Voting Disk的控制权。超过1/2 Instances数量的子集群存活
  - **Oracle层面**
    - Network Heartbeat(LMON Process)
    - File-based Heartbeat(CKPT Process)
      - Control File，CKPT进程每3S更新一次信息

RAC通讯算法

- 注意点（我的理解，方便以后大家阅读）
  - 消息编码
    - AST： 锁持有者发送给锁请求者的消息，用于唤醒等待中的请求者
    - BAST： 锁请求者（或者是master节点）发送给锁持有者的消息，表名当前请求锁被阻塞
  - 主要类型
    - 同步消息：锁请求，等
    - 异步消息：获取锁之后的响应，等
  - 操作实现：
    - 队列（queue）： 所有的异步消息都会进入队列，等待调度发送
    - 非队列： 同步消息一般而言不会进入队列，除非遭遇流控
  - 消息流控
    - 令牌（Ticket）：每个节点都会持有一定数量的令牌。
    - 使用：锁请求消息发送时，需要消耗令牌数；当请求消息的响应从master节点返回，则增加令牌数。
    - 流控：如果需要令牌的消息发送不能获得令牌，则需要进入队列，等待其他节点返回令牌之后，才能继续发送消息。

RAC资源分配算法

- 结构图



RAC资源分配算法

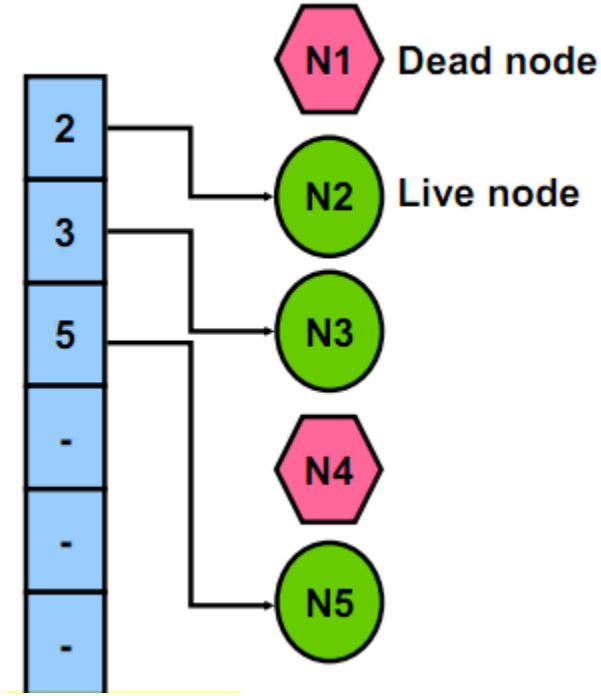
- 概念
  - 物理节点（Physical node）
    - 物理节点，就是实际的rac中的数据库instance
  - 逻辑节点（Logical node）
    - 物理节点在rac内部的索引方式。逻辑节点是一个hash链表结构，链表中的每一项，保存的是物理节点号。Hash_node_kjga
  - 桶（Buckets）
    - 完成资源（PCM/Non-PCM）到逻辑节点的映射。桶也是hash链表结构，链表中的每一项，保存的是逻辑节点号。Res_hashed_val_kjga：non-pcm；pcm_hv_kjga：pcm
  - 资源（Resource）
    - **PCM**(parallel cache management)：所有的data block，redo block
    - **Non-PCM**：PCM资源之外的所有资源，包括表，数据文件，SCN，事务等需要在全局同步

RAC资源分配算法

- 概念（续）
  - Master节点
    - 所有的资源（PCM/Non-PCM），都是被一个指定的物理节点管理。
    - Master节点，就相当于一个调度中心，当RAC中的一个节点需要访问某一资源时，首先需要访问此资源的master节点，判断资源是否可以获得，是否正在被其他节点独占访问而需要等待，等等。
    - Non-PCM资源，在所有RAC节点中平均分配
    - PCM资源，不是在所有节点中平均分配。简单来说，节点的data buffer越大，其所管理的PCM资源越多
    - 接下来，将详细介绍资源是如何在rac中调度的

RAC资源分配算法

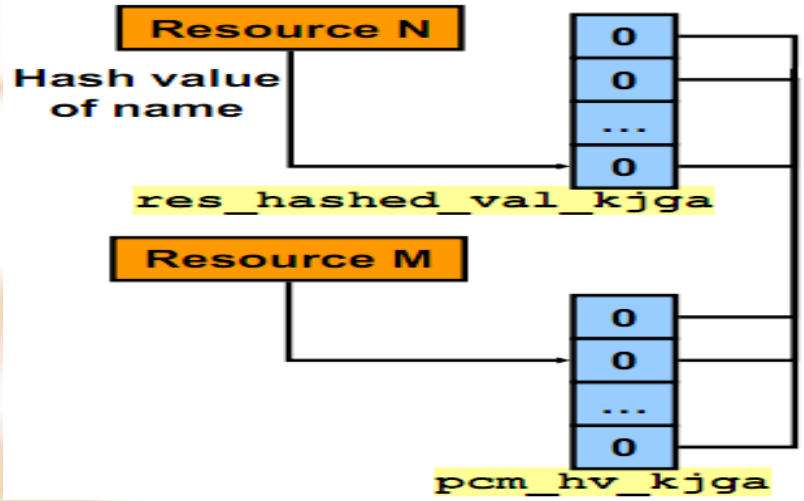
- 物理节点到逻辑节点的映射
  - 左侧为hash_node_kjga，其中的每一项，保存的是物理节点号
  - hash_node_kjga[0]中，必须指向一个物理节点

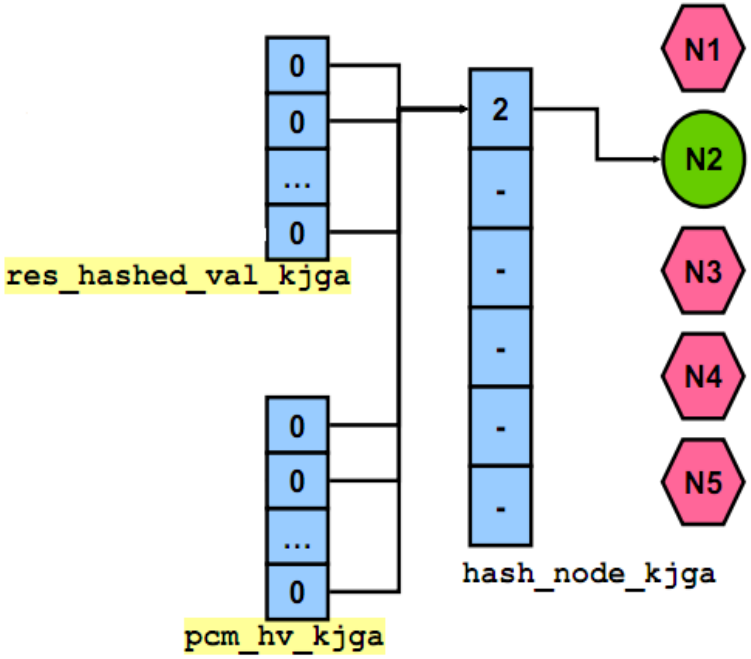


- 逻辑节点链表存在的意义
  - 屏蔽instance加入/离开的影响。逻辑节点链表hash_node_kjga的地址与大小是不变的
  - 方便节点管理，如果需要更改master节点，只需要将hash_node_kjga中对应的值改掉即可，不需要其他操作
  - 逻辑节点，其功能与我们经常使用的vip类似。instance的变化对于rac透明

RAC资源分配算法

- 桶到逻辑节点的映射（右图）
  - 两个桶（分别对应与PCM/Non-PCM）
  - 桶中保存的是对应的逻辑节点的下标
  - 桶的大小为128（两个都是128）
- 资源到桶的映射（下图）
  - 资源（PCM/Non-PCM），通过hash算法，计算出128中的某一个数字，映射到对应桶的位置（pcm资源→pcm_hv_kjga）





RAC资源分配算法

- 资源->桶->逻辑节点，意义所在
  - 为什么不是直接由资源映射到逻辑节点，而要多此一举，在其中添加一层桶？
  - 优势一：固定hash算法

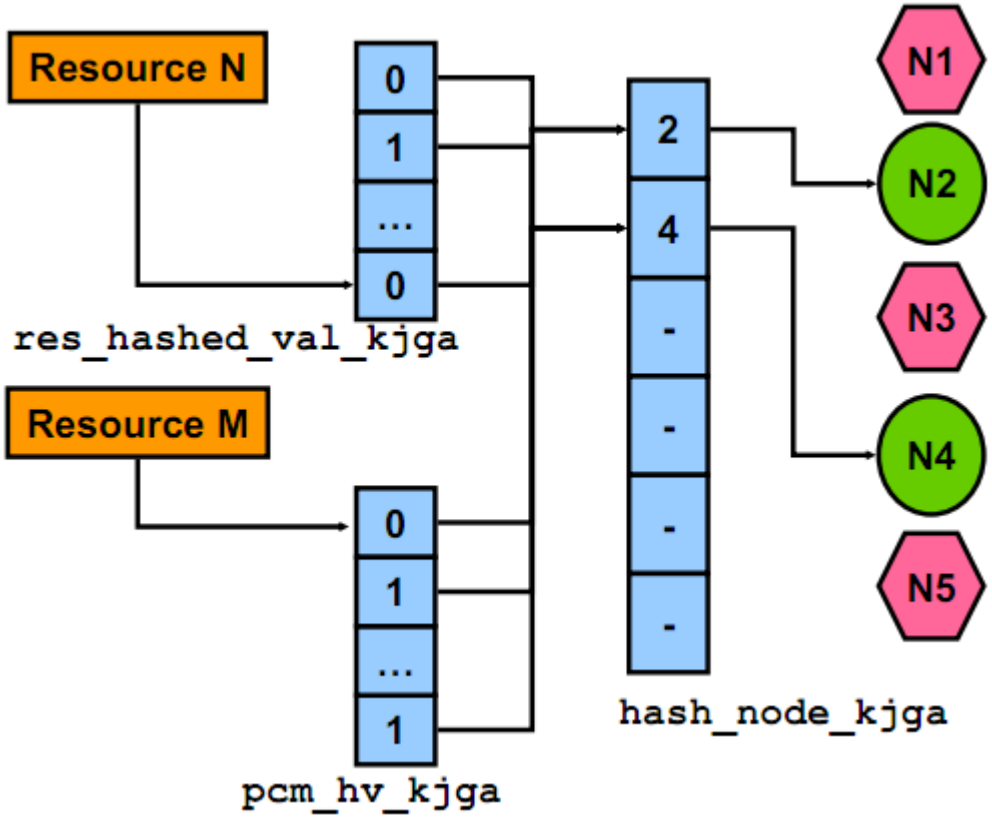
    如果直接由资源映射到逻辑节点，那么随着节点的加入/离开，需要采用新的hash算法，而这个开销是相当大的，所有的资源重计算
  - 优势二：平衡资源分布

    hash算法很难做到平均分布，为了达到平均分布，采用128个桶来达到更好的平衡
  - 优势三：更方便的资源重分配

    在节点加入/离开的过程中，资源可以快速重分配。简单来说，如果节点加入，只需要将pcm_hv_kjga中某几项对应的逻辑节点号更改为新的即可。详细算法，将在后面给出。

RAC资源分配算法

- 资源重分配（Resource Remapping，2节点）



- 注意：桶中的编号
  - Round-robbin方式

RAC资源分配算法

- 资源重分配（多节点，节点加入）
  - 1. 计算每个节点需要管理的buckets数量（pcm，需要加上权重）
  - 2. 将目前管理数量大于 平均数量的 节点，取出其中多余的buckts，设置为未分配
  - 3. 遍历res_hased_val_kjga，pcm_hv_kjga中的每一项，如果当前项未分配，则将分配给最新加入的节点
  - 4. 刷新各节点的配置文件
- 资源重分配（多节点，节点离开）
  - 1. 同上
  - 2. 将res_hased_val_kjga，pcm_hv_kjga中，指向离开节点的所有项，设置为未分配
  - 3. 同上
  - 4. 同上
- 注意点：节点加入/离开，没有涉及到hash函数的变化

Cache Fusion

- 名称简介
  - Cache fusion：数据块（data block），从一个节点，通过网络传输到另外一个节点，其中不需要硬盘的参与（包括读硬盘/写硬盘）
  - 实现方式：cache fusion通过加锁的方式来实现
  - Data block持有模式：rac中的每个节点，对于其data cache中的block，都有持有模式标识。
  - 模式包含三类信息：锁信息，块角色，过去映像（Past Image，pi block）

Cache Fusion

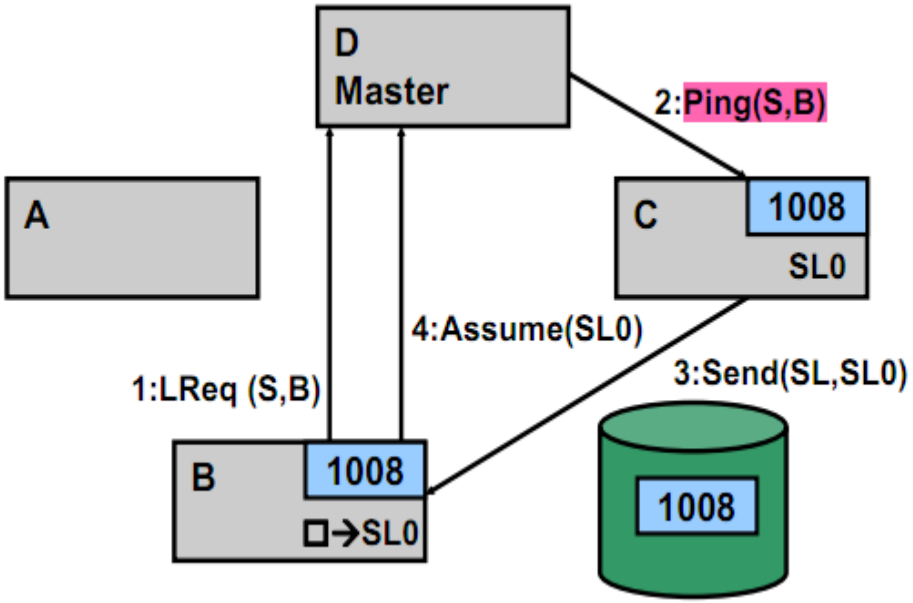
- 块持有模式
  - 锁信息：
    当前节点对于block的持有类型，有三类：只读（S），排他（X），NULL
  - 块角色（role）：
    块角色有两种：局部（L，local）；全局（G，global）
    局部 VS 全局：代表块是局部可脏，还是全局可脏？
  - 过去映像（Past Image）：
    标识当前节点持有的数据块是否为脏块
    取值：0代表不是past image，>0的整数，表示是过去映像，同时，整数大小表示过去映像的新旧程度
    产生：当前节点修改了块，接着其他节点需要读取最新版本/修改块中其他数据，此时产生
- 例如SG0
  - 读访问，全局块（代表脏块是全局的），不是past image块（块未被本节点修改）

Cache Fusion（准备）

- 单节点
  - 读CR块
    - 从硬盘读出当前块，然后通过回滚段构造出需要的数据块。CR块不需要加锁
  - 读当前块
    - 直接从硬盘中读取SCN最新的数据块，数据块的持有模式为：SL0（shared，local，non-pi block）
  - 写当前块
    - 直接从硬盘读出SCN最新的数据块，数据块的持有模式为：XL0（exclusive，local，non-pi block）
- 当这些准备完毕之后，才会有后续的R/R，R/W, W/R, W/W

Cache Fusion

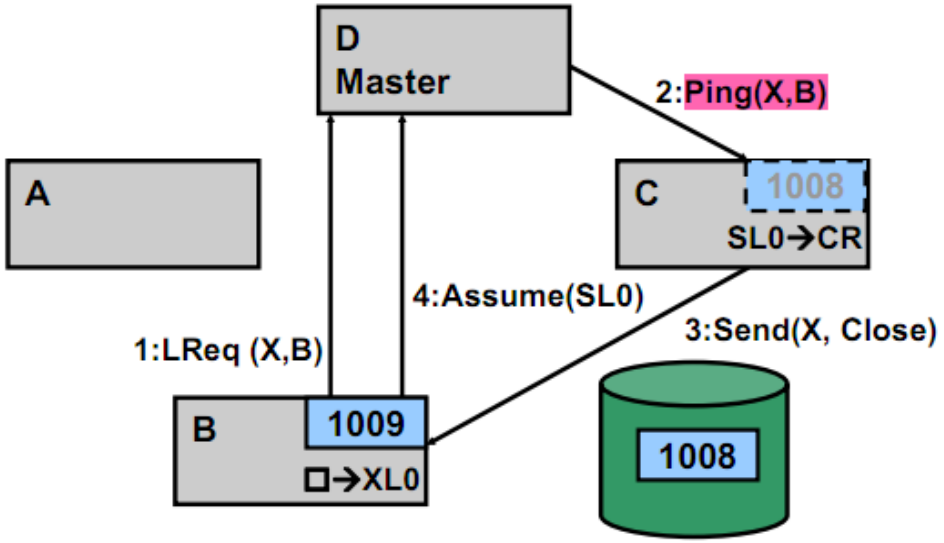
- R/R（节点C持有SL0，节点B想继续读取）
  - 节点B读CR块
    - 节点C通过当前块，构造CR块，传递给节点B
    - 持有模式：节点C: SL0不变；节点B: CR块不参与Cache Fusion



  - 当前节点C持有SL0模式，此时节点B想读block的当前块
    - 节点C将当前块，传递给节点B.
    - 持有模式：节点C: SL0；　　节点B: SL0
    - 节点C，B同时持有块的当前版本

Cache Fusion

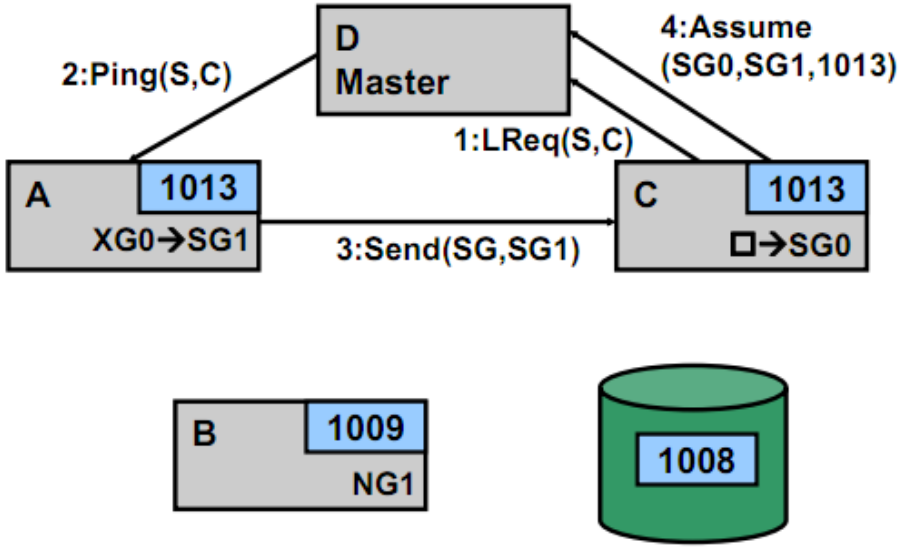
- R/W（节点C持有SL0，节点B想修改此Block）



  - 节点C，将块传递给节点B，
    同时释放锁，模式变化：SL0->CR块，不参与以后的Cache fusion
  - 节点B，收到C传递过来的block，
    模式变化： ->XL0，局部修改模式，无PI block

Cache Fusion

- W/R（节点A持有最新的修改块，节点C想读取）
  - 节点C读取的SCN < 当前块的SCN（CR 读）
    - 节点A构造CR块，传递个节点C
    - 持有模式：A不变；C不持有任何模式，



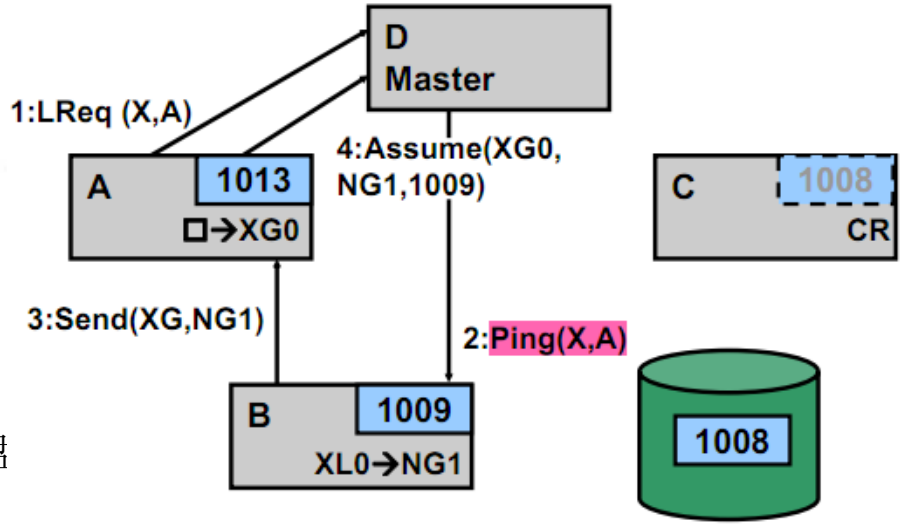
  - 节点C读取的SCN >= 当前块的SCN
    - 节点A将日志回刷
    - 节点A持有模式：XG0 -→ SG1
    - 节点C持有模式：　　-→ SG0
    - 节点A，持有当前块的PI块

Cache Fusion

- W/R优化（节点C读取的SCN >= 当前块的SCN）
  - 不足之处：前页中，节点C的读取，导致节点A的锁模式降档，其实有其不足之处。考虑以下场景：节点C读取，之后节点A又准备修改此Block，那么就会导致新一轮的节点C降档，节点A加X锁的过程
  - 问题产生的本质？节点C需要拿到S锁的原因？（为了下次读取能够沿用此block，而不至于重新从节点A获取）。此处，涉及到一个节点A的更新频率与节点C（或是其他节点）的读取频率之间的一个博弈。读取频率高，A降档更为合适；更新频率高，A不降档，传递CR块给C的方式更为合适。如何优化？
  - 参数：_fairness_threshold (默认值为4)。连续_fairness_threshold次读取，其中没有更新操作发生，则A降档；期间，每次传递的都是CR块，A保持X锁模式。

Cache Fusion

- W/W（节点B持有最新脏块，节点A想修改）



  - 节点B，首先将redo log刷新到磁盘
  - 节点B，将脏块发送给节点A
  - 节点B，持有模式：XL0 -→ NG1（null锁，全局块，同时是第一个pi块）
  - 节点A，收到最新的脏块，持有模式： -→ XG0

Cache Fusion

- 数据块写回磁盘
  - 持有数据块版本的任何一个节点，都有可能激活DBWR，将其中的脏块写回磁盘。针对其所持有块的模式不同，写回操作也不相同。
    - SL0：　非脏块，不用回写
    - SG0:　全局有SGi，past image，但是我持有的一定是最新块映像，直接回写，或者是让持有最大past image号的 节点回写均可
    - Ngi：　全局past image块，此时需要找到rac中，持有X锁的节点回刷，（若无X锁节点，则定位到最大的past image号节点，回刷block）
    - XL0:　当前一定是最新脏块，刷回；
    - Xgi：　当前一定是最新脏块，刷回；同时，其余节点的past iamge块，都置为CR块。
  - 所有的回刷block操作，都会将全局中，所有的past iamge，置为cr块。

Cache Fusion

- 总结
  - 1. 为什么需要对block加锁？
    - 为了判断当前block是否在cache中，在哪个节点的cache中。在单实例oracle中，block是不需要加锁的，因为判断block是否在内存中，可以通过hash快速查找；而单实例更没有节点选择的问题。
  - 2. 为什么需要S模式？
    - 在block被读入内存时，需要对此block加S锁，主要有三个作用：1）说明block目前在内存中；2）说明目前是以共享只读形式访问的；3）S锁与X锁冲突，当一个节点需要修改block时，可以一次性找到所有的S锁，并释放锁，drop他们的block。

Cache Fusion

- 总结（续）
  - 3. 为什么需要X锁？
    - X锁时排它锁，说明在同一时刻，RAC中，只有一个节点可以修改block。
  - 4. 为什么需要N锁？
    - 首先，只有past image需要加N锁，其意义在于：1）表明该block的身份，是PI block。2）N锁不与X锁冲突，因此可以共存。3）在任何一个节点做checkpoint时，其N锁对应的pi block，需要发送消息给master节点，找到对应的current block，并写入磁盘

Cache Fusion恢复

- 恢复起点
  - 最新的PI 块（past image block）
  - 硬盘上的映像（无PI块）
- Instance vs Crash vs Block Recovery
  - Instance Recovery：SMON
  - Crash Recovery：Foreground Process
  - Block Recovery：PMON
- 日志合并
  - 在实际恢复前，失败节点的日志，需要首先被合并

Cache Fusion恢复（cont.）

- 两次日志读取（two-pass log read）
  - First-pass log read
    - 构造恢复集（Recovery Set）
    - Block Written Records（BWR）
  - Recovery claim locking
  - Second-pass redo application

Cache Fusion恢复（cont.）

- First-Pass Log Read（recovery set vs bwr）
  - 读日志，将修改的block放入Recovery Set
  - Recovery Set: Hash Table结构保存。包含了block的最新与最旧的版本（SCN，Seq#）。
  - BWR： Block Writen Records。顾名思义，记录block被刷回硬盘的信息（于redo buffer中）。
  - BWR功能：
    - BWR version > latest PI，recovery no need
    - 剪裁Recovery Set

Cache Fusion恢复（cont.）

- Recovery Claim Locks
  - 恢复节点（recovering node）的smon进程，对于Recovery set中的每一个block，发送RecoveryClaimLock消息至其master节点。
  - 如果一个block对应的主节点down，那么在做完remastering前，recovery等待。
  - 功能：
    - 1. 重构失败前的锁信息
    - 2. recovering节点获得恢复block的最新版本

Cache Fusion恢复（cont.）

- Lock role的功能：
  - 1. 正常运行时
    - 用来检测，定位block的位置，及状态
  - 2. 实例恢复时
    - 用于判断recovery set中的block，是否需要做cache recovery（rolling forward），大部分情况下，cache recovery是不需要的，因为通过lockrole，就可以判断出，此时在recovering节点/其他节点上的block，就是最新的block

Cache Fusion恢复（cont.）

- Recovery Claim Locks
  - Recovery buffer
    - 存放获得的最新的需要恢复的block，buffer pool。（50%，冷端）
  - RecoveryDoneClaiming消息
    - Recovering节点获得所有Recovery set中block的锁之后，发送此消息
  - 访问权限控制
    - 消息发送之后，Rac解除全冻结状态，可以提供给前台访问（但是参与恢复的block除外）

Cache Fusion恢复（cont.）

- Second-Pass Log Read
  - 操作：对recovery set中的block，应用日志的阶段，cache recovery （rolling forward）。恢复完成之后，脏块被DBWR进程写回磁盘，当前块的lockrole从XO（恢复状态）变为XL，其他节点上的PI block 被invalidate。相当于checkpoint
  - 在block恢复完成之后，清除恢复标记，此时才能够响应应用对于此block的BAST请求

Cache Fusion恢复（cont.）

- 大恢复集 vs 部分实例恢复锁模式
  - recovering instance支持的recovery set大小与其data cache相关
  - 当构造出的recovery set大于recovering instance所能支持的最大大小，smon进程就转化为partial ir lock mode

Cache Fusion恢复（cont.）

- partial IR Lock Mode处理方式：
  - 简单来说，就是分批处理，将recovery set转换为多个M blocks大小的recovery lists，按照first-dirty SCN的顺序
  - 在第一组recovery list完成之后，IDLM并不能发送RecoveryDoneClaiming，因为不是所有的block都恢复完成
  - Recovery lists VS Reused lists ？

Cache Fusion恢复（cont.）

- 并行处理
  - Lazy remastering vs first-pass log read
- PCM不可用期
  - IDLM主节点丢弃dead节点持有的锁资源
  - SMON进程发送RecoveryDoneClaiming消息
- 例外
  - 不需要IDLM参与的操作，可以继续处理。例如：当前block上的lockrole为XL，那则可以继续处理。

参考资料

- *Oracle*. DSI408: Real Application Clusters Internals
- *Oracle*. Oracle8i Parallel Server
- *Jonathan Lewis*. Oracle Core: Essential Internals for DBAs and Developers

THE END